Handelshögskolan vid Göteborgs universitet
Institutionen för informatik

J(ava)Script

- en standard, två viljor och miljontals användare -

Jörgen Hanson
Examensarbete I 10p
ADB-programmet 80p
Vårterminen 1999

Handledare:
Roy Corneliusson

ABSTRAKT

Rapportens syfte är att ge en grundläggande förståelse för de problem som är förknippade med JavaScripts och JScripts inkompatibilitet, genom att identifiera de båda skriptspråkens huvudsakliga likheter och olikheter.

Vid publicerandet av webbläsaren Netscape Navigator 2.0 presenterades ett inbyggt skriptspråk kallat JavaScript. Det växte snabbt i popularitet och det dröjde inte länge förrän Microsoft, med webbläsaren Internet Explorer 3.0, bestämde sig för att ge stöd åt skriptspråket. Microsoft passade samtidigt på att döpa om produkten till JScript och tillföra språket ytterligare funktionalitet.

Det fanns potential för JavaScript och JScript att till viss del tillfredsställa webbens behov av dynamik och interaktivitet, men språken var inkompatibla. För att försöka lösa problemet påbörjades utvecklingen av en skriptspråksstandard. European Computer Manufacturers Association (ECMA) stod för arbetet och i juni 1997 godkändes officiellt första upplagan av standarden ECMA-262. Den implementerades i JavaScript 1.3 och JScript 3.0. Ett problem kvarstod dock. Skriptspråken var fortfarande inte fullt ut kompatibla, eftersom de var för sig tillförde ytterligare funktionalitet utöver standarden.

I uppsatsen diskuteras problemet utifrån tre skilda områden. Det första området inkluderar standarden ECMA-262. Det andra området innefattas av den funktionalitet som skriptspråken tillför utöver standarden och som kan identifieras i båda språken. Det tredje området består av de faciliteter som är unika för respektive skriptspråk. För att kunna identifiera delområde två har en fallstudie utförts. Metoden för studien har varit att testa de objekt, egenskaper, metoder, operatorer och satser som finns tillgängliga i både JavaScript och JScript, men ej i ECMA-262.

Rapportens fallstudie och diskussion gav upphov till följande slutsatser:

- JavaScript 1.3 och JScript 3.0 är trots inkompatibilitetsproblematiken två mycket lika skriptspråk.
- Stommen i språken består av skriptspråksstandarden ECMA-262, vilken tillhandahåller grundläggande syntax och semantik i språken samt typer, operatorer, uttryck, satser och ett antal inbyggda objekt.
- Det finns ett behov av en standard för att ny funktionalitet skall fungera identiskt i båda språken.
- Den mest påtagliga skillnaden mellan skriptspråken finns ej i språkens kärna, utan snarare i den miljö i vilka de verkar.
- Mycket tyder på att ECMAScript-standarden kommer att utvecklas till den grad att skillnaden mellan språken blir försumbar.

FÖRORD

Jag vill först och främst tacka Canopus AB för att jag fick möjlighet att skriva min uppsats på en inspirerande arbetsplats. Det var Markus Östlund på Canopus som gav mig idén att fördjupa mig i skriptspråksproblematiken. Jag hoppas att min analys tillfört något av värde till företaget.

Jag skulle även vilja tacka min handledare Roy Corneliusson för ett vakande öga över uppsatsens form och utseende.

Jörgen Hanson
Göteborg 1999-05-26

INNEHÅLLSFÖRTECKNING

INLEDNING **1**

BAKGRUND 1
PROBLEMFORMULERING 2
SYFTE 2
AVGRÄNSNINGAR 2
METOD 3

TEORETISK REFERENSRAM **5**

SKRIPTSPRÅK 5
Skillnader mellan skriptspråk och vanliga programmeringsspråk *5*
Webbaserade skriptspråk *5*
ECMASCRIPT 7
ECMA *7*
Språkstrukturen i ECMAScript *8*
Implementering av ECMAScript-standarden *11*
Sammanfattning av ECMA-262 ..... *11*
DOKUMENTOBJEKTMODELLEN 18
World Wide Web Consortium *19*
Document Object Model Level 1 Specification *20*
JAVASCRIPT 23
Skillnader mellan JavaScript 1.3 och ECMAScript *24*
JSCRIPT 27
Skillnader mellan JScript 3.0 och ECMAScript *28*

FALLSTUDIE **30**

STRIKT JÄMFÖRELSE 31
DO...WHILE OCH SWITCH 31
ARRAY - CONCAT OCH SLICE 33
FUNCTION - CALLER 34
STRING 36
REGEXP 40

DISKUSSION **43**

ANALYSOMRÅDE 1 43
ANALYSOMRÅDE 2 44
ANALYSOMRÅDE 3 45

SLUTSATSER **46**

KÄLLFÖRTECKNING **47**

FIGURFÖRTECKNING

Figur 1: Relationen mellan webbskriptspråk, värdmiljö och Java.....6
Figur 2: Relationen mellan objekten Employee, Manager och WorkerBee.....9
Figur 3: Dokumentobjektmodellen.....18
Figur 4: Tabell i HTML representerad av dokumentobjektmodellen.....21
Figur 5: Förhållandet mellan kärnan, klientsidan och serversidan i JavaScript.....23
Figur 6: Fallstudiens målområde.....30
Figur 7: Tre analysområden.....43

BILAGOR

Bilaga 1: Ordlista
Bilaga 2: Uttryck i ECMAScript
Bilaga 3: Egenskaper och metoder i ECMAScripts inbyggda objekt
Bilaga 4: Ytterligare funktionalitet i JavaScripts inbyggda objekt
Bilaga 5: Ytterligare funktionalitet i JScripts inbyggda objekt

INLEDNING

Om fem år kommer nedan beskrivna problem inte längre att existera.

Låt oss emellertid inte gå händelserna i förväg, utan skapa oss en bild av hur situationen ser ut idag. De två mest använda webbläsarna på marknaden är Netscape Navigator och Microsoft Internet Explorer. Det finns ett antal tekniker för att skapa dynamik och interaktivitet på webben, men endast två standardiserade programmeringsspråk på klientsidan. Det ena är Java, ett plattformsoberoende fullständigt språk. Det andra är ECMAScript, en skriptspråksstandard implementerad i respektive webbläsare i form av Netscapes JavaScript 1.3 och Microsofts JScript 3.0.

Skriptspråken behandlar webbläsaren och dess element som objekt, vilka kan kontrolleras med hjälp av inbyggd kod i webbsidan. Ett webbskriptspråk tillhandahåller ett enkelt och effektivt sätt för att skapa dynamik och interaktivitet i samband med utnyttjandet av Internet-teknik. Det finns dock ett problem. Trots en fullgod skriptspråksstandard är JavaScript och JScript inte fullt ut kompatibla. De tillhandahåller var för sig egen funktionalitet utöver standarden. En viss del av denna ytterligare funktionalitet är identisk språken emellan, men finns ej specificerad i skriptspråksstandarden.

Finns det någon möjlighet att ge användare vägledning i problematiken kring skriptspråkens inkompatibilitet? Vi börjar med att titta på bakgrunden till problemet.

Bakgrund

Internet breder ut sig mer för var dag som går. De senaste tio åren har inneburit en explosionsartad utveckling och det har kommit att bli en del av vardagen för många människor. Internet fick dock ingen större uppmärksamhet förrän Tim Berners-Lee på CERN kom på idén om något som kom att kallas ”World Wide Web”, eller kort, webben. En ganska enkel idé där den enda interaktivitet som existerade mellan dator och användare var textlänkar till andra sidor och formulär för att kunna överföra information till en server. Mjukvaran som användes för att läsa in och visa sidorna hade ett enkelt gränssnitt och en primitiv översättare som kunde skilja på text, bilder, länkar och formulär. I takt med ökad popularitet kom kraven att bli större på webbläsarna och framförallt på de tekniker som användes för att skapa dynamik och interaktivitet. Den kanske mest ambitiösa planen för att lösa problemet utvecklades av Sun Microsystems. De presenterade Java, ett komplett plattformsoberoende språk baserat på C++. De flesta webbläsare på marknaden såg till att ge stöd för språket med hjälp av en Java-kompilator. Samtidigt identifierade mjukvaruleverantörerna det ökade behovet av att förenkla utvecklingen på webben. Microsoft, med webbläsaren Internet Explorer, lovade att tillföra den fulla kraften av Windows gränssnitt till webben genom VBScript[1] och ActiveX[2]-kontroller. Men det var deras största konkurrent Netscape som drog det längsta

[1] Ett webbskriptspråk från Microsoft baserat på Visual Basic. Fungerar endast i webbläsaren Internet Explorer.
[2] En teknik för att inkludera olika typer av objekt (text, bild, ljud, film, knappar, datorprogram etc) på en webbsida. Den är utvecklad av Microsoft och kan ses som ett svar på SUN:s Java-teknik.

strået när de introducerade skriptspråket JavaScript, vars popularitet snabbt växte sig stark bland webbutvecklare. Plötsligt bestämde sig Microsoft för att ge stöd åt skriptspråket, men med det egna produktnamnet JScript. I förlängningen visade det sig dock inte bara vara namnet som skilde dem åt, utan Microsoft hade bestämt sig för att utveckla språket på egen hand, samtidigt som Netscape släppte en ny version av sitt skriptspråk. En ohållbar utveckling tog fart där ingen var vinnare, framförallt inte användarna som protesterade mot den ökade inkompatibiliteten i språken. Netscape lämnade då en förfrågan till European Computer Manufacturers Association (ECMA) om utvecklandet av en standard baserad på JavaScript. Resultatet blev ECMA-262, en standard baserad på ett flertal språk, men främst JavaScript och JScript. Både Netscape och Microsoft skyndade att ge stöd för standarden, samtidigt som respektive språk fortsatte att utvecklas på eget håll.

Mycket har även hänt vid sidan av skriptspråkens utveckling. Det har uppstått ett ökat behov för dynamik och interaktivitet på webben. Det blir t ex allt vanligare att Internet-teknik används för att utveckla intranät och extranät i företag och elektronisk handel växer sig allt starkare. Ett vanligt krav är att ett färdigt system skall fungera i åtminstone de två populäraste webbläsarna, dvs Netscape Navigator och Internet Explorer. Förutom typ av webbläsare finns ofta även ett krav på vilken version av webbläsare som systemet skall ge stöd för.

Det har idag gått snart ett år sedan ECMAScript-standarden publicerades. Standarden stöds fullt ut av JavaScript 1.3 och JScript 3.0. Om man som systemutvecklare lyckas övertala en kund om att använda den webbläsarversion som stödjer standarden, kvarstår ett problem. Språken är fortfarande inte fullt ut kompatibla med varandra, eftersom de utvecklats på olika håll av två skilda företag med olika mål. Det finns ingen guide eller mall som visar på vilket sätt språken skiljer sig åt.

Problemformulering

Med ovan som bakgrund är det intressant att studera de båda skriptspråkens inkompatibilitet. Uppsatsens problem är formulerat på följande sätt:

- Vad finns för huvudsakliga likheter och olikheter mellan JavaScript och JScript?
- Hur kan en framtida utveckling tänkas se ut?

Syfte

Uppsatsen skall ge en grundläggande förståelse för de problem som är förknippade med JavaScripts och JScripts inkompatibilitet, genom att identifiera de båda skriptspråkens huvudsakliga likheter och olikheter.

Avgränsningar

Uppsatsen avgränsar sig till att jämföra JavaScript 1.3 i Netscape Navigator 4.06-4.5 och JScript 3.0 i Microsoft Internet Explorer 4. Ingen hänsyn har tagits till eventuell bakåtkompatibilitet till tidigare versioner av språken.

Rapportens syfte är inte att vara en nybörjarguide eller fullständig referens till skriptspråken. Den riktar sig till personer med grundläggande kunskap om Internet och de tekniker, främst HTML (HyperText Mark-up Language), JavaScript och JScript, som används för presentation av information på webben.

Uppsatsen är avgränsad till att jämföra kärnan i skriptspråken, vilket medför att de skillnader som kan härledas till webbläsarnas objektmodeller, och som ej är direkt relaterade till skriptspråkens kärna, inte kommer att analyseras. Ett problem som kvarstår är att identifiera vad som är kärnan i skriptspråken och vad som tillhandahålls via andra faciliteter i webbläsaren. Jag har utgått från ECMAScript-specifikationen som skriptspråkens kärna. Det är dock ett faktum att det är svårt att beskriva ett webbskriptspråk utan att ge en bild av den miljö i vilken det verkar. Uppsatsen innehåller därför en presentation av dokumentobjektmodellen i syfte att ge en förståelse för hur skriptspråken interagerar med en webbläsare.

