JSpider – Version 1.4.2 (8.05.2019)

JavaScript-Bibliothek auf Basis von ECMAScript 2015 (ES6)

Dokumentation der Programmierschnittstelle (API) mit Einstiegsbeispiel

Globale Konstanten – Funktionen – Objekterweiterungen

`"use strict"`

Sollte auf jeden Fall eingestellt sein. Es verhindert z.B. versehentliche globale Variablendeklarationen

Globale Konstanten (pseudo-private)

Hinweis: Konstanten, die mit doppeltem Unterstrich beginnen, sind zur „privaten" Nutzung vorgesehen. Ein direkter Zugriff auf diese Datenstrukturen von außen ist nicht empfehlenswert.

`const DEG2RAD = 0.0174533`

Konstante zur Umrechung von Bogen- in Gradmaß und umgekehrt.

`const __startTime = new Map()`

Sammlung der Startzeiten bei Verwendung unterschiedlicher Zeitmess-Timer ↗ setTimer() | getTimer()
key: timerName oder -Number value: Startzeit

`const __worlds = new Map()`

Falls mehrere World-Objekte (Canvas) erzeugt werden, werden diese über die HTML-ID hier verwaltet.
key: id value: World-Objekt

`const __handlers = new Map()`

Sammlung aller mit der listen()-Methode scharf gemachten Event-Handler, und zwar sowohl für World-Objekte, als auch für Spider-Objekte gemeinsam. Ist erforderlich, damit die Zuordnung Event → Event-Handler auch wieder gelöst werden kann. key: event-ID value: handler

`const __images = new Map()`

Sammlung aller mittels registerImage() registrierten Bilddateien.
Breite und Höhe des Bildes werden beim Registrieren ermittelt und mit in der Map gespeichert.
key: Name des Bildes (str) value: {src: Bild (Dateiname|base64-img), width:Breite, height:Höhe}

`const __builtinImages = new Set()`

Enthält die eingebauten Spider-Images, die als base64-Image direkt im Quelltext integriert sind

Farben

Alle Farbzuweisungen von World- oder Spiderobjekten verlaufen implizit über Setter und sind „managed".
Möglich sind folgende Notationen (immer als Zeichenketten), wobei alle Farbangaben bis auf „random" HTML-konform sind und wie üblich mittels CSS zugewiesen werden können.

"#3F84A2"	Übliche hexadezimale HTML-Notation
"rgb(23,232,187)"	HTML-konforme dezimale Notation
"rgba(105,34,87,0.65)"	HTML-konforme dezimale Notation mit Alphakanal (Transparenz von 0.0 bis 1.0)
""	Transparenz (leere Zeichenkette)
"random"	Zufällige Farbe (entspricht einem Aufruf von randomColor())

Globale Funktionen

`getElementById(id) → DOM-Element`

Referenz auf document.getElementById(id) zur besseren Lesbarkeit und Vereinfachung.
Erzeugt ein JavaScript-DOM-Objekt aus einem HTML-Objekt.
id ist die einem HTML-Tag über das id-Attribut zugewiesene (eindeutige) Kennung.

`write(txt)`

Referenz auf document.write(txt) zur besseren Lesbarkeit und Vereinfachung.
Gibt einen Text im Browserfenster aus an der aktuellen Stelle im DOM-Baum.

`writeln(txt)`

Wie write(txt) mit zusätzlichem Zeilenumbruch.

`read(txt) → String | Number | Array | false`

Ruft die JavaScript-Funktion prompt(txt) auf, liefert aber einen aufbereiteten Rückgabewert:

false Bei leerer Eingabe oder Schließen des Eingabefensters über Abbruch-Button
Number Wenn die Eingabe als Zahl aufgefasst werden kann, wird eine Zahl zurückgeliefert
String Wenn die Eingabe nicht als Zahl aufgefasst werden kann, wird eine Zeichenkette zurückgeliefert
Array Enthält die Eingabe mindestens ein Komma, dann wird die Eingaba an allen Kommastellen getrennt und ein Array der Einzelteile zurückgeliefert. Für die Einzelteile gilt jeweils das vorher gesagt: Number oder String.

`round(x [,n=0]) → Number | NaN`

Komfortfunktion auf Basis und als Ersatz für Math.round().
Rundet x auf n Nachkommastellen bzw. bei fehlendem n auf einen ganzzahligen Wert und liefert diesen Wert zurück.

`trunc(x) → Number (int) | NaN`

Referenz auf Math.trunc: Schneidet die Nachkommastellen von x ab und liefert diesen Wert zurück..

`getDays(dateStr) → Number (int) | undefined`

Erwartet ein Datum als Zeichenkette in der Schreibweise "tt.mm.yyyy" und liefert die Anzahl der Tage vom aktuellen Datum bis zu diesem Datum. Liegt das Datum in der Vergangenheit, ist der Wert negativ.

`getWeekday(dateStr) → String | undefined`

Erwartet ein Datum als Zeichenkette in der Schreibweise "tt.mm.yyyy" und liefert den zugehörigen Wochentag (englisch).

`getDate() → dateStr`

Liefert das aktuelle Datum in der Schreibweise "tt.mm.yyyy".

`getTime() → timeStr`

Liefert die aktuelle Uhrzeit in der Schreibweise "hh:mm:ss".

`setTimer([n=0|txt])`

Drückt den Startknopf der Stoppuhr. Optional: verschiedene Timer parallel laufen lassen; dann Namen geben.
Verwendet zum Speichern der Startzeit(en) die globale Variable __startTime.

`geTimer([n=0|txt]) → Number (int)`

Liefert die seit dem Aufruf von setTimer() [ggf. identifiziert/zugeordnet durch n|txt] vergangene Zeit in Millisekunden.

`randomNumber(a,b) → Number (int) | NaN`

Liefert eine zufällige ganze Zahl aus dem Intervall [a; b] bzw. NaN, falls nicht a ≤ b ist.

`randomChar(text) → String (char) | ""`

Liefert ein zufälliges Zeichen aus der Zeichenkette text bzw. eine leere Zeichenkette, falls text keine Zeichen enthält.

`randomColor() → String (#rrggbb)`

Liefert eine zufällige Farbe als RGB-Code in Hexadezimaler HTML-Notation zurück.

`min2max(a,b) → a-b`
`max2min(a,b) → b-a`

Hilfsfunktionen, mit denen es möglich ist, Arrays mit Zahlen korrekt zu sortieren (die builtin-Methode sort() sortiert nämlich IMMER nach Unicode, so dass Zahlen grundsätzlich falsch sortiert werden.
Verwendung: array.sort(min2max) bzw. array.sort(max2min). Sortiert jeweils die Elemente von array in-place.
Aber Vorsicht: Beim Sortieren von Zeichenketten muss sort() ohne Argument aufgerufen werden!

`registerImage(name,filename)`

Registriert ein Bild mit dem Dateinamen filename unter dem selbst wählbaren Namen name (String). Ist dieser Name schon vergeben, wird dies mit einer Fehlermeldung quittiert. Wird nur für die Spider-Images benötigt.

`isRegisteredImage(name) → true | false`

Prüft, ob ein Bild mit Namen name bereits registriert wurde.

Globale Objekterweiterungen

`array.toString() → String`

Liefert eine pretty-print Darstellung eines Arrays, wobei die Typen der Objekte sichbar bleiben, d.h. Zeichenketten erscheinen in Anführungszeichen. Das ist per default nicht so, wenn ein Array etwa über write(array) angezeigt wird.

`array.shuffle()`

Mischt die Elemente des Arrays in-place mit Hilfe des Fisher-Yates-Algorithmus.

`map.toString() → String`

Liefert eine pretty-print Darstellung einer Map: {key:value; key:value; …}
Maps haben per default keine eigene toString-Methode, so dass write(map) nicht den Inhalt einer map anzeigt!

`map.save(name)`

Speichert den Inhalt der Map im Browsercache (localStorage) unter dem Namen name ab. So lange der Browsercache nicht gelöscht wird, bleibt die gespeicherte Map über die Lebenszeit der Sitzung hinaus erhalten. Die Einstellung „Cache beim Beenden leeren“ verhindert z.B. eine sinnvolle Nutzung dieser Speichermöglichkeit.

`map.load(name) → true | false`

Liest den Inhalt der Map, die im Browsercache unter dem Namen name abgelegt wurde. Ist ein solcher Eintrag im Browsercache nicht vorhanden, ist der Rückgabewert false, ansonsten true.

class World

Ein World-Objekt wird als HTML-Canvas in die Seite eingebunden. Möglich ist es, dieses Canvas zuvor per HTML selbst im Dokument zu platzieren und (mit CSS) zu formatieren und dann per ID-Übergabe an den Konstruktor die Zuordnung vorzunehmen, oder eben auch nicht. In dem Fall wird ein entsprechendes Canvas inkl. generierter ID erzeugt und dem DOM sowie der HTML-Seite hinzugefügt.