Metod

Jag märkte i ett tidigt skede att det skulle komma att bli svårt att hitta litteratur som knyter an till uppsatsens problem. Själva ämnet berörs ofta i litteraturen, men det finns ingen bok som redogör för problemet fullt ut. Samtidigt finns inte många böcker som berör de versioner av språken rapporten är avgränsad till att beskriva. Situationen har därför varit sådan att jag har fått använda mig av mer aktuella källor på webben. Jag förstod tidigt att det skulle komma att krävas en stor portion källkritik. Det finns många felaktiga tolkningar och påståenden att finna om man inte är uppmärksam. I uppsatsen har jag valt att endast referera till grundkällor, dvs Netscapes och Microsofts referensmaterial. Därmed inte sagt att jag ej har haft stor nytta av övrig information på webben. För att lära mig grunderna i de båda skriptspråken har jag spenderat åtskilliga timmar framför utmärkta kurser och guider, samtidigt som jag har haft stor nytta av diverse diskussionsgrupper på Internet.

Mycket av teorin i uppsatsen innefattar förklaring av språkens syntax. För att undvika missförstånd har jag valt att uteslutande använda engelska termer. Alla nyckelord symboliseras av fet stil (t ex **new**, **for**, **if**). Ord i kursiv stil representerar namn eller uttryck som definieras av användaren (t ex *statement*, *expression*, *argument*). De delar som är omgivna av hakparenteser [] är frivilliga, dvs de är inte nödvändiga för att fullgöra ett uttryck eller sats. I bilaga 1 finns en ordlista som förklarar de vanligaste engelska termer som förekommer i uppsatsen.

Jag har även valt att göra vissa prioriteringar vid förklaring av skriptspråkens syntax. Eftersom rapporten ej syftar till att ge en fullständig syntaxreferens av språken, är vissa delar mer utförligt beskrivna med förklarande exempel etc, medan andra delar har bedömts som mindre viktiga och därmed fått en mindre beskrivning. För fullständig information om en specifik del av syntaxen hänvisas till respektive skriptspråks referensmaterial.

Förutom studier av teoretisk karaktär har jag valt att använda mig av en fallstudie för underlag till diskussion. Enligt Backman (1998) kan en sådan studie bestå av ett eller

flera fall. Vidare kan den vara antingen beskrivande (deskriptiv), förklarande eller undersökande (explorativ). Studien i uppsatsen är undersökande och består av flera fall. Eftersom jag ej har haft tillgång till skriptspråkens källkod eller annan tydlig specifikation av språkens inre semantik, har jag valt att undersöka konkreta fall och jämföra resultatet. Varje enskild ytterligare funktionalitet, som båda skriptspråken tillhandahåller utöver ECMAScript-standarden, har undersökts. Fallstudiens förberedelse bestod av att försöka utveckla kodexempel att jämföra i respektive webbläsare. För att undvika missförstånd har jag använt mig uteslutande av engelska termer även i fallstudiens kod.

TEORETISK REFERENSRAM

Skriptspråk

De programmeringsspråk som används för att manipulera, modifiera och automatisera faciliteter i ett existerande system kallas för skriptspråk. I sådana system finns redan användbar funktionalitet tillgänglig genom ett användargränssnitt och skriptspråket är en mekanism för att utnyttja funktionaliteten till programkontroll. På detta sätt sägs det att det existerande systemet tillhandahåller en värdmiljö av objekt och faciliteter, vilka fullbordar skriptspråkets möjligheter.

Skillnader mellan skriptspråk och vanliga programmeringsspråk

Som komplement till dagens programmeringsspråk finns applikationer som tillhandahåller en miljö för systemutveckling med hjälp av skriptspråk. Det blir allt vanligare att mjukvaruföretag försöker sälja ett eget gränssnitt till utvecklare och ge möjlighet till programkontroll genom enklare skript. Det går att identifiera två stora skillnader mellan fullständiga programmeringsspråk och skriptspråk:

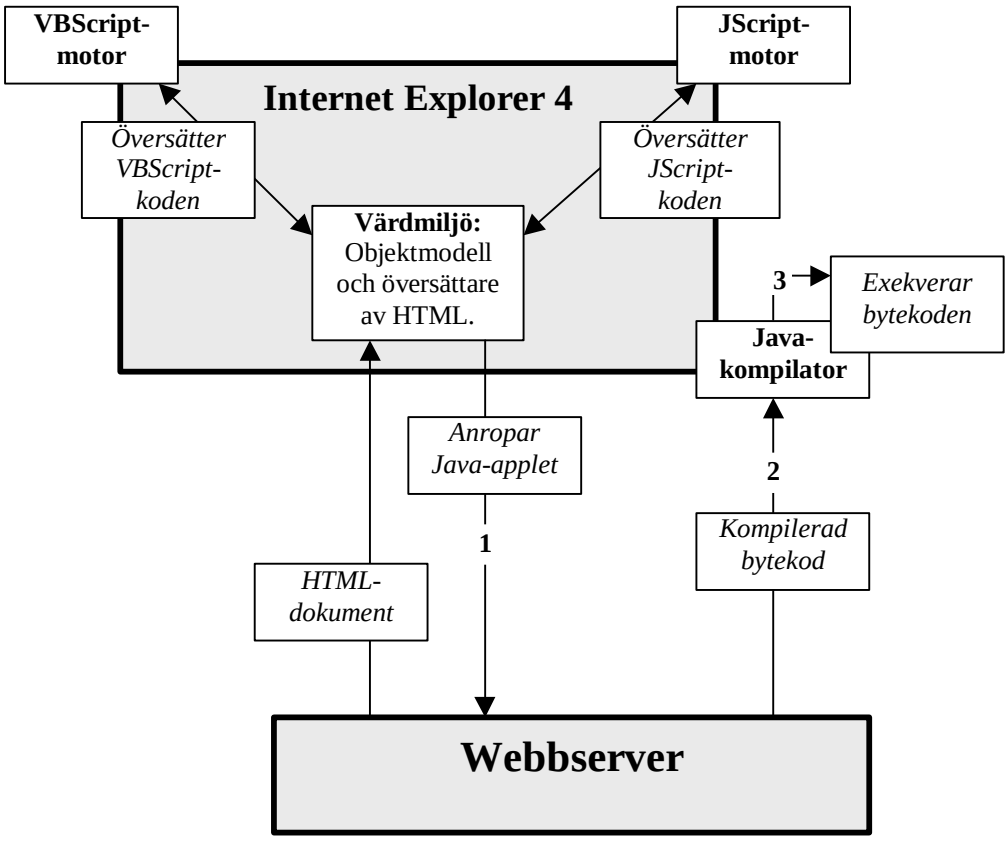
1. Ett skriptspråk kan inte exekveras utan en miljö att verka i. Ett programmeringsspråk däremot är självförsörjande och kan skapa kompilerade binära filer som exekveras självständigt.
2. Syntaxen och reglerna i ett skriptspråk är ofta mindre strikta och invecklade än i vanliga programmeringsspråk.

Fördelar och nackdelar med skriptspråk gentemot fullständiga programmeringsspråk kan identifieras i ovanstående skillnader. En fördel är att det ofta finns en färdig struktur för utveckling av gränssnitt och det går enkelt och snabbt att skapa applikationer i en skriptspråksmiljö. Däremot finns inte samma frihet i design av gränssnitt och inte samma möjligheter att skapa en applikation från grunden med kontroll över datorns alla faciliteter.

Webbaserade skriptspråk

Det finns webbaserade skriptspråk att tillgå på både klient- och serversidan. På klientsidan kan en webbläsare tillhandahålla en värdmiljö åt ett eller flera skriptspråk. Denna värdmiljö innehåller objekt som representerar det aktuella dokumentet, menyer, dialogboxar och textfält mm. Vidare finns funktionalitet för att inkludera skriptkod i händelser som t ex användarinput, mushändelser och selektioner mm. Skriptkoden är inbakad i HTML-dokumentet och den färdiga webbsidan är en kombination av användargränssnittselement, text och bilder. Skriptkoden reagerar direkt på användarinteraktion och det finns inget behov av ett huvudprogram.

Det finns två huvudsakliga skillnader mellan ett webbskriptspråk och programmeringsspråket Java. Det första är att Java är skilt från HTML, medan ett skriptspråk integreras med, och är inbakat i, HTML-koden. Det andra är att webbskriptspråket översätts av en skriptspråksmotor (ej kompilerad kod) på klienten, medan Java producerar kompilerad bytekod som laddas ner från en server och exekveras på klienten med hjälp av en Java-kompilator. Nedan följer ett exempel på relationen mellan webbskriptspråk, värdmiljö och Java.



Figur 1: Relationen mellan webbskriptspråk, värdmiljö och Java.

På serversidan tillhandahåller en webbserver en värdmiljö för beräkning av data på servern. Den innehåller objekt som kan representera förfrågningar, klientfiler och mekanismer för att dela på gemensamma data mm. Genom att använda skript på både klient- och serversidan är det möjligt att distribuera databeräkningar mellan klienten och servern samtidigt som ett egendefinierat användargränssnitt tillhandahålls.

ECMAScript

I november 1996 påbörjades utvecklingen av en skriptspråksstandard för webben. Uppgiften tillföll European Computer Manufacturers Association (ECMA) och arbetet baserades på främst JavaScript från Netscape Communications och JScript från Microsoft Corporation. Första utgåvan godkändes officiellt i juni 1997. Den specifikation som redogörs för i uppsatsen, andra utgåvan av ECMA-262, anammades i juni 1998 och är idag gällande. En tredje utgåva är planerad till slutet av 1999 och den kommer att innehålla den andra versionen av språket (ECMA, 1998).

ECMA

1961 bildades European Computer Manufacturers Association (ECMA). Det är en internationell europabaserad organisation som arbetar med standardisering av informations- och kommunikationssystem. ECMA består av ordinarie, associerade och SME-medlemmar.

- En **ordinarie** medlem är ett företag som i Europa utvecklar, producerar och marknadsför hårdvaru- eller mjukvaruprodukter eller tjänster inom informationsteknologi- eller telekommunikationsområdet i syfte att behandla digital information för affärsverksamhet, forskning, kontroll, kommunikation eller annat liknande ändamål.
- En **associerad** medlem är ett företag som har intresse och erfarenhet i ärenden relaterade till en eller flera av förbundets tekniska kommittéer, men som ej är kvalificerad att bli ordinarie medlem.
- En **SME**-medlem är ett företag som har intressen liknande ovan och har en årlig omsättning av mindre än hundra miljoner schweiziska franc.

ECMA:s formella mål är att:

- utveckla, i samarbete med lämpliga nationella, europeiska och internationella organisationer, standarder och tekniska rapporter i syfte att möjliggöra och standardisera användningen av ICT-system (Information and Communication Technology);
- uppmuntra till korrekt användning av standarder genom att inverka på de miljöer i vilka de tillämpas och;
- förkunna de olika standarder som ECMA producerar.

Standardiseringsarbetet inom ECMA utförs av ett antal tekniska kommittéer, i vissa fall uppdelade i arbetsgrupper, var och en aktiva i en bestämd sektor. ECMA-standarder utvecklas av kvalificerade experter inom informationsteknologi- och telekommunikationsindustrin. De har som främsta uppgift att på ett övergripande sätt förse tekniska lösningar klara för implementering i produktmiljöer. Organisationen har under trettio år arbetat fram ca 270 officiella standarder och publicerat över 70 tekniska rapporter. Alla standarder och tekniska rapporter är fritt tillgängliga för alla intressenter utan avgifter eller andra restriktioner.

ECMA har alltid arbetat för att försöka verka förutseende, vilket bl a innebär att de försöker identifiera teknologiska trender på ett tidigt stadium. Som en följd har många standarder blivit accepterade som en grund för internationella och europeiska standarder. För att säkerställa samarbete har organisationen upprättat formella relationer med alla europeiska och internationella standardiseringsorgan. De kommande åren ser ECMA många viktiga utmaningar i standardiseringen av informations- och kommunikationssystem, speciellt inom följande områden:

- Multimedia
- Lagringsmedier med hög kapacitet
- Höghastighetstelekommunikation
- Mjukvarukonstruktion
- IT-säkerhet
- Applikationsportabilitet

Språkstrukturen i ECMAScript

ECMAScript är ett objektorienterat programmeringsspråk tillägnat att utföra beräkningar och manipulera objekt i en värdmiljö. Det är inte avsett att vara självförsörjande, dvs kunna klara sig utan en objektmiljö att verka i. Exempelvis finns inga bestämmelser i specifikationen för input av extern data eller output av behandlade resultat. Istället förväntas det av den värdmiljö, i vilken ECMAScript-programmet verkar, att tillhandahålla inte bara objekt och andra faciliteter beskrivna i specifikationen utan även vissa miljöspecifika värdobjekt, vilkas beskrivning och beteende är undantagna specifikationens begränsning. ECMAScript var ursprungligen konstruerat att vara ett skriptspråk för webben, tillhandahållande en mekanism för att ge liv åt statiska webbsidor i olika webbläsare och utföra databeräkningar även på serversidan, som en del av en webbaserad klient-server-arkitektur. Specifikationen tillhandahåller dock skriptegenskaper för en rad olika värdmiljöer och kärnan i skriptspråket är specificerad utan förbindelse till en speciell typ av miljö.