World::Konstruktor

`web = new World(id)`

Erzeugt ein World-Objekt und verwendet als Darstellungsbereich ein zuvor im HTML-Teil erstelltes Canvas mit dessen Einstellungen, das über sein id-Attribut an den Konstruktor übergeben wird.
ACHTUNG: Die Maße des Canvas dürfen nicht über CSS festgelegt werden, sondern müssen über die Canvas-Attribute width und height festgelegt werden, weil es sonst zu Fehldarstellungen kommt.
Beispiel (HTML): <canvas id="welt" width="500" height="200" style="border:solid 2px green">

`web = new World(width,height)`

Erzeugt ein World-Objekt als Canvas an der aktuellen Stelle im DOM bzw. sichtbar auf der HTML-Seite mit den Maßen width und height (in px). Es wird keine Verknüpfung zu einem bereits erstellten HTML-Canvas (über eine id) hergestellt, sondern ein neues Canvas-Element wird erzeugt.

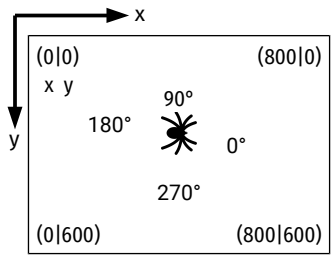
`web = new World()`

Wie die vorherige Variante, allerdings ohne Festlegung von width und height. Die Canvasgröße ist dann 800 px x 600 px.

Hinweise: Zwar ist es möglich, über web.canvas.width bzw. web.canvas.height die Größe des dargestellten Canvas zur Laufzeit des Programms zu verändern, allerdings sollte man davon absehen, weil das zu unerwünschten Seiteneffekten führen kann. Empfehlenswert ist es, die Größe eines World-Objekts einmalig beim Erstellen des Objekts festzulegen und danach so zu belassen.

World::Koordinatensystem

World-Objekte besitzen ein eingebautes Koordinatensystem, das sich an dem graphischer Systeme orientiert (so auch in HTML/CSS eingesetzt wird) und vom in der Mathematik üblichen Koordinatensystem unterscheidet – siehe Abbildung:



World::Datenattribute (public | managed access)

`.name (ro)`

Entspricht der ID des Canvas, falls beim Konstruktoraufruf eine ID übergeben wurde.
Ansonsten lautet der Namen „web-<nr>“, wobei nr eine laufenden Nummer beginnend mit 1 ist.

`.color`

Hintergrundfarbe des Canvas als Zeichenkette. Kann gelesen oder neu gesetzt werden.
Zulässige Farbangaben ↗ Abschnitt „Farben“.

`.image`

Hintergrundbild des Canvas (Dateiname als Zeichenkette). Kann gelesen oder neu gesetzt werden.
Falls die tatsächliche Größe dieser Bilddatei nicht mit den Abmessungen des Canvas überstimmt, wird es unter Beibehaltung seines Seitenverhältnisses so skaliert (verkleinert oder vergrößert), dass entweder Länge oder Breite (oder eben beides, wenn das Seitenverhältnis des Bildes mit dem des Canvas übereinstimmt) mit dem Canvas übereinstimmen. Die linke obere Ecke des Bildes liegt dabei in der linken oberen Ecke des Canvas. Eine Wiederholung (Kachelung) oder Skalierung unter Missachtung des Seitenverhältnisses des Bildes wird nicht durchgeführt.

`.spiders (ro)`

Array mit allen existierenden Spider-Objekten, die innerhalb dieses World-Objekts erzeugt wurden. Das Array enthält die Spider-Objekte in der Reihenfolge, in der sie erzeugt wurden, und ist eine flache Kopie. Änderungen an den Spider-Objekten wirken sich also auf die tatsächlichen Objekte aus, Änderungen am Array hingegen (etwa das Entfernen eines Spider-Objekts oder das Ändern der Reihenfolge) bleiben hingegen ohne Effekt. Durch Iteration über dieses Array erreicht man alle existierenden Spider-Objekte, auch „anonyme“, die keinem Bezeichner (mehr) zugewiesen sind.

World::Methoden

`.clear([x1,y1,x2,y2])`

Löscht alle Zeichnungen in einem rechteckigen Bereich des Canvas: (x1|y1) ist die linke obere, (x2|y2) die rechte untere Ecke dieses Bereichs. Aufruf ohne Argumente löscht das gesamte Canvas.
Achtung: Entfernt werden nur Zeichnungen der Spider! Die Spider-Objekte selbst und auch deren Abdrücke (↗ *.stamp*()) bleiben erhalten, da sie sich „oberhalb“ der Zeichenfläche in einer zweiten Ebene befinden. Zum „Entfernen“ eines Spider-Abdrucks dient die Spider-Methode *unstamp*() und Spider-Objekte kann man unsichtbar machen: visible = false

`.listen(event,handler)`

Lässt das World-Objekt auf bestimmte Ereignisse (↗ Kapitel „Ereignisse und Ereignisbehandlung“) lauschen. Dazu wird der Name des gewünschten Ereignisses einem Funktionsnamen (Handler) zugeordnet. Diese Funktion wird aufgerufen, sobald das Ereignis eintritt.
Beachte: Auf Events, die während der Ausführung einer laufenden Spider-Aktion ausgelöst werden, wird nicht reagiert. Erst bei „ruhendem System“, d.h. mit Erreichen von world.animate() lauscht das World-Objekt auf Ereignisse.
Einem bestimmten Ereignis kann immer nur *ein* Handler zugeordnet werden. Will man mehrere Handler an ein Ereignis binden, dann kann man als Handler eine „Containerfunktion“ benutzen, die der Reihe nach die einzelnen Handler aufruft.
Um eine Bindung zwischen Event und Handler zu lösen, wird das entsprechende Event mit *false* als zweitem Argument (statt eines Funktionsnamens) aufgerufen.

`.animate()`

DIE zentrale Funktion, die die Animation-Loop in Gang bringt und für den Ablauf des gesamten Programms zuständig ist. Muss (und darf auch nur) einmalig am Ende des Hauptprogramms aufgerufen werden.
Innerhalb von Event-Handlern darf *animate*() nicht erneut aufgerufen werden, weil am Ende der Ausführung einer mit *listen*() als Event-Handler registrierten Funktion der Aufruf von *animate*() automatisch (implizit) erfolgt.
In der Praxis funktioniert die Ausführung von Spider-basierten Programmen so, dass alle world- und spider-Methoden nicht direkt ausgeführt werden, sondern zunächst in einer command-queue gesammelt werden. Beim Aufruf von animate() wird die Warteschlange dann abgearbeitet. Nur so ist eine animierte Darstellung bestimmter Abläufe möglich.

class Spider

Spider::Konstruktor

`spider = new Spider(world)`

Erzeugt ein Spider-Objekt innerhalb eines World-Objekts mit dem Objektbezeichner world.
Bewirkt, dass ein noch unsichtbares Spider-Objekt genau in der Mitte des world-Canvas positioniert wird und nach rechts (= Osten = 0°) ausgerichtet ist. Dieses Spider-Objekt hat – wenn man es sichtbar macht – die Gestalt einer Spinne.

Spider::Datenattribute (public | managed access)

Hinweis: Bei Werten mit Zuweisungsmöglichkeit ist der Default-Wert in eckigen Klammern notiert.

`.name (ro)`

Eindeutiger Name des Spider-Objekts (nicht zu verwechseln mit dem Namen des Bezeichners, dem das Objekt zugewiesen wird; dieser kann sich durch Zuweisung ändern, der Name nicht), der bei der Instanzbildung automatisch generiert wird. Aufbau: "spy-<nr>", wobei <nr> die ID des Spider-Objekts ist, die wiederum der laufenden Nummer der Erzeugung von Spider-Objekten entspricht.

`.x (ro)` **`.y (ro)`**

Koordinaten (des Mittelpunktes der Bounding-Box) einer Spider-Shape innerhalb des World-Koordinatensystems.

`.angle (ro)`

Winkel (Ausrichtung) der Spider. Dabei entspricht 0° der Ausrichtung nach rechts (Osten). Das Winkelmaß wird in Gradmaß angegeben, und zwar im mathematisch üblichen Sinn (d.h. gegen den Uhrzeigersinn).

`.speed ← Number (1..100) [4]`

Geschwindigkeit der Spider bei Bewegungen und Drehungen. Je größer der Wert, desto schneller die Bewegung.

`.width ← Number [50]` **`.height ← Number [50]`**

Breite bzw. Höhe der Spider-Shape in px, und zwar ggf. einschließlich eines mit border gesetzten Randes (d.h. es ist immer die Brutto-Breite. Wird der Rand breiter gemacht, verringert sich der Innenbereich, aber nicht die Außenmaße).

`.color ← String ["white"]`

Setzt die (Hintergrund-)Farbe der Spider-Shape. Eingefärbt wird ein Rechteck mit den Spider-Maßen width und height. Zu Farbangaben ↗ Hinweise oben.

`.border ← String [""]`

Entspricht dem CSS-Attribut border (ist nur ein Wrapper – der zugewiesene Wert wird ohne Prüfung direkt durchgereicht und als CSS-Style gesetzt). Beispiel: "solid 4px red“. Weitere Möglichkeiten: ↗ CSS-Dokumentation.

`.shadow ← String [""]`

Entspricht dem CSS-Attribut box-shadow (ist nur ein Wrapper – der zugewiesene Wert wird ohne Prüfung direkt durchgereicht und als CSS-Style gesetzt). Beispiel: "4px 6px 3px blue". Weitere Möglichkeiten: ↗ CSS-Dokumentation.