Grundläggande språk och värdfaciliteter tillhandahålls av **objekt**, och ett ECMAScript-program är en samling kommunicerande objekt. Ett objekt i skriptspråket består av en obestämd samling egenskaper. Varje **egenskap** kan ha inga eller ett obestämt antal **attribut**, vilka bestämmer hur varje egenskap kan användas. Egenskaper är behållare som innehåller andra objekt, primitiva värden eller metoder. Ett **primitivt värde** tillhör någon av följande inbyggda typer: **Undefined**, **Null**, **Boolean**, **Number** eller **String**. En **metod** är en **funktion** relaterad till ett objekt via en egenskap.

ECMA-262 specificerar en samling inbyggda objekt: **Global**, **Object**, **Function**, **Array**, **String**, **Boolean**, **Number**, **Math** och **Date**. Standarden definierar också en mängd inbyggda operatorer. Syntaxen liknar på vissa punkter Javas syntax. ECMAScript är dock inte lika strikt, för att kunna fungera som ett lättanvänt skriptspråk. Exempelvis så behöver inte en variabels typ eller en typ förknippad med en egenskap vara deklarerad. ECMAScript innehåller inga riktiga objektklasser som i t ex C++, Smalltalk eller Java, utan istället används speciella funktioner (konstruktorer) som skapar objekt genom att exekvera kod, vilken avsätter plats för objekten att initialiseras i. Alla funktioner,

inklusive konstruktorer, är objekt, men inte alla objekt är konstruktorer. Varje konstruktor har en egenskap kallad **Prototype** som används för att implementera prototypbaserat arv (se nedan) och gemensamma egenskaper. Objekt skapas, med vissa undantag, genom att använda en konstruktor i kombination med uttrycket **new**.

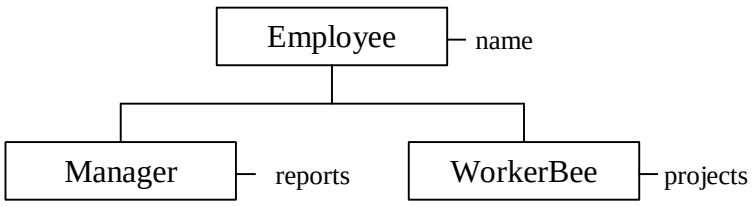
`new String("EcmaScript")`	skapar ett nytt strängobjekt.

Att kalla på en konstruktor utan att använda **new** ger olika konsekvenser beroende på konstruktorns beteende.

ECMAScript stödjer prototypbaserat arv, vilket är en teknik för att utnyttja de fördelar som arv medför utan att använda objektklasser. Prototypbaserat arv kan uppfattas som avancerat vid första anblick. Nedan följer ett försök att förklara hur logiken kring tekniken är uppbyggd.

Varje konstruktor har en associerad prototyp (Prototype) och varje objekt som skapas av konstruktorn har en implicit referens till dess prototyp. En prototyp kan i sin tur referera till en prototyp högre upp i den så kallade prototypkedjan (the prototype chain). När programmet refererar till en egenskap i ett objekt, så refereras till egenskapen i det första objektet i prototypkedjan som innehåller en egenskap med det namnet. Med andra ord, först kontrolleras om det objekt som refereras till har en sådan namngiven egenskap; om det objektet innehåller den namngivna egenskapen, så är det den egenskapen som kommer att användas; om objektet inte innehåller den namngivna egenskapen, så kontrolleras prototypen för objektet därnäst i prototypkedjan. Alla objekt som inte direkt innehar en speciell namngiven egenskap som dess prototyp har, delar den egenskapen och dess värde med övriga objekt som refererar till samma prototyp.

I ett klassbaserat objektorienterat programmeringsspråk så innehas i allmänhet ett objekts tillstånd av instanser, metoder av klasser och arv finns endast för struktur och beteende. I ECMAScript så är tillståndet och metoderna en del av objektet och struktur, beteende och tillstånd är alla ärvda. Nedanstående exempel[3] och tabell visar de tydligaste skillnaderna mellan klassbaserat och prototypbaserat arv.



Figur 2: Relationen mellan objekten Employee, Manager och WorkerBee

[3] Baserat på ett exempel ur Client-Side JavaScript Guide Version 1.3 (1998).

Vid implementering av arv enligt figur 2, kommer koden i respektive språk skilja sig åt som följer:

ECMAScript	Java

`function Employee ()`
`{`
`  this.name = “”;`
`}`
	`public class Employee`
	`{`
	`  public String name;`
	`  public Employee`
	`  {`
	`  this.name = “”;`
	`  }`
	`}`

`function Manager ()`
`{`
`  this.reports = [];`
`}`
`Manager.prototype = new Employee;`
 	
`function WorkerBee ()`
`{`
`  this.projects = [];`
`}`
`WorkerBee.prototype = new Employee;`
	`public class Manager extends Employee`
	`{`
	`  public Employee[] reports;`
	`  public Manager ()`
	`  {`
	`  this.reports = new Employee[0];`
	`  }`
	`}`
	
	`public class WorkerBee extends Employee`
	`{`
	`  public String[] projects;`
	`  public WorkerBee ()`
	`  {`
	`  this.projects = new String[0];`
	`  }`
	`}`

Prototypbaserat arv	**Klassbaserat arv**
Alla objekt är instanser.	Klasser och instanser är skilda entiteter.
En samling objekt definieras och skapas med konstruktorfunktioner.	En klass definieras med en klassdefinition och instansieras med konstruktormetoder.
Ett enskilt objekt skapas med operatorn **new**.	Samma tillvägagångssätt.
En objekthierarki konstrueras genom att ett objekts konstruktorfunktion tilldelas ett annat objekts prototyp	En objekthierarki konstrueras genom att definera underklasser i klassdefinitionen.
Egenskaper ärvs genom prototypkedjan.	Egenskaper ärvs genom klasskedjan.
Konstruktorfunktionen eller prototypen specificerar initiala egenskaper. Egenskaper kan i efterhand dynamiskt läggas till eller tas bort för individuella objekt eller för en samling objekt med samma prototyp.	Klassdefinitionen specificerar alla egenskaper till alla instanser av klassen. Egenskaper kan inte läggas till dynamiskt.

Implementering av ECMAScript-standarden

Innan ett skriptspråk kan anses fullt ut överensstämmande med ECMAScript-standarden, krävs att ett antal formella kriterier är uppfyllda:

- En regelrätt implementering av ECMAScript-standarden måste tillhandahålla och stödja alla typer, värden, objekt, egenskaper, funktioner, och programsyntax som beskrivs i specifikationen.
- En regelrätt implementering av standarden skall tolka tecken i enlighet med Unicode-standarden, version 2.0, och ISO/IEC 10646-1 med UCS-2 som den antagna kodningsformen, implementeringsnivå 3. Om den antagna uppdelningen i ISO/IEC 10646-1 inte på annat sätt specificeras, antas BMP-uppdelning, samling 300.
- En regelrätt implementering av ECMAScript är tillåten att tillhandahålla ytterligare typer, värden, objekt, egenskaper och funktioner utöver de som beskrivs i specifikationen. I synnerhet är en regelrätt implementering av ECMAScript tillåten att tillhandahålla egenskaper som inte beskrivs i specifikationen, och värden för dessa egenskaper, för objekt som beskrivs i specifikationen.
- En regelrätt implementering av ECMAScript är tillåten att stödja programsyntax som inte beskrivs i specifikationen. I synnerhet är en regelrätt implementering av ECMA-Script tillåten att stödja programsyntax som utnyttjar de “framtida reserverade orden” i specifikationen.

Sammanfattning av ECMA-262

Nedan följer en sammanfattning av specifikationen, med tyngdpunkt på de delar som knyter an till uppsatsens syfte.

Kapitel 1-7 ger en bakgrund till och överblick över skriptspråksstandarden. Övergripande förklaringar av språkets syntax presenteras, dvs förklaring av definitioner och grammatikens uppbyggnad. Kapitlen specificerar hur koden skall översättas av skriptmotorn för att kunna tolkas på ett korrekt sätt. Det redovisas i detalj vilka syntax-ord som är reserverade, hur kommatering skall utföras, vilka tecken som får användas i strängar och i övrig kod, hur siffror skall behandlas, regler för automatisk generering av semikolon mm.

Kapitel 8 redogör för typer (types). Ett värde är en entitet som tillhör en av nio typer. Nedan följer de nio typerna med en kort förklaring.

Undefined	innehar alltid värdet **undefined**.
Null	innehar alltid värdet **null**.
Boolean	representerar en logisk entitet med två värden: **true** och **false**.
String	representerar en sträng innehållande tecken.
Number	representerar ett numeriskt värde.
Object	representerar en ostrukturerad samling egenskaper. Varje egenskap består av ett namn, ett värde och ett antal attribut.
Reference	Värden av typ Reference, List och Completion används endast som mellanlagringsresultat i en uttrycksutvärdering och kan inte lagras i ett objekts egenskaper. De är språkets interna typer.
List	
Completion	

Kapitel 9 redogör för typkonvertering. ECMAScript utför vid exekvering automatiskt typkonvertering vid behov. Specifikationen innehåller ett antal operatorer för konvertering. Dessa operatorer är inte en del av språket, utan definieras i syfte att förstå språklogiken i specifikationen. Varje operator kan ta input i form av ett värde av typen Undefined, Null, Boolean, Number, String eller Object. Följande operatorer specificeras:

ToPrimitive	konverterar ett värde till en primitiv typ, dvs icke-objekttyp. Operatorn kan ta två argument, dels värdet, dels ett valfritt argument som anger den typ som föredras om värdet kan konverteras till mer än en primitiv typ.
ToBoolean	konverterar ett värde till typen Boolean.
ToNumber	konverterar ett värde till typen Number.
ToInteger	konverterar ett värde till ett numeriskt heltal.
ToInt32	konverterar ett värde till ett av 2^{32} heltal mellan -2^{31} och 2^{31}-1.
ToUint32	konverterar ett värde till ett av 2^{32} heltal mellan 0 och 2^{32}-1.
ToUint16	konverterar ett värde till ett av 2^{16} heltal mellan 0och 2^{16}-1.
ToString	konverterar ett värde till typen String.
ToObject	konverterar ett värde till typen Object.

I **Kapitel 10** definieras och specificeras den exekverbara kodens sammanhang. Kapitlet specificerar källkodens uppdelning i ett antal logiska delar och dess inbördes förhållande.

Kapitel 11 redogör för ECMAScripts uttryck (Expressions). Samtliga uttryck med förklaring finns tillgängliga i bilaga 2.

Kapitel 12 redogör för satser (statements). Följande satser med tillhörande syntax är specificerade:

Block	**{**[*statementList*]**}**
VariableStatement	**var** *variableDeclarationList*;
EmptyStatement	;
ExpressionStatement	*expression*;
IfStatement	**if** (*expression*) *statement* **else** *statement*
	if (*expression*) *statement*
IterationStatement	**while** (*expression*) *statement*
	for ([*expression*]; [*expression*]; [*expression*]) *statement*
	for (**var** *variableDeclarationList*; [*expression*]; [*expression*]) *statement*
	for (*leftHandSideExpression* **in** *statement*) *statement*
	for (**var** *identifier* [*initializer*] **in** *statement*) *statement*
ContinueStatement	**continue**;
BreakStatement	**break**;
ReturnStatement	**return** *expression*;
WithStatement	**with** (*expression*) *statement*

I **Kapitel 13** specificeras hur en funktionsdeklaration hanteras. Följande syntax deklarerar en ny funktion i ECMAScript:

```
function identifier ([formalParameterList]) block
```

Utifrån ovan givna deklaration skapas en egenskap, tillhörande det globala objektet, vars namn blir *Identifier* och vars värde är ett funktionsobjekt med given parameterlista ([*formalParameterList*]) och funktionskod (*block*).

Kapitel 14 ger instruktioner för hur ett program i ECMAScript skall utläsas, i vilken ordning programmet skall utvärderas och processeras.

I **kapitel 15** specificeras ECMAScripts grundläggande inbyggda objekt. Det finns ett antal inbyggda objekt tillgängliga när ett ECMAScript-program börjar exekveras. Längst upp i hierarkin finns det globala objektet. Övriga inbyggda objekt är åtkomliga som initiala egenskaper av det globala objektet.

Egenskaper och metoder tillhörande de inbyggda objekten finns att tillgå i bilaga 3, om de inte av särskilda skäl valts att redovisas nedan.

Det globala objektet (The Global Object)
Syftet med det globala objektet är att fånga in globala egenskaper och metoder i **ett** objekt. Det globala objektet saknar konstruktor, vilket innebär att det inte går att skapa globala objekt med operatorn **new**. Det är inte heller möjligt att åkalla objektet som en funktion.

Object
Alla ECMAScript-objekt härstammar från objektet **Object**. Nedanstående egenskaper och metoder tillhör prototypen till **Object** och är därmed åtkomliga för alla ECMAScript-objekt.