`.edges ← String [""]`

Entspricht dem CSS-Attribut border-radius (ist nur ein Wrapper – der zugewiesene Wert wird ohne Prüfung direkt durchgereicht und als CSS-Style gesetzt). Beispiel: "5px". Weitere Möglichkeiten: ↗ CSS-Dokumentation.

`.cursor ← String ["default"]`

Entspricht dem CSS-Attribut cursor (ist nur ein Wrapper – der zugewiesene Wert wird ohne Prüfung direkt durchgereicht und als CSS-Style gesetzt). Beispiel: "pointer" (≙ Cursor als Hand). Weitere Möglichkeiten: ↗ CSS-Dokumentation.

`.visible ← Number | Boolean [false]`

Sichtbarkeitsstatus der Spider-Shape. Entweder true (sichtbar) oder false (unsichtbar) oder ein Zahlenwert im Bereich von 0 (= unsichtbar) bis 100 (= sichtbar). Zwischenwerte geben den Grad der Transparenz an.

`.scale ← Number [100]`

Skalierungsfaktor der Spider-Shape als Prozentzahl. 100% entspricht der realen Größe. Skalierungsbereich ist [0...500].

`.label ← String [""]`

Beschriftungstext einer Spider-Shape. Wird automatisch horizontal und vertikal zentriert. Schriftart, -stil und -größe entsprechen den Festlegungen in den entsprechenden Attributen (siehe unten: *fontname, fontsize, fontstyle*), die Schriftfarbe der festgelegten Zeichenfarbe (.linecolor). Eine Änderung der Ausrichtung ist nicht vorgesehen.
Falls eine Spider-Shape über ein *image und* ein *label* verfügt, erscheint *image* im Hintergrund, das *label* davor.

`.image ← String ["spider“]`

Hintergrundbild einer Spider-Shape bzw. – falls kein Hintergrund, keine Umrandung und kein Label gesetzt sind – das Bild, das der Spider-Shape selbst entspricht. Beim zugewiesenen String muss es sich um einen gültigen Image-Namen handeln, d.h. den Namen einer zuvor über *registerImage*() für die Benutzung registrierten oder Bilddatei oder einer der vordefinierten Spider-Images.

spider	ant	bug	plane	heli	car
					

`.linesize ← Number [2]`

Stärke des Spinnenfadens in Pixeln. Zulässig sind Werte im Bereich [1..100].

`.linecolor ← String ["black"]`

Farbe des Spinnenfadens. Zu Farbangaben ↗ Hinweise oben.

`.lineend ← String ["round"] | "flat"`

Linienendenstil ist standardmäßig abgerundet – führt zu den besten Ergebnissen. Um mittels einer dicken Linie aber z.B. ein ausgefülltes Rechteck zu zeichnen, sehr unpraktisch. Verwendung von „flat“ liefert gerade abgeschnittene Enden.

`.fillcolor ← String [""]`

Füllfarbe für das Zeichnen von geschlossenen Figuren (↗ spinWeb()). Zu Farbangaben ↗ Hinweise oben.

`.fontname ← String ["sans"]`

Schriftart („font-family“) für alle Textausgaben einer Spider (label und print()). Verwendet werden können alle auf dem (ausführenden, d.h. Anwender-)System installierten Schriften.

`.fontsize ← Number [20]`

Schriftgröße in Pixeln aus dem Bereich [4; 800] für alle Textausgaben einer Spider (label und print()).

`.fontstyle ← String ["normal"]`

Schriftstil für alle Textausgaben einer Spider (label und print()). Gültig sind die Angaben "normal“, "bold" und "italic“ einzeln und in Kombination (z.B. "bold italic").

`.textalign ← String ["left"]`

Textausrichtung bei Ausgaben über print() ohne Auswirkung auf das label-Attribut (label wird immer horizontal und vertikal zentriert in die Spider-Shape gesetzt). Gültig sind "left“, "right", "justify" und "center" (CSS-konform).

Spider::Methoden

`.turn(w)`

Dreht die Spider um w Grad gegen den Uhrzeigersinn (für w > 0) bzw. im Uhrzeigersinn (für w < 0).

`.turnTo(w)`

Dreht die Spider auf das Winkelmaß w Grad, orientiert am in der Mathematik üblichen Winkel-Koordinatensystem:
0° ≙ Osten | 90° ≙ Norden | 180° ≙ Westen | 270° ≙ Süden Negative Werte für w sind ebenfalls möglich.