Objektets konstruktor:
`new Object()`

Egenskaper

`constructor`	innehåller en referens till den funktion som skapar ett objekts prototyp. Egenskapen är medlem av prototypen för varje objekt som har en prototyp. Det inkluderar alla inbyggda ECMAScript-objekt frånsett **Global** och **Math**.
	
	`ex.`
	`function Car (value) {`
	`  this.brand = value;`
	`}`
	
	`myNewCar= new Car("Alfa Romeo 164 v6");`
	`document.writeln(myNewCar.constructor);`
	
	Resultat:
	`function Car (value) { this.brand = value; }`

`prototype`	representerar prototypen för en typ av objekt. Används för att tillföra egenskaper och metoder till alla instanser av en objektkategori.
	
	`ex.`
	`function array_max( )`
	`{`
	`  var i, max = this[0];`
	`  for (i = 1; i < this.length; i++)`
	`  {`
	`  if (max < this[i])`
	`  max = this[i];`
	`  }`
	`  return max;`
	`}`
	`Array.prototype.max = array_max;`
	`var x = new Array(1, 2, 3, 4, 5, 6);`
	`var y = x.max( );`

Metoder

`toString``()`	returnerar en sträng som specificerar det aktuella objektet.
	ex.
	`var myObject = new Object()`
	`document.writeln(myObject.toString)`
	
	Resultat:
	`[object Object]`

`valueOf``()`	returnerar det primitiva värdet av ett givet objekt.

Varje inbyggt objekt (förutom **Math)** har en egen **toString-** och **valueOf**-metod som upphäver **Object.toString()** och **Object.valueOf()**. Nedanstående tabeller visar respektive objekts metodbeteende.

toString

Objekt	**Beteende**
Array	konverterar elementen i arrayen till en sträng separerade av kommatecken.
Boolean	returnerar ”true” om värdet är **true**, annars returneras ”false”.
Function	returnerar en sträng med följande utseende: ”function *functionname*( ) { [native code] }”
Number	returnerar värdet som en sträng (ex 52 returnerar ”52”).
String	returnerar värdet av objektet.
Date	returnerar en sträng som representerar objektets datum och tid.

valueOf

Objekt	**Beteende**
Array	konverterar elementen i arrayen till en sträng separerade av kommatecken.
Boolean	returnerar boolean-värdet.
Function	returnerar funktionen.
Number	returnerar det numeriska värdet.
String	returnerar strängen.
Date	returnerar objektets tid i millisekunder sedan 1 januari 1970 UTC.

Function

Objektet specificerar en sträng ECMAScript-kod som skall kompileras som en funktion. När en funktion exekveras skapas objektet **arguments** som en egenskap till den aktuella funktionen. Objektet har egenskaperna **callee** och **length** (se nedan).

Objektets konstruktor:

```
1.    new Function ([argstr1[, argstr[, ... argstrN]],] functionBody)

2.    function name([argument1[, argument2[, ... argumentN]]])
      {
            statements
      }
```

Nedanstående två exempel skapar samma funktion:

```
1.    var name = new Function (“x”, ”y”, “return x * y”)

2.    function name (x, y)
      {
            return x * y
      }
```

Array
Objektet ger möjlighet att arbeta med arrayer.

Objektets konstruktor:
```
1.    new Array (arrayLength)
2.    new Array (element0, element1, ..., elementN)
```

String
Objektet representerar en serie av tecken i en sträng.

Objektets konstruktor:
```
new String (string)
```

Boolean
Objektet hanterar boolean-värden i objektform. Det är mycket viktigt att inte blanda ihop objektet Booleans **true** och **false** med det primitiva värdets **true** och **false**.

Objektets konstruktor:
```
new Boolean (value)
```

Number
Objektet ger möjlighet att arbeta med numeriska värden. Objektets egenskaper tillhör klassen själv och inte individuella Number-objekt.

Objektets konstruktor:
```
new Number (value)
```

Math
Objektet tillhandahåller egenskaper och metoder för matematiska konstanter och funktioner. Objektet har ingen konstruktor. Det går inte att skapa enskilda objekt. Alla egenskaper och metoder är statiska.

Date
Objektet tillhandahåller funktionalitet för behandling av datum och tid med millisekunders precision. Tiden mäts i millisekunder från och med 1 januari 1970 UTC.

Objektets konstruktor:

```
1.    new Date ()
2.    new Date (milliseconds)
3.    new Date (dateString)
4.    new Date (year, month, day[, hour, minutes, seconds, millisec])
```

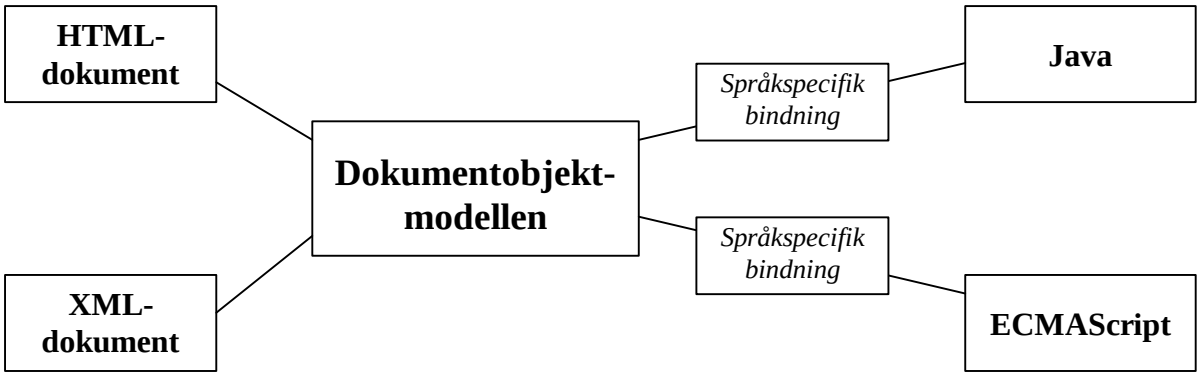
Om ett objekt skapas utan argument antas systemets datum och tid.

Dokumentobjektmodellen

Följande kapitel redogör för hemligheten bakom dynamiken i webbsidor, dvs skriptspråkens koppling till annars statiska webbsidor. Varken JavaScript eller JScript skulle kunna tillföra någon dynamik utan de objekt som tillhandahålls av respektive webbläsare. Objekten är en del av det aktuella dokument som är öppet i webbsidan. Varje webbläsare har en egen modell för att hantera och manipulera de objekt som existerar i ett dokument. Ett av de främsta problemen med **dynamisk HTML** har varit (och är till viss del) att dokument struktureras och hanteras separat i olika webbläsare. För att lösa problemet startades ett arbete i World Wide Web Consortium (W3C) som gick ut på att skapa en objektmodellsstandard för HTML-dokument, som kan manipuleras av ett skriptspråk.

Den 1 oktober 1998 publicerades dokumentobjektmodellen (DOM) i en specifikation från W3C. Modellen är ett programmeringsgränsnitt (API) för HTML- och XML-dokument. Den definierar den logiska strukturen i ett dokument, hur det skall hanteras och på vilket sätt åtkomst skall ske av dokumentet . Med objektmodellen kan programmerare bygga dokument, navigera i dess struktur och lägga till, ändra eller ta bort element och innehåll. Namnet "dokumentobjektmodell" valdes eftersom det är en objektmodell i bemärkelsen av en traditionell objektorienterad design, dvs dokumentet utformas med hjälp av objekt. Vidare omfattar modellen inte bara dokumentets struktur, utan även dess beteende och de objekt av vilka det består. Som objektmodell specificerar den följande (W3C, 1998):

- De **gränssnitt** och **objekt** som används för att representera och manipulera ett dokument.
- **Semantiken** i dessa gränssnitt och objekt, inkluderat både beteende och attribut.
- **Relationerna** mellan gränssnitten och objekten.



Figur 3: Dokumentobjektmodellen

Vidare följer en presentation av World Wide Web Consortium och en sammanfattning av innebörden i "Document Object Model Level 1 Specification".

World Wide Web Consortium

I oktober 1994 bildades World Wide Web Consortium (W3C) i syfte att utveckla Internets mest kända del, webben, till dess fulla potential. Genom att utveckla gemensamma protokoll vill W3C främja webbens framväxt och försöka säkerställa dess interoperatibilitet.

W3C är ett internationellt konsortium i samarbete med Massachusetts Institute of Technology Laboratory for Computer Science (MIT/LCS) i USA, Institut National de Recherde en Informatique et en Automatique (INRIA) i Europa och universitetet Keio i Japan. Konsortiet leds av Tim Berners-Lee, ledare och skapare av World Wide Web, och Jean-François Abramatic, styrelseordförande.

W3C tillhandahåller ett antal allmänna tjänster:

- En förvaring av information om webben för utvecklare och användare, speciellt de officiella specifikationerna.
- Provkodsimplementering för att ge konkret form åt och främja standarder.
- Flera olika prototyper och provapplikationer för att demonstrera användningen av ny teknologi.

Konsortiets stora styrka ligger i den breda tekniska expertis som innehas av dess medlemmar. Medlemskap i W3C är öppet för vilken organisation som helst, som skriver under ett medlemskapsavtal. Konsortiet består idag av ca 300 medlemsorganisationer, vilka bl a inkluderar hård- och mjukvaruförsäljare, telekommunikationsföretag och statliga och akademiska enheter. Verksamheten finansieras av dess medlemmar och är neutral inför skilda medlemsorganisationer och säljande företag. Arbetet sker ur ett globalt perspektiv med målet att producera specifikationer och referenser som är fritt tillgängliga till alla möjliga intressenter. Specifikationer som utvecklas inom W3C måste formellt godkännas av medlemmarna. Samstämmighet nås efter att en specifikation gått igenom tre granskningsstadier: Working Draft, Proposed Recommendation och till sist Recommendation.

I konsortiet finns även en rådgivande kommitté, Consortiums's Advisory Committee (AC), bestående av en officiell representant från varje medlemsorganisation. Kommittén fungerar som den primära förbindelsen mellan medlemsorganisationen och W3C. Dess främsta uppgift är att påverka konsortiets utveckling, dvs utarbeta förslag om verksamhetens framtida inriktning.

Document Object Model Level 1 Specification

Följande kapitel syftar till att ge en översiktsbild av dokumentobjektmodellen som den är specificerad i "Document Object Model Level 1 Specification". Kapitlet baseras på specifikationen, men redogör inte konkret för de specificerade gränssnitt och objekt som modellen tillhandahåller.

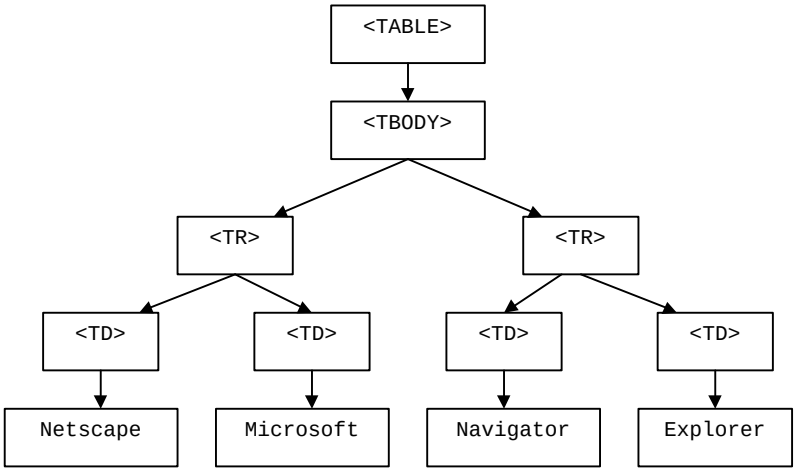
Specifikationen är en W3C-rekommendation. Det innebär att den är slutgiltig och kan implementeras utan oro för att något skall ändras. Ett viktigt mål med dokument-objektmodellen, som med alla W3C-specifikationer, är att tillhandahålla ett standard-programmeringsgränssnitt som kan användas i en mängd olika miljöer och applikationer. Modellen förändrar inga andra standarder, utan har ett rent gränssnitt gentemot andra specifikationer. Dokumentobjektmodellen specificerar gränssnitt som kan användas för att hantera XML och HTML-dokument. Det är viktigt att förstå att dessa gränssnitt är en abstraktion. De är ett hjälpmedel för att specificera en sätt för åtkomst och manipulering av en applikations interna framställning av ett dokument. Gränssnitten förutsätter inte någon speciell konkret implementering. Varje DOM-applikation har fria händer att upprätthålla representationen av dokument på ett lämpligt sätt, så länge de stödjer gränssnitten som presenteras i DOM-specifikationen. Dokumentobjektmodellen är avsedd att undvika imlementeringsberoende:

- De attribut som definieras i specifikationen är inte konkreta objekt.
- Applikationer får tillhandahålla ytterligare gränssnitt och objekt som inte finns i DOM-specifikationen och fortfarande anses DOM-kompatibla.
- Eftersom det är gränssnitten som specificeras och inte de faktiska objekten som skall skapas, så kan inte dokumentobjektmodellen veta vilka konstruktorer som skall åkallas vid implementering. (I allmänhet åkallar DOM-användare createXXX()-metoden i dokumentklassen för att skapa dokumentstrukturer, och DOM-implementeringar skapar deras egna interna motsvarigheter av dessa strukturer i deras implementeringar av createXXX()-funktionerna.)

Dokumentobjektmodellen består för närvarande av två delar, DOM Core (kärnan) och DOM HTML. Kärnan motsvarar funktionaliteten som används för XML-dokument och utgör basen för DOM HTML. I specifikationen ingår även ett appendix för koppling till Java och ECMAScript. De bindningar som specificeras ger information om hur modellen skall implementeras för att kunna hanteras av t ex ett ECMAScript-kompatibelt språk. En överensstämmande implementering av DOM måste tillhandahålla alla de fundamentala gränssnitten i kärnan med tillhörande semantik. Dessutom måste den implementera minst en av DOM HTML- eller XML-gränssnitten med tillhörande semantik.

I modellen presenteras dokument med hjälp av en logisk struktur, vilken kan liknas vid ett träd, eller mer exakt, en samling träd. En tabell i HTML representeras t ex på följande sett i dokumentobjektmodellen:

```
<TABLE>
<TBODY>
<TR>
<TD>Netscape</TD>
<TD>Microsoft</TD>
</TR>
<TR>
<TD>Navigator</TD>
<TD>Explorer</TD>
</TR>
</TBODY>
</TABLE>
```



Figur 4: Tabell i HTML representerad av dokumentobjektmodellen. Källa: W3C, 1998

Modellen specificerar dock inte att dokument måste implementeras i en trädstruktur, ej heller specificerar den hur relationerna mellan objekten skall implementeras. Dokumentobjektmodellen är en logisk modell som kan implementeras på en rad olika lämpliga sätt. I specifikationen används termen strukturmodell (structure model) för att beskriva den trädliknande representationen av objekt (ordet träd används ej med anledning av implementeringsfriheten). En viktig egenskap hos strukturmodeller i DOM är strukturell isomorfism (structural isomorphism). Det innebär att om två implementeringar av dokumentobjektmodellen används för att skapa en framställning av samma dokument, så skapas samma strukturmodell, med exakt samma objekt och relationer.

Nedan följer ett antal punkter som förklarar vad dokumentobjektmodellen **inte** är:

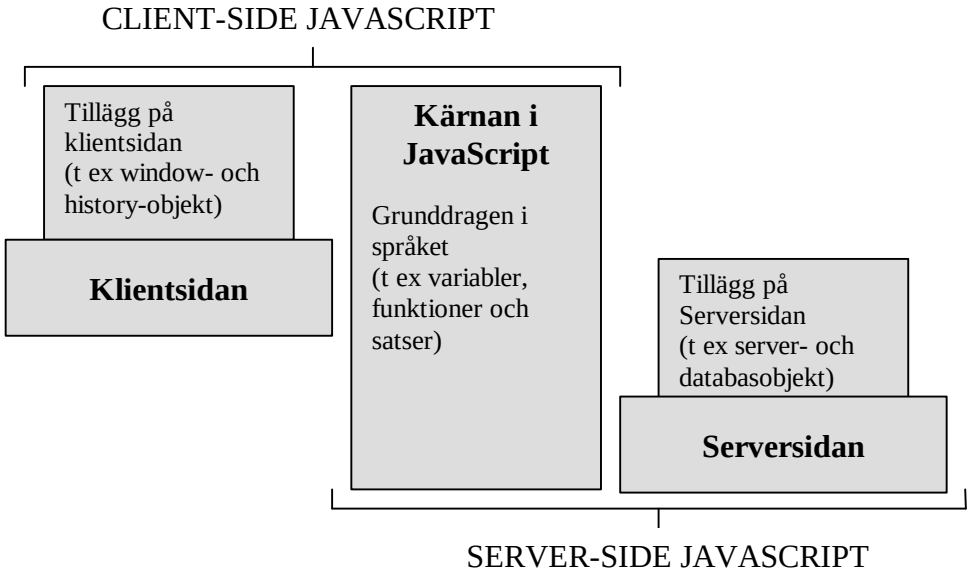
- Trots att dokumentobjektmodellen influerades mycket av **dynamisk HTML** så implementerar den inte allt som ingår i begreppet dynamisk HTML. Det visas kanske allra tydligast med att t ex händelser inte har definierats. Level 1 är dock utformad för att lägga en stadig grund för denna typ av funktionalitet genom att tillhandahålla en robust flexibel modell av dokumentet i sig.
- Dokumentobjektmodellen är inte ett sätt att tillföra objekt till XML eller HTML. Istället för att specificera hur objekt kan representeras i XML eller HTML, så specificerar modellen hur XML och HTML-dokument representeras av objekt, så att de kan användas i objektorienterade program.
- DOM definierar inte “den sanna inre semantiken” i XML eller HTML. Den är definierad av W3C-rekommendationen för respektive språk. Modellen är en programmeringsmodell utformad att respektera den semantiken.
- Dokumentobjektmodellen, trots dess namn, är inte en konkurrent till komponentobjektmodellen (COM). COM, liksom CORBA, är ett språkoberoende sätt att specificera gränssnitt och objekt. DOM är en mängd gränssnitt- och objektspecifikationer utformade i syfte att hantera HTML- och XML-dokument.

W3C arbetar idag på att ta fram Level 2 av specifikationen, vilken kommer att innefatta hantering av CSS (Cascading Style Sheets) som är kopplat till ett HTML- eller XML-dokument. Level 2 är också tänkt att inkludera en modell för händelser (event model), och bättre stöd för queries, samt en mer utvecklad kärna.

JavaScript

1994-1995, långt innan ECMAScript såg dagens ljus, skissade Netscape Corporation på ett skriptspråk som var tänkt att öka interaktiviteten på en webbsida. Produkten kallades LiveScript. Det var planerat att bli en stor nyhet vid presentationen av företagets nya version av webbläsaren Netscape Navigator. För att ge produkten ytterligare understöd beslöt sig Netscape för att licensiera namnet Java från Sun Microsystems och vid publicerandet av Netscape Navigator 2.0 föddes JavaScript (Netscape Communications Corporation, 1998). Den stora nyheten med produkten jämfört med tidigare webbskriptspråk, som t ex CGI[4]-skript, var att koden låg inbäddad i ett HTML-dokument och exekverades på klient-datorn. Tidigare fanns endast möjlighet att exekvera kod med skript på serversidan eller med hjälp av en Java-kompilator på klient-sidan.

JavaScript är ett plattformsoberoende objektorienterat skriptspråk. Det är uppdelat i tre delar: Core JavaScript (språkkärnan), Client-side JavaScript (klientsidan) och Server-side JavaScript (serversidan). Figur 5 illustrerar förhållandet mellan de tre delarna.



Figur 5: Förhållandet mellan kärnan, klientsidan och serversidan i JavaScript. Källa: Netscape Communications Corporation, 1998.

Kärnan innehåller alla nyckelord, språkets syntax och regler för uttryck och variabler. Den tillhandahåller även fördefinierade objekt och funktioner, t ex Array, Date och Math. Kärnan i JavaScript kan kompletteras med ytterligare objekt. På klientsidan utökas språket genom att tillhandahålla objekt som ger kontroll över en webbläsare och dess

[4] Common Gateway Interface. En gränssnittsstandard som tillåter interaktion mellan en webbserver och en klientapplikation. Ett CGI-program exekveras på servern. CGI-skript kan skrivas i vilket programmeringsspråk som helst som hanteras av servern, t ex C/C++, PERL, Visual Basic etc. Vanliga användningsområden är interaktiva formulär, gästböcker, databaskopplingar mm.

dokumentobjektmodell. Exempelvis så tillåter klientsidans utbyggnader en applikation att placera ut element i ett HTML-formulär och reagera på användarhändelser som musklick, input i formulär, och navigation i sidan. Klientapplikationer översätts och exekveras i en webbläsare, som t ex Netscape Navigator. På serversidan utökas språkets kärna genom att tillhandahålla objekt som krävs för att köra JavaScript på en server. Exempelvis så tillåter serversidans utbyggnader en applikation att kommunicera med en relationell databas. Det finns även möjlighet att utföra filhantering. Dessa applikationer exekveras på en server, som t ex Netscape Enterprise Server.

I uppsatsen behandlas endast kärnan i språket och till viss del de ytterligare faciliteter som tillhandahålls av klientsidan. Följande tabell visar vilka värdapplikationer som stödjer vilken version av JavaScript. Tidigare versioner än Navigator 2.0 stödjer inte JavaScript.

Värdapplikationer	Version av JavaScript
Navigator 2.0	1.0
Navigator 3.0	1.1
Navigator 4.0-4.05	1.2
Navigator 4.06-4.5	1.3

Netscape har idag utvecklat en ny kärna, JavaScript 1.4, vilken bl a innehåller stöd för felhantering, ett antal nya operatorer och förändringar av objektet **Function**. Versionen stöds emellertid inte ännu officiellt av någon webbläsare.

Skillnader mellan JavaScript 1.3 och ECMAScript

JavaScript har ett speciellt förhållande till ECMAScript. Det var på Netscapes initiativ som ECMA började arbeta på en standard för ett skriptspråk baserat på främst JavaScript 1.1. Innan ECMA var färdig med sitt arbete hade dock Netscape färdigutvecklat en ny version av sitt skriptspråk, JavaScript 1.2, vilket under rådande omständigheter ej var fullt ut kompatibelt med ECMA-262, när specifikationen officiellt publicerades. Netscape släppte då JavaScript 1.3 som är den nuvarande versionen, och enligt Netscape, fullt kompatibel med ECMA-262. Utöver specifikationen erbjuder JavaScript 1.3 även ytterligare funktionalitet, vilken redogörs för nedan.

Operatorer

JavaScript 1.3 tillhandahåller två ytterligare operatorer. De används i samband med strikt jämförelse av två operander, dvs när det är viktigt att även jämföra om operanderna är av samma typ.

===	returnerar **true** om operanderna är lika och av samma typ. *variable1* **===** *variable2*
!==	returnerar **true** om operanderna inte är lika och/eller inte av samma typ. *variable1* **!==** *variable2*

Satser

JavaScript 1.3 har kompletterats med två vedertagna satser för att kunna tillhandahålla ökad kodflexibilitet. Satserna utnyttjar de framtida reserverade orden i ECMAScript **do** och **switch.** Skriptspråket har även utökats med **label**, **export** och **import**.

do...while	
En do...while-sats exekverar ett kodblock och repeterar sedan exekveringen i en loop, så länge ett givet villkor är uppfyllt.	**do** *statements* **while** (*condition*)

switch	
En switch-sats möjliggör exekvering av en eller fler satser där ett specifikt uttrycks värde matchar en given etikett (label).	**switch** (*expression*) **{** **case** *label* **:** *statementlist* **case** *label* **:** *statementlist* ... **default :** *statementlist* **}**

label	
En etikett (label) låter en sats erhålla en identifierare som kan refereras till från andra ställen i programmet. Används tillsammans med **break**- och **continue**-satser.	*Label* **:** *Statement*

export	
Satsen låter ett signerat skript exportera egenskaper, funktioner och objekt till andra signerade eller osignerade skript.	**export** *name1, name2, ..., nameN* **export** * *nameN* = en lista över de egenskaper, funktioner och objekt som skall exporteras.

import	
Satsen låter ett skript importera egenskaper, funktioner och objekt från ett signerat skript som har exporterat information enligt ovan.	**import** *objectName.name1, objectName.name2,* *..., objectName.nameN* **import** * *nameN* = en lista över de egenskaper, funktioner och objekt som skall importeras till ett givet mottagande objekt.

Egenskaper och metoder

JavaScript 1.3 har utökat vissa av de inbyggda objekten i ECMAScript med nya egenskaper och metoder. De objekt som förändrats är **Array**, **String**, **Function och Object**. Den ytterligare funktionalitet som tillförts presenteras i bilaga 4

Objekt

JavaScript har tillfört ett nytt objekt till språket.

RegExp

Objektet möjliggör sökning i strängar enligt ett mönster baserat på **regular expressions**[5]. Det har egenskaper och metoder som kan användas för att hitta och ersätta matchningar i en sträng.

Utöver de egenskaper som tillhör ett egenskapat objekt, har det fördefinierade **RegExp**-objektet statiska egenskaper som sätts när ett **regular expression** används. Objektet kan inte skapas direkt, men är alltid tillgängligt för användning. Dess egenskaper har **undefined** som värde tills en lyckad sökning har utförts.

Objektet finns även tillgängligt i JScript 3.0, men med vissa skillnader. Objektets egenskaper och metoder finns tillgängliga i bilaga 4 och alla specialtecken som används vid sökning med hjälp av **regular expressions** presenteras i fallstudien.

Objektets konstruktor:

```
1.    new RegExp ("pattern" [, "flags"])
2.    /pattern/flags
```

Följande flaggor kan specificeras:

- g: global match
- i: ignore case
- gi: både global match och ignore case

[5] Uttryck baserat på ett mönster där specialtecken används för att symbolisera en viss typ av tecken eller serie av tecken.