`.jump(d)`

Bewegt ein Spider-Objekt auf direktem Wege (d.h. so schnell wie möglich) um d Pixel vorwärts (d > 0) bzw. rückwärts (d < 0) bezogen auf die aktuelle Ausrichtung der Spider.

`.jumpTo(x,y [,"noturn"])`

Bewegt ein Spider-Objekt auf direktem Wege (d.h. so schnell wie möglich) mit seinem Mittelpunkt an den Punkt (x|y) des Koordinatensystems. Dazu wird die Spider zunächst in Zielrichtung ausgerichtet und dann ans Ziel bewegt.
Diese Ausrichtung (als Drehung sichtbar) lässt sich unterdrücken durch das optionale zusätzliche Argument "noturn".

`.move(d)`

Bewegt ein Spider-Objekt in der durch *speed* festgelegten Geschwindigkeit um d Pixel vorwärts (d > 0) bzw. rückwärts (d < 0) bezogen auf die aktuelle Ausrichtung der Spider.

`.moveTo(x,y [,"noturn"])`

Bewegt ein Spider-Objekt in der durch speed festgelegten Geschwindigkeit mit seinem Mittelpunkt an den Punkt (x|y) des Koordinatensystems. Dazu wird die Spider zunächst in Zielrichtung ausgerichtet und dann ans Ziel bewegt.
Diese Ausrichtung (als Drehung sichtbar) lässt sich unterdrücken durch das optionale zusätzliche Argument "noturn".

`.spin(d)`

Bewegt ein Spider-Objekt in der durch *speed* festgelegten Geschwindigkeit um d Pixel vorwärts (d > 0) bzw. rückwärts (d < 0) bezogen auf die aktuelle Ausrichtung der Spider und zeichnet dabei eine Linie gemäß der durch die Attribute .linesize und .linecolor festgelegten Linienstärke bzw. -farbe.

`.spinTo(x,y [,"noturn"])`

Bewegt ein Spider-Objekt in der durch *speed* festgelegten Geschwindigkeit mit seinem Mittelpunkt an den Punkt (x|y) des Koordinatensystems und zeichnet dabei eine Linie gemäß der durch die Attribute .linesize und .linecolor festgelegten Linienstärke bzw. -farbe. Dazu wird die Spider zunächst in Zielrichtung ausgerichtet und dann ans Ziel bewegt.
Diese Ausrichtung (als Drehung sichtbar) lässt sich unterdrücken durch das optionale zusätzliche Argument "noturn".

`.return()`

Lässt die Spider auf schnellstem Wege zu ihrer Ausgangsposition (bei „Geburt“) zurückkehren. Dazu wird die Spider in Richtung des Welt-Mittelpunktes ausgerichtet, springt dann dorthin und dreht sich anschließend nach Osten (0°).

`.getAngleTo(x,y) → Number`

Liefert den Winkel zurück, auf den man die Spider drehen muss, damit sie zum Punkt (x|y) hin ausgerichtet ist.

`.getDistanceTo(x,y) → Number`

Liefert die Distanz in Pixeln zurück, um die man die Spider vorwärts bewegen muss, um zum Punkt (x|y) zu gelangen.

`.spinWeb(n,d)`

Zeichnet ein regelmäßiges Vieleck mit n Seiten und Seitenlänge d ausgehend von der aktuellen Position und Ausrichtung der Spider. Die Spider selbst ändert Position und Ausrichtung dabei nicht. Die erste Seite des Vielecks verläuft in Ausrichtung der Spider.
Farbe und Stärke der Umrandung werden durch .linecolor und .linesize festgelegt, die Füllfarbe durch .fillcolor.

`.spinWeb(`x_1`,`y_1`,`x_2`,`y_2`,...,`x_n`,yn)`

Zeichnet ein Polygon mit den Eckpunkten $P_1(x_1|y_1)$, $P_2(x_2|y_2)$, …, $P_n(x_n|y_n)$. Bezugspunkt der Koordinaten ist dabei der Mittelpunkt der Spider, der dem Nullpunkte eines dazu relativen Koordinatensystems entspricht, das so orientiert ist wie das World-Koordinatensystem (↗ Seite 4). Die als Argumente übergebenen Koordinaten können auch negativ sein.
Die Spider selbst ändert Position und Ausrichtung dabei nicht.
Farbe und Stärke der Umrandung wird durch .linecolor und .linesize festgelegt, die Füllfarbe durch .fillcolor.

`.dot(r [,w])`

Zeichnet einen Kreis mit einem Radius von r Pixeln. Der Mittelpunkt entspricht dem Mittelpunkt der Spider, die ihre Position und Ausrichtung beim Zeichnen des Kreises nicht verändert.
Bei Angabe eines Mittelpunktswinkels w wird statt eines Kreises nur der zugehörige Kreissektor gezeichnet.