JScript

JScript 3.0 är Microsofts implementering av språkspecifikationen ECMA 262. Det är en fullständig implementering plus ytterligare utbyggnader för att ta fördel av de faciliteter som Microsoft Internet Explorer har. Microsoft understryker att JScript inte är en nedbantad version av något annat språk (det är endast avlägset och indirekt relaterat till t ex Java). Det är dock ett begränsat språk. Det går ej att skriva fristående applikationer i JScript och det har liten förmåga att läsa och skriva filer. Dessutom kan JScript endast köras i samband med en översättare, antingen via en webbserver eller webbläsare (Microsoft Corporation, 1998).

Första officiella versionen, JScript 1.0, presenterades i samband med Microsoft Internet Explorer 3.0. Utvecklingen av skriptspråket var en direkt följd av den hårda konkurrens som rådde på webbläsarmarknaden, främst Netscapes succé med JavaScript. Första versionen av JScript baserades på Netscapes JavaScript. Nedan följer en tabell som visar utvecklingen av JScript i förhållande till kompatibla värdapplikationer.

Värdapplikationer	Version av JScript				
	1.0	2.0	3.0	4.0	5.0
Microsoft Internet Explorer 3.0	x				
Microsoft Internet Information Server 1.0		x			
Microsoft Internet Explorer 4.0			x		
Microsoft Internet Information Server 4.0			x		
Microsoft Windows Scripting Host 1.0			x		
Microsoft Visual Studio 6.0				x	
Microsoft Internet Explorer 5.0					x

Den version som redogörs för i uppsatsen är JScript 3.0. Microsoft har idag givit ut JScript 4.0 anpassad till Microsoft Visual Studio 6.0, och senast JScript 5.0 tillsammans med Microsoft Internet Explorer 5.0 och Microsoft Internet Information Server 5.0.

JScript 3.0 stöds i både 16- och 32-bitars Windowsmiljö och på Macintosh- och UNIX-plattformar. Det finns fyra separata klasser av objekt tillgängliga i JScript:

- Inbyggda objekt
- Användardefinierade skriptobjekt
- Objekt i Internet Explorer
- Övriga objekt i webbsidan

JScript-motorn tillhandahåller kärnfunktionaliteten vid exekvering, vilket även innefattar de inbyggda objekten **Math**, **Date**, **String** etc. Ytterligare objekt kan tillföras av användaren som skapar skriptet. Majoriteten av objekt som används i skripten tillhandahålls dock av Internet Explorer, t ex **window**, **document** etc. Allt som är specifikt för Internet är tillgängligt via Internet Explorer. Den som skriver webbsidan kan även tillföra ytterligare objekt med hjälp av taggen <**OBJECT**>.

Skillnader mellan JScript 3.0 och ECMAScript

Enligt Microsoft är JScript 3.0 fullt kompatibelt med ECMAScript-standarden. Det innebär att JScript kan antas uppfylla de krav som ECMA ställer på en regelrätt implementering av standarden (se kapitel 2.2.2). Utöver den funktionalitet som ECMA-262 tillhandahåller, har dock JScript 3.0 inslag som Microsoft anser kan komma att krävas i en framtida ECMAScript-specifikation. Det medför nya operatorer, uttryck, satser, nya objekt och ytterligare funktionalitet i språkets inbyggda objekt.

Uttryck och operatorer

En av de funktionaliteter Microsoft anser krävas i en kommande ECMAScript-specifikation är villkorlig kompilering (conditional compilation). Villkorlig kompilering tillhandahåller möjligheten att använda olika kodvägar beroende på vissa bestämda variabler. Det innebär att det på ett enkelt sätt går att skriva skript som är avsedda för en speciell plattform och/eller webbläsare.

Stommen i villkorlig kompilering utgörs av tre kommandon:

`@cc_on`	aktiverar villkorlig kompilering i skriptmotorn.
`@if`	exekverar en grupp kodsatser baserat på ett villkor.
`@set`	skapar variabler som används i samband med villkorlig kompilering.

Följande fördefinierade variabler är tillgängliga:

`@_win32`	returnerar **true** om datorn körs under ett Win32-system.
`@_win16`	returnerar **true** om datorn körs under ett Win16-system.
`@_mac`	returnerar **true** om datorn körs under ett Apple Macintosh-system.
`@_alpha true`	returnerar **true** om datorn körs med DEC Alpha-processor.
`@_x86 true`	returnerar **true** om datorn körs med Intel-processor.
`@_mc680x0 true`	returnerar **true** om datorn körs med Motorola 680x0-processor.
`@_PowerPC true`	returnerar **true** om datorn körs med Motorola PowerPC processor.
`@_jscript`	returnerar alltid **true**.
`@_jscript_build`	returnerar konstruktionsnumret (the build number) på JScript-motorn.
`@_jscript_version`	returnerar aktuell JScript-version (ex 3.0).

JScript 3.0 tillhandahåller, likt JavaScript, operatorer för strikt jämförelse av två operander.

`===`	returnerar **true** om operanderna är lika och av samma typ. *`variable1 === variable2`*
`!==`	returnerar **true** om operanderna inte är lika och/eller inte av samma typ. *`variable1 !== variable2`*

Satser

Även JScript 3.0 tillför de ytterligare satserna **do..while**, **switch** och **label**.

Egenskaper och metoder

JScript 3.0 har utökats med nya egenskaper och metoder för de inbyggda objekten **Array**, **String**, **Function** och **Date,** samt det globala objektet. I bilaga 5 följer en sammanställning av den ytterligare funktionaliteten.

Objekt

JScript har tillfört tre nya objekt till språket. Objektens egenskaper och metoder finns att tillgå i bilaga 5.

RegExp

Objektets konstruktor:

```
1.    new RegExp ("pattern" [, "flags"])
2.    /pattern/flags
```

Följande flaggor kan specificeras:

- g: global match
- i: ignore case
- gi: både global match och ignore case

VBArray

Objektet ger tillgång till Visual Basic:s ”säkra” arrayer. Objekt av typen VBArray är endast läsbara och kan inte skapas direkt. Argumentet måste ha erhållit ett VBArray-värde innan det kan skickas vidare till VBArray-konstruktorn. Det kan endast utföras genom att återfå värdet från ett existerande ActiveX- eller annat objekt.

Objektets konstruktor:

```
new VBArray(safeArray)
```

Enumerator

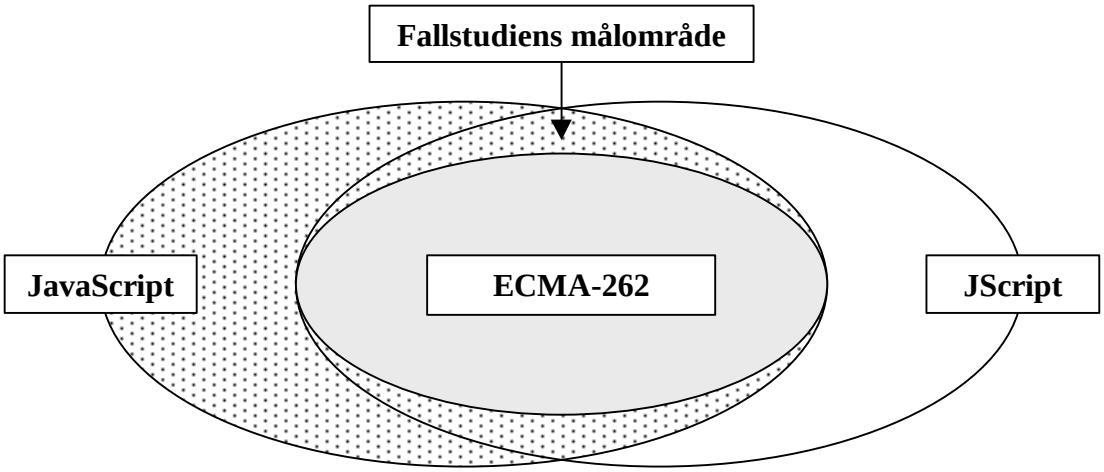
Objektet möjliggör uppräknande av element i en samling (**collection** = ett objekt som består av en samling poster). En samling skiljer sig från arrayer på det sätt att elementen inte är direkt åtkomliga. Istället för att använda index, som med arrayer, så kan man endast peka på första eller nästa element i samlingen.

Objektets konstruktor:

```
new Enumerator(collection)
```

FALLSTUDIE

I ECMAScript-standarden specificeras hur varje uttryck, sats, objekt, metod och egenskap skall implementeras i ett skriptspråk. JavaScript 1.3 och JScript 3.0 är fullt ut kompatibla med ECMA-262 och kan därmed antas vara implementerade på samma sätt och framförallt producera samma resultat. Utöver det som står specificerat i standarden tillhandahåller emellertid de båda skriptspråken ytterligare funktionalitet. En viss del av denna funktionalitet går att identifiera i båda skriptspråken. Det finns dock inga garantier för att de är implementerade på samma sätt och framförallt inte att de producerar samma resultat. Syftet med fallstudien är att identifiera och testa de uttryck, satser, objekt, metoder och egenskaper som faller innanför ramen för båda skriptspråken och som ej finns specificerade i ECMAScript. Resultatet av fallstudien fullbordar underlaget för en analys, där huvudsakliga likheter och olikheter mellan de båda skriptspråken kan identifieras.



Figur 6: Fallstudiens målområde.

Följande funktionalitet, identifierad i båda skriptspråken, ingår i fallstudien:

- Strikt jämförelse (`===` och `!==`)
- **do...while** och **switch**
- **Array** – ytterligare metoder
- **Function** – egenskapen caller
- **String** – ytterligare metoder
- **RegExp** – nytt objekt

Strikt jämförelse

Både JavaScript 1.3 och JScript 3.0 tillhandahåller operatorer för strikt jämförelse. Det innebär att vid jämförelse tas hänsyn även till om variablerna är av samma typ.

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      var test1 = false;
      var test2 = false;
      var test3 = false;
      var test4 = false;
      var test5 = false;
      var test6 = false;

      var aBoolean = true;
      var aString = "test";
      var aNumber = 1;

      if (aBoolean === "true") test1 = true;
      if (aBoolean === true) test2 = true;
      if (aString !== "test") test3 = true;
      if (aString !== 10) test4 = true;
      if (aNumber === "1") test5 = true;
      if (aNumber === true) test6 = true;

</SCRIPT>
</HEAD>
</HTML>
```

Tre olika typer av variabler deklareras:
- boolean
- string
- number

Testvariabler sätts till det värde (**true** eller **false**) som respektive villkorstest resulterar i.

Resultat:	test1	test2	test3	test4	test5	test6
JavaScript 1.3	`false`	`true`	`false`	`true`	`false`	`false`
JScript 3.0	`false`	`true`	`false`	`true`	`false`	`false`

do...while och switch

Båda satserna finns att tillgå i de flest moderna programmeringsspråk. Satsen **do...while** exekverar en rad satser en gång och repeterar sedan exekveringen i en loop tills att ett givet villkor resulterar i värdet **false**. Satsen **switch** möjliggör exekvering av en eller flera satser där ett specificerat värde matchar en **label**.

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      var testVar=1;

      do
      {
            document.writeln(testVar);
            testVar++;
      }
      while (testVar<5)

      do
      {
            document.writeln(testVar);
            testVar++;
      }
      while (testVar<5)

</SCRIPT>
</HEAD>
</HTML>
```

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      var testVar="ECMAScript";

      switch (testVar)
      {
            case "JavaScript" :
                  document.writeln("Netscape");
                  break;
            case "JScript" :
                  document.writeln("Microsoft");
                  break;
            case "ECMAScript" :
                  document.writeln("ECMA");
                  break;
            default :
                  document.writeln("No Script!");
      }

</SCRIPT>
</HEAD>
</HTML>
```

Resultat:	JavaScript 1.3	JScript 3.0
do...while	1 2 3 4 5	1 2 3 4 5
switch	ECMA	ECMA

Array - concat och slice

De båda skriptspråken tillhandahåller två ytterligare metoder till objektet **Array**. Den första, **concat**, skiljer sig åt (i respektive referensmaterial) genom att JScript returnerar en ny array bestående av två arrayer och JavaScript returnerar en ny array bestående av två eller fler arrayer. Den andra, **slice**, returnerar en ny array bestående av en del av en given array.

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      array1=new Array("JavaScript", "JScript");
      array2=new Array("Netscape", "Microsoft");
      array3=new Array("Java", "Kaffe");
      arrayConcat=new Array

      function testConcat1()
      {
            arrayConcat=array1.concat(array2);
            alert(arrayConcat.toString());
      }

      function testConcat2()
      {
            arrayConcat=array1.concat(array2, array3);
            alert(arrayConcat.toString());
      }

</SCRIPT>
</HEAD>
<BODY>
      <FORM name="testForm">

            <INPUT type=button id=btnTest1
                  value="Concat 2 arrays"
                  onClick="testConcat1()">

            <INPUT type=button id=btnTest2
                  value="Concat 3 arrays"
                  onClick="testConcat2()">
      </FORM>
</BODY>
</HTML>
```

Funktionen testConcat1() tilldelar arrayConcat en array bestående av array1 och array2.