`.stamp([name])`

Hinterlässt einen Abdruck der Spider-Shape, der sich von der Shape selbst nicht unterscheidet (d.h. Skalierung, Drehung etc. werden genau so übernommen). Die Spider-Abdrücke liegen auf einer Zwischenebene zwischen den Zeichnungen der Spider (unterste Schicht) und den Spider-Objekten selbst (oberste Schicht). Soll ein einzelner Abdruck gezielt wieder entfernt werden, dann geschieht das über einen frei wählbaren Namen (String oder Number), der beim Erstellen des Abdrucks als Argument mitgegeben wird.

`.unstamp([name])`

Entfernt einen Spider-Abdruck namens name bzw. – falls ohne Argument aufgerufen – alle Abdrücke dieser Spider.

`.flipIn(["center"|"left"|"right"|"middle"|"top"|"bottom"])`

Ändert den Sichtbarkeitszustand der Spider von sichtbar (visible=true) zu unsichtbar (visible=false) und verbindet dies mit einem animierten „Einklappen“. Das Argument (Vorgabewert bei fehlendem Argument: "center") gibt die Seite der Spider-Shape an, an der „geklappt“ wird. Beachte: Liefert nur richtige Effekte, wenn Spider auf 0° ausgerichtet ist!

`.flipOut(["center"|"left"|"right"|"middle"|"top"|"bottom"] [,visible])`

Ändert den Sichtbarkeitszustand der Spider von unsichtbar (visible=false) zu sichtbar (optionales zweites Argument legt visibility-Wert 0..100 fest; Vorgabewert: visible = true) und verbindet dies mit einem animierten „Ausklappen“. Das Argument (Vorgabewert bei fehlendem Argument: "center") gibt die Seite der Spider-Shape an, an der „geklappt“ wird.

`.print(txt [,width])`

Gibt an der aktuellen Position der Spider und in deren Ausrichtung den Text txt auf der Zeichenfläche aus. Schriftart, -größe, -stil und Ausrichtung sind durch die Attribute *.fontname*, *.fontsize*, *fontstyle* und *.textalign* festgelegt, die Schriftfarbe durch *.linecolor*. Der optionale Parameter width gibt eine Maximalbreite der Textausgabe vor. Die Schrift wird dann in ihrer Breite so gestaucht, dass diese Vorgabe erfüllt wird.

`.input(txt,handler)`

Öffnet dasselbe Eingabefenster wie die globale read()-Funktion und erwartet eine Benutzereingabe. txt ist eine Benutzerinformation, die oberhalb der Eingabezeile erscheint. Die Methode liefert nicht direkt einen Rückgabewert (das ist bedingt durch die Abarbeitung der Animations-Warteschlange nicht möglich), sondern führt unmittelbar nach dem Schließen des Eingabefensters die Funktion *handler* aus – erst danach werden die weiteren Zeilen nach *.input*() abgearbeitet. Der Funktion *handler* wird – wie bei allen „echten Events“ – ein Event-Objekt als Argument mitgegeben, dessen Attribut *.input* die Benutzereingabe enthält, und zwar je nach Eingabe als Zeichenkette, Zahl oder Array (vgl. dazu die Beschreibung von *read*()).

`.wait(t)`

Hält den Programmablauf für eine Zeitdauer von t Millisekunden an.

`.countdown(t,handler)`

Führt die Funktion *handler* nach Ablauf von t Millisekunden einmal aus. Der Funktion *handler* wird – wie bei allen „echten" Events – ein Event-Objekt als Argument mitgegeben, das allerdings nur die Atribute *.type* ("timer") und *.source* besitzt.
Soll eine Funktion wiederholt ausgeführt werden, dann muss innerhalb von *handler* die *countdown*()-Methode erneut mit entsprechenden Argumenten aufgerufen werden.
Um die Ausführung von *handler* vor Ablauf der festgelegten Zeit zu stoppen (d.h. *handler* soll nicht ausgeführt werden), ruft man *.countdown*(-1,handler) auf.
Beachte: Anders als die „echten Events" wird *handler* nicht erst ausgeführt, wenn alle anderen Aktionen der Animations-Warteschlange abgearbeitet sind, sondern mit höchster Priorität, d.h. sofort.

`.setTimer()`

Drückt den Startknopf der Stoppuhr. Funktioniert genau wie die globale Funktion setTimer(), die allerdings im Rahmen von Spider-Animationen nicht korrekt funktioniert.

`getTimer() → Number (int)`

Liefert die seit dem Aufruf der Methode *.setTimer*() vergangene Zeit in Millisekunden. Dient als Ersatz für die globale Funktion getTimer(), die im Rahmen von Spider-Animationen nicht korrekt funktioniert.
Beachte: Die Zeitmessung funktioniert nur korrekt, wenn .getTimer() direkt zu Beginn eines Event-Handlers aufgerufen wird. In Fällen, in denen z.B. die Zeit bis zu einer Benutzeraktion gemessen werden soll, ist das i.d.R. kein Problem, weil diese Aktion meist in einem Event resultiert und die Bedingung damit erfüllbar ist.
Möchte man hingegen innerhalb eines Code-Blocks einer Spider-Animation z.B. die Laufzeit messen, dann kann man sich damit behelfen, dass man an der Stelle, an der die Zeit genommen werden soll, die .countdown()-Methode mit t=0 einsetzt und .getTimer() dann zu Beginn des zugeordneten Event-Handlers ausführt.

`.listen(event,handler)`

Lässt das Spider-Objekt auf bestimmte Ereignisse (↗ Kapitel „Ereignisse und Ereignisbehandlung") lauschen. Dazu wird der Name des gewünschten Ereignisses einem Funktionsnamen (handler) zugeordnet. Diese Funktion wird aufgerufen, sobald das Ereignis eintritt.
Beachte: Auf Events, die während der Ausführung einer laufenden Spider-Aktion ausgelöst werden, wird nicht reagiert. Erst bei „ruhendem System", d.h. mit Erreichen von world.animate() lauscht das Spider-Objekt auf Ereignisse.
Einem bestimmten Ereignis kann immer nur *ein* Handler zugeordnet werden. Will man mehrere Handler an ein Ereignis binden, dann kann man als Handler eine „Containerfunktion" benutzen, die der Reihe nach die einzelnen Handler aufruft.
Um eine Bindung zwischen Event und Handler zu lösen, wird das entsprechende Event mit *false* als zweitem Argument (statt eines Funktionsnamens) aufgerufen.

Beispiel

Ein einfaches JavaScript-Programm mit JSpider-Animation sieht so aus:

```
let netz = new World();          // World-Objekt erzeugen
let spinne = new Spider(netz);   // Spider-Objekt erzeugen
spinne.visible = true;           // Spinne sichtbar machen (optional)
spinne.turn(360);                // Spinne macht eine Umdrehung
spinnen.spin(100);               // Spinne bewegt sich 100px nach vorne
netz.animate();                  // Animation wird ausgeführt
```

Ereignisse und Ereignisbehandlung

Die grundlegende Funktionsweise der Ereignisbehandlung wird im Rahmen der *.listen*()-Methoden erklärt.
Es folgt eine Übersicht aller Events und aller Attribute des Event-Objekts sowie erforderliche Erläuterungen.

Ereignisse::Event-Typen

Event-Typ	Lauscher		Auslösendes Ereignis | Erläuterungen
	World	Spider	
"click"	✓	✓	Einfacher Mausklick (zusätzlich möglich: <Atl>, <Shift>, <Ctrl>)
"dblclick"	✓	✓	Doppelter Mausklick (zusätzlich möglich: <Atl>, <Shift>, <Ctrl>)
"wheel"	✓	✓	Mausrad
"mouseenter"		✓	Mauszeiger erfasst Spider-Shape
"mouseleave"		✓	Mauszeiger verlässt Spider-Shape
"mousemove"	✓	✓	Mauszeiger bewegt sich im World- bzw. Spider-Objekt
"drag"		✓	Mausklick auf Spider-Shape und bei gedrückter Maustaste verschieben
"drop"		✓	Maustaste nach Aufnehmen und Verschieben des Spider-Objekts loslassen
"keypress"	✓	✓	Taste(nkombination) drücken

Ereignisse::Event-Objekt

Attribut	Event-Typen	Inhalt des Attributs | Erläuterungen
.source	alle	Referenz auf das (World- oder Spider-)Objekt, das auf das Event lauscht.
.type	alle	Auslösender Event-Typ (Bezeichnung gemäß obiger Tabelle)
.key	"keypress", "click", "dblclick"	Name der gedrücken Tastenkombination (z.B. "AltCtrl-T") bzw. im Falle von darstellbaren Zeichen dieses Zeichen (z.B. "u").
.x	alle Maus-Events außer "wheel"	Position (x|y) des Mauszeigers im Koordinatensystem der Event-Quelle (d.h. entweder ein World-Objekt oder innerhalb einer Spider-Shape)
.y		
.wheel	Mausrad	±1 für Drehrichtung des Mausrades: 1 → hoch | −1 → runter
.input	.input()-Methode	Benutzereingabe nach Schließen des Eingabefenster (vgl. read())