Funktionen testConcat2() tilldelar arrayConcat en array bestående av array1, array2 och array3.

Två knappar skapas.

- btnTest1 exekverar funktionen testConcat1() vid knapptryck.
- btnTest2 exekverar funktionen testConcat2() vid knapptryck.

Resultat:	JavaScript 1.3	JScript 3.0
testConcat1	`JavaScript,JScript, Netscape,Microsoft`	`JavaScript,JScript, Netscape,Microsoft`
testConcat2	`JavaScript,JScript, Netscape,Microsoft, Java,Kaffe`	`JavaScript,JScript, Netscape,Microsoft, Java,Kaffe`

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      array1=new Array("JavaScript", "JScript"
                       ,"Netscape", "Microsoft"
                       ,"Java", "Kaffe");
      arraySlice=new Array

      function testSlice1()
      {
            arraySlice=array1.slice(3);
            alert(arraySlice.toString());
      }

      function testSlice2()
      {
            arraySlice=array1.slice(1,4);
            alert(arraySlice.toString());
      }

</SCRIPT>
</HEAD>
<BODY>
      <FORM name="testForm">

            <INPUT type=button id=btnTest1
                  value="Slice, start=3"
                  onClick="testSlice1()">

            <INPUT type=button id=btnTest2
                  value="Slice start=1, end=4"
                  onClick="testSlice2()">
      </FORM>
</BODY>
</HTML>
```

Funktionen testSlice1() tilldelar arraySlice en array bestående av array1 från index 3 till slutet av arrayen.

Funktionen testSlice2() tilldelar arraySlice en array bestående av array1 från index 1 till index 4 (ej inklusive).

Två knappar skapas.

- btnTest1 exekverar funktionen testSlice1() vid knapptryck.
- btnTest2 exekverar funktionen testSlice2() vid knapptryck.

Resultat:	JavaScript 1.3	JScript 3.0
testSlice1	`Microsoft,Java,Kaffe`	`Microsoft,Java,Kaffe`
testSlice2	`JScript,Netscape,Microsoft`	`JScript,Netscape,Microsoft`

Function - caller

Studien av följande egenskap skiljer sig från övriga. Både JavaScript 1.3 och JScript 3.0 specificerar egenskapen **caller**, men på olika sätt. I JScript är egenskapen placerad direkt under det aktuella **Function**-objektet, medan JavaScript placerar egenskapen under egenskapen **arguments** och meddelar samtidigt att den ej rekommenderas att användas i JavaScript 1.3. Mycket riktigt visar det sig att den ej fungerar tillfredsställande i JavaScript 1.3, men av någon anledning går det utmärkt att använda egenskapen på samma sätt som i JScript, dvs direkt under **Function**-objektet.

Egenskapen **caller** returnerar en referens till den funktion som anropade den aktuella funktionen. Egenskapen är endast definierad medan den aktuella funktionen exekveras.

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      function testCaller()
      {
            alert(testCaller.caller);
      }

      function callTestCaller()
      {
            testCaller();
      }

      testCaller();

</SCRIPT>
</HEAD>
<BODY>
      <FORM name="testForm">

            <INPUT type=button id=btnTest1
            value="Call function from event"
                  onClick="testCaller();">

            <INPUT type=button id=btnTest2
            value="Call function from function"
                  onClick="callTestCaller();">

      </FORM>
</BODY>
</HTML>
```

Huvudfunktionen testCaller() visar en meddelanderuta innehållande den sträng som returneras från metoden testCaller.caller.

Funktionen callTestCaller() exekverar funktionen testCaller().

Satsen testCaller() exekverar globalt funktionen testCaller() vid programmets start.

Två knappar skapas.

- btnTest1 exekverar funktionen testCaller() vid knapptryck.
- btnTest2 exekverar funktionen callTestCaller() vid knapptryck.

Resultat:	JavaScript 1.3	JScript 3.0
Globalt	`null`	`null`

Från annan funktion	`function callTestCaller(){`
	`  testCaller();`
	`}`
		`function callTestCaller()`
		`{`
		`testCaller()`
		`}`

Från en händelse	`function onclick(event){`
	`  testCaller();`
	`}`
		`function anonymous()`
		`{`
		`testCaller()`
		`}`

String

Först följer en studie av de metoder som är direkt kopplade till HTML. De har funnits tillgängliga sedan första versionen av respektive språk, men är ej specificerade i ECMAScript-standarden på grund av dess koppling till HTML. Sedan följer kodexempel för **concat**, **slice** och **substr**. Metoderna **match**, **replace** och **search** studeras under objektet **RegExp**.

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      var test1=new String("test1");
      var test2=new String("test2");
      var test3=new String("test3");

      test1=test1.bold();
      test1=test1.italics();
      test1=test1.fixed();
      test1=test1.fontsize(16);
      test1=test1.fontcolor("green");
      test1=test1.link("http://www.altavista.com");

      test2=test2.blink();
      test2=test2.sub ();
      test2=test2.big();
      test2=test2.anchor("ancTest");

      test3=test3.small();
      test3=test3.sup();
      test3=test3.strike();

      alert(test1);
      alert(test2);
      alert(test3);

</SCRIPT>
</HEAD>
</HTML>
```

Res:	JavaScript 1.3	JScript 3.0
test1	<A HREF=”http://www.altavista.com”><FONT COLOR=”green”><FONT SIZE=”16”><TT><I><B>test1</B></I></TT></FONT></FONT></A>	<A HREF=”http://www.altavista.com”><FONT COLOR=”green”><FONT SIZE=”16”><TT><I><B>test1</B></I></TT></FONT></FONT></A>
test2	<A NAME=”ancTest”><BIG>_{<BLINK>test2</BLINK>}</BIG></A>	<A NAME=”ancTest”><BIG>_{<BLINK>test2</BLINK>}</BIG></A>
test3	<STRIKE>^{<SMALL>test3</SMALL>}</STRIKE>	<STRIKE>^{<SMALL>test3</SMALL>}</STRIKE>

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      string1=new String("JavaScript");
      string2=new String(", JScript");
      string3=new String(", Java");
      stringConcat=new String

      function testConcat1()
      {
            stringConcat=
                  string1.concat(string2);
            alert(stringConcat.toString());
      }

      function testConcat2()
      {
            stringConcat=
                  string1.concat(string2,string3);
            alert(stringConcat.toString());
      }

</SCRIPT>
</HEAD>
<BODY>
      <FORM name="testForm">

            <INPUT type=button id=btnTest1
                  value="Concat 2 strings"
                  onClick="testConcat1()">

            <INPUT type=button id=btnTest2
                  value="Concat 3 strings"
                  onClick="testConcat2()">
      </FORM>
</BODY>
</HTML>
```

Funktionen testConcat1() tilldelar stringConcat en sträng bestående av string1 och string2.

Funktionen testConcat2() tilldelar stringConcat en sträng bestående av string1, string2 och string3.

Två knappar skapas.

- btnTest1 exekverar funktionen testConcat1() vid knapptryck.
- btnTest2 exekverar funktionen testConcat2() vid knapptryck.

Resultat:	JavaScript 1.3	JScript 3.0
testConcat1	`JavaScript,JScript`	`JavaScript,JScript`
testConcat2	`JavaScript,JScript, Java`	`JavaScript,JScript, Java`

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      testString=new String("JavaScript, JScript
                        Netscape, Microsoft");
      stringSlice=new String

      function testSlice1()
      {
            stringSlice=testString.slice(21);
            alert(stringSlice.toString());
      }

      function testSlice2()
      {
            stringSlice=testString.slice(12,19);
            alert(stringSlice.toString());
      }

</SCRIPT>
</HEAD>
<BODY>
      <FORM name="testForm">

            <INPUT type=button id=btnTest1
                  value="Slice, start=21"
                  onClick="testSlice1()">

            <INPUT type=button id=btnTest12
                  value="Slice start=1, end=19"
                  onClick="testSlice2()">
      </FORM>
</BODY>
</HTML>
```

Funktionen testSlice1() tilldelar stringSlice en sträng bestående av testString från index 21 till slutet av strängen.

Funktionen testSlice2() tilldelar stringSlice en sträng bestående av testString från index 12 till index 19 (ej inklusive).

Två knappar skapas.

- btnTest1 exekverar funktionen testSlice1() vid knapptryck.
- btnTest2 exekverar funktionen testSlice2() vid knapptryck.

Resultat:	JavaScript 1.3	JScript 3.0
testSlice1	`Netscape, Microsoft`	`Netscape, Microsoft`
testSlice2	`JScript`	`JScript`

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      testString=new String("JavaScript, JScript,
                        Netscape, Microsoft");
      aSubString=new String;

      function testSubStr1()
      {
            aSubString=testString.substr(11);
            alert(aSubString.toString());
      }

      function testSubStr2()
      {
            aSubString=testString.substr(0,10);
            alert(aSubString.toString());
      }

</SCRIPT>
</HEAD>
<BODY>
      <FORM name="testForm">

            <INPUT type=button id=btnTest1
                  value="substr start=11"
                  onClick="testSubStr1()">

            <INPUT type=button id=btnTest12
                  value="substr start=0 length=10"
                  onClick="testSubStr2()">
      </FORM>
</BODY>
</HTML>
```

Funktionen testSubStr1() tilldelar aSubString en sträng bestående av testString från index 11 till slutet av strängen.

Funktionen testSubStr2() tilldelar aSubString en sträng bestående av testString från index 0 med längden 10.

Två knappar skapas.

- btnTest1 exekverar funktionen testSubStr1() vid knapptryck.
- btnTest2 exekverar funktionen testSubStr2() vid knapptryck.

Resultat:	JavaScript 1.3	JScript 3.0
testSubStr1	`JScript, Netscape, Microsoft`	`JScript, Netscape, Microsoft`
testSubStr2	`JavaScript`	`JavaScript`

RegExp

Båda skriptspråken tillhandahåller objektet **RegExp** som används för att söka i och manipulera strängar med hjälp av **regular expressions**. Förutom de egenskaper och metoder som tillhör själva objektet tillhandahåller skriptspråken metoderna **match**, **replace** och **search** tillhörande objektet **String**. Nedan följer ett utdrag ur studien av de viktigaste funktionerna i objektet, men först en lista över alla de specialtecken som finns tillgängliga i **regular expressions**.

Specialtecken

****	indikerar att nästa tecken skall behandlas annorlunda. Om tecknet i vanliga fall behandlas bokstavligt, så indikerar det att tecknet är ett specialtecken. Om tecknet i vanliga fall behandlas som ett specialtecken, så indikerar det att tecknet skall tolkas bokstavligt.
^	matchar början av inmatningen. `ex. "^A" ger ingen träff i "an A", men i "An A".`
$	matchar slutet av inmatningen.
*****	matchar det föregående tecknet 0 eller mer gånger. `ex. "zo*" matchar antingen "z" eller "zoo".`
+	matchar det föregående tecknet 1 eller mer gånger. `ex. "zo+" matchar "zoo", men inte "z".`
?	matchar det föregående tecknet 0 eller 1 gång. `ex. "e?le?" ger träff i "angel" och i "angle".`
.	matchar ett enskilt tecken förutom tecknet för ny rad.
(*x*)	matchar **x** och håller träffen i minnet.
x|*y*	matchar antingen **x** eller **y**.
{*num*}	matchar exakt **n** förekomster av det föregående tecknet. `ex. "a{2}" matchar "aa" i "yeaaaaah", men inget i "yeah".`
{*num*,}	matchar minst **n** förekomster av det föregående tecknet.
{*n*,*m*}	matchar minst **n** och max **m** förekomster av det föregående tecknet.
[*xyz*]	matchar vilken som helst av **xyz**, där **xyz** är en grupp av tecken. Ett bindestreck kan användas för att symbolisera en rad tecken. `ex. "[a-f]" matchar "e" i "test".`
[^*xyz*]	matchar vilket tecken som helst som inte är **xyz**.
\b	matchar gränslinjer i ord, t ex mellanslag. `ex. "er\b" matchar "er" i "never" men inte "er" i "verb".`
\B	matchar en icke-gränslinje i ett ord. `ex. "er\b" matchar "er" i "verb" men inte i "never".`
\d	matchar en siffra.
\D	matchar ett tecken som inte är en siffra.
\f	matchar ett tecken av typen **form-feed.**
\n	matchar ett tecken av typen **linefeed**.
\r	matchar ett tecken av typen **carriage return**.
\s	matchar ett tomt tecken. Samma som `[\f\n\r\t\v]`.
\S	matchar ett tecken som inte är ett tomt tecken.
\t	matchar ett tecken av typen **tab**.

\v	matchar ett tecken av typen **vertical tab**.
\w	matchar vilket alfanumeriskt tecken som helst inkluderat **underscore**.
\W	matchar ett tecken som inte är alfanumeriskt.

Metoden **replace** används nedan för att vända på ett namn och placera efternamnet först och förnamnet sist. Observera användningen av minnesegenskaperna $1 och $2.

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      re = /(\w+)\s(\w+)/;
      str1 = "John Smith";
      str2 = "Jörgen Hanson";
      newstr1 = str1.replace(re,"$2, $1");
      newstr2 = str2.replace(re,"$2, $1");
      document.writeln(newstr1 + "<br>" +
                  newstr2 + "<br>");

</SCRIPT>
</HEAD>

</HTML>
```

Resultat:	JavaScript 1.3	JScript 3.0
replace	Smith, John Hanson, Jörgen	Smith, John JöHanson, rgen

Nedan följer två exempel som testar egenskaper och metoder oberoende av om de endast tillhör ett av skriptspråken eller båda. Syftet är att försöka ge en bild av hur stor skillnaden det är mellan implementeringen av objektet i JavaScript respektive JScript.

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      re = /e(f+)(ed)/ig;
      arr  = re.exec("ABCDEFfedcba");

</SCRIPT>
</HEAD>
</HTML>
```

Resultat:	JavaScript 1.3	JScript 3.0
arr	EFfed,Ff,ed	EFfed,Ff,ed
arr.index	4	4
arr.input	ABCDEFfedcba	ABCDEFfedcba
re.lastindex	9	undefined
re.ignoreCase	true	undefined
re.global	true	undefined
re.source	e(f+)(ed)	e(f+)(ed)
RegExp.lastMatch	EFfed	undefined
RegExp.leftContext	ABCD	undefined
RegExp.rightContext	cba	undefined
RegExp.$1,$2	Ff,ed	Ff,ed
RegExp.lastParen	ed	undefined
RegExp.input		ABCDEFfedcba
RegExp.lastIndex	undefined	9

```
<HTML>
<HEAD>
<SCRIPT language="javascript">

      re = /yea{1,5}h!+\s(JavaScript|JScript)/

      str1="ECMAScript yeaaaaah!!! JavaScript, JScript";
      str2="ECMAScript yeah!!! JScript, JavaScript";
      str3="ECMAScript yeaaaaaah!!! JScript, JavaScript";

</SCRIPT>
</HEAD>
</HTML>
```

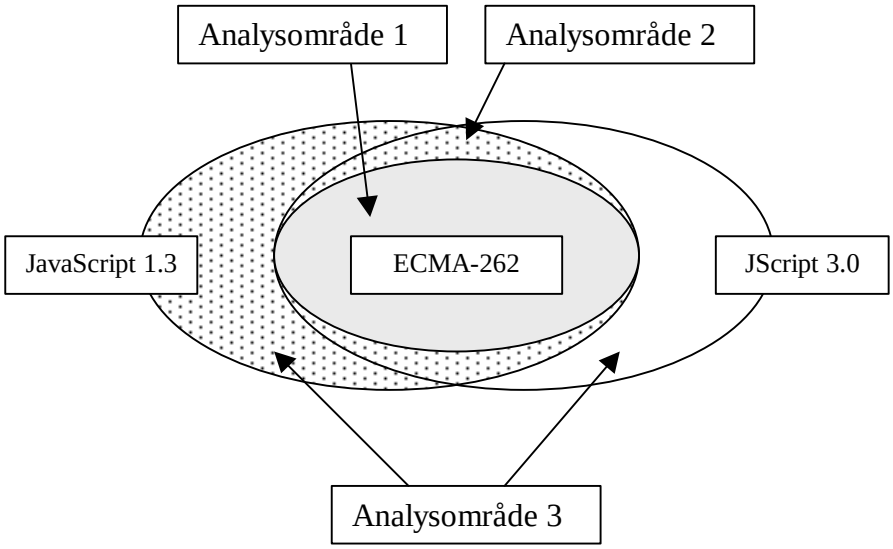
Resultat:	JavaScript 1.3	JScript 3.0
str1.match(re)	yeaaaaah!!! JavaScript, JavaScript	yeaaaaah!!! JavaScript, JavaScript
str2.search(re)	11	11
re.test(str3)	false	false

DISKUSSION

Uppsatsens problem analyseras utifrån tre skilda områden.

- Skriptspråkens likheter:
  Analysområde 1 består av skriptspråksstandarden ECMA-262.
  Analysområde 2 innefattas av fallstudiens målområde.

- Skriptspråkens olikheter:
  Analysområde 3 består av det som är specifikt för respektive skriptspråk.



Figur 7: Tre analysområden.

Analysområde 1

ECMA-262 är den enda standard som finns tillgänglig för webbskriptspråk idag. Netscape och Microsoft gör gällande att JavaScript 1.3 och JScript 3.0 är fullt kompatibla med standarden. Det innebär att de båda skriptspråken har en gemensam grund utarbetad och specificerad av en oberoende part. ECMAScript-standarden tillhandahåller grundläggande syntax och semantik samt de typer, operatorer, uttryck och satser som krävs för att kunna ge skriptspråket stabilitet. Utöver den inre semantiken utgörs den verkliga kraften i skriptspråksstandarden av de inbyggda objekten:

- **Det globala objektet** möjliggör globala egenskaper och metoder.
- **Object** utgör stommen för alla ECMAScript-objekt.
- **Function** specificerar en sträng ECMAScript-kod som kompileras som en funktion.
- **Array, String, Boolean** och **Number** hanterar respektive typer i objektform.
- **Math** innehåller egenskaper och metoder för matematiska konstanter och funktioner.
- **Date** tillhandahåller funktionalitet för behandling av datum och tid.

Vid närmare analys av den enorma utveckling som har skett och sker av webben, märks ett ökat behov av en väl utvecklad skriptspråksstandard. ECMA arbetar idag på en tredje utgåva av standarden och den kommer att innehålla den andra versionen av språket. Mycket tyder på en utveckling där analysområde 1 breder ut sig och stjäl mark från analysområde 2 och 3. En sådan utveckling är framförallt välkomnad av systemutvecklare och övriga användare, men det bör även vara ett mål för företag som Netscape och Microsoft. En bredare standard minskar bl a de utvecklingskostnader som är relaterade till JavaScript och JScript.

I samma utsträckning som JavaScript och JScript baseras sig på ECMAScript bygger standarden på dessa skriptspråk. Mycket av det som går att identifiera i analysområde 2 och 3 kan komma att ligga till grund för fortsatt utveckling av standarden.

Analysområde 2

Utöver de likheter som kan urskiljas i ECMA-262 finns ett område för analys där de båda skriptspråken tillhandahåller samma funktionalitet oberoende av standarden. En viss del av området kan förklaras med att skriptspråken påverkats av varandra. En annan del kan bero på liknande inverkan från andra programmeringsspråk. Ytterligare en del kan bestå av tänkbara framtida krav från skriptspråksstandarden. I fallstudien undersöks alla de likheter som identifierats mellan språken, som ej ingår i ECMA-262. Utan någon standard finns inga garantier för att likheterna är utformade på samma sätt. Det är emellertid enklare att anta att vedertagna uttryck och satser fungerar identiskt. Svårare är det med liknande objekt, egenskaper och metoder. Nedan följer en sammanfattning av fallstudiens resultat:

- Metoderna **concat** och **slice** tillhörande objektet Array fungerar enligt fallstudien identiskt i båda skriptspråken, trots att de specificeras olika i respektive referensmaterial.
- Operatorerna för **strikt jämförelse** och de vedertagna satserna **do...while** och **switch** fungerar identiskt.
- Egenskapen **caller** tillhörande objektet **Function** ger liknande resultat när funktionen åkallas globalt eller från en annan funktion (observera att resultatet inte är identiskt). När den åkallas från en händelse skiljer sig resultatet åt. Det har troligtvis sin förklaring i respektive webbläsares objektmodell och inte i själva egenskapen.
- Alla metoder som tillförts objektet **String** fungerar identiskt i båda skriptspråken.
- Endast ett nytt gemensamt objekt har tillförts både JavaScript och JScript. **RegExp** ger tillgång till sökning och manipulering av strängar med hjälp av **regular expressions**. Det är tydligt att skriptspråken påverkats av ett tredje programmeringsspråk vid implementering av objektet. Skriptspråken tillhandahåller exakt samma konstruktorer och specialtecken. Det märks dock att implementeringen inte är standardiserad, eftersom objektets egenskaper och metoder inte är identiska. Ytterligare en intressant detalj är JScripts hantering av alfanumeriska tecken. Vid omplacering av för- och efternamn fungerar JavaScript även i samband med det svenska namnet ”Jörgen”, medan JScript producerar ett mycket tveksamt resultat där ”ö” inte antas vara ett alfanumeriskt tecken..

Mycket talar för att det som upptas av analysområde 2 kan komma att bli en del av analysområde 1, dvs funktionalitet som uppträder i de båda språken har stor möjlighet att bli en del av skriptspråksstandarden. Det finns ingen anledning för ECMA att avstå från att införa den gemensamma funktionalitet som språken redan har.

Analysområde 3

Utvecklingen har varit sådan att JavaScript och JScript har varit föremål för interna mål hos Netscape respektive Microsoft. I väntan på ytterligare standardisering utvecklas de båda skriptspråken på olika sätt. Ny funktionalitet tillförs och språken integreras med andra produkter och målsättningar i respektive företag. Ett typiskt exempel är JScripts tillförande av metoden **getVarDate** i **Date** och objektet **VBArray**, som båda används för integration med andra produkter från Microsoft.

Ytterligare funktionalitet, som endast finns tillgänglig i JScript, är villkorlig kompilering, objektet **Enumerator** och globala funktioner som ger information om aktuell skriptmotor. JavaScript tillför egen funktionalitet genom ytterligare metoder till de inbyggda objekten **Array**, **Function** och **Object** och införandet av **import**- och **export**-satser.

Den mest påtagliga skillnaden mellan de båda skriptspråken finns dock ej i språkens kärna, utan snarare i den miljö i vilken de verkar. Netscape specificerar JavaScript gemensamt med webbläsaren Navigator och dess objektmodell. Microsoft, däremot, ser JScript som bara en av flera produkter som kan användas i kombination med objektmodellen i Internet Explorer. Det finns både fördelar och nackdelar med respektive företags strategi. Eftersom JavaScript är det enda skriptspråk Netscape förfogar över, är de mycket måna om dess framtid och det faktum att ECMAScript utvecklas fortlöpande. Microsoft är inte låsta vid endast ett skriptspråk, utan innehar en rad teknologier för webbutveckling. Den stora fördelen med Microsoft är dess arbete med dokumentobjektmodellen i Internet Explorer. Ett skriptspråks prestanda är i allra högsta grad beroende av den miljö i vilken det verkar. Mycket tyder på att även DOM-standarden kommer att utvecklas ytterligare och framförallt utökas med en modell för händelser, vilket är den mest uppenbara kopplingen mellan ett dokument och ett skriptspråk.

SLUTSATSER

JavaScript 1.3 och JScript 3.0 är trots inkompatibilitetsproblematiken två mycket lika skriptspråk. Den främsta anledningen till denna slutsats är att grundstommen i språken finns specificerade i skriptspråksstandarden ECMA-262. Standarden tillhandahåller grundläggande syntax och semantik i språken samt typer, operatorer, uttryck, satser och ett antal inbyggda objekt.

Utöver det som kan identifieras i standarden finns ytterligare liknande funktionalitet i skriptspråken. Båda språken har tillfört det nya objektet **RegExp** och nya egenskaper och metoder till de inbyggda objekten **Array**, **String** och **Function**. De har även valt att tillhandahålla operatorer för strikt jämförelse och satserna **do...while** och **switch**. Majoriteten av den ytterligare funktionalitet som är lika i båda språken är även identisk, dvs fungerar på samma sätt och ger samma resultat. Ett undantag är de skillnader som identifierats i objektet **RegExp** och egenskapen **caller**. Sådana skillnader stärker behovet av en väl utvecklad standard vid införandet av ny funktionalitet i de båda skriptspråken.

De olikheter som identifierats mellan skriptspråken består av ny funktionalitet i respektive språk. Den mest påtagliga skillnaden finns dock ej i språkens kärna, utan snarare i den miljö i vilken de verkar. Kompatiblitetsproblematiken är främst ett resultat av de skillnader som kan härledas till respektive webbläsares objektmodell.

Mycket tyder på att ECMAScript-standarden kommer att utvecklas till den grad att skillnaden mellan JavaScript och JScript blir försumbar. Om dokumentobjektmodellen utvecklas på samma sätt och implementeras i respektive webbläsare, kommer skriptspråksproblematiken i framtiden inte att existera. Förhoppningsvis är det ett faktum redan inom fem år.

KÄLLFÖRTECKNING

Backman, J. (1998). *Rapporter och uppsatser*. Lund: Studentlitteratur.

ECMA. (1998). *Standard ECMA-262 2nd Edition.*
[www dokument]. URL http://www.ecma.ch

Microsoft Corporation. (1998). *Windows Script Technologies JScript*
[www dokument]. URL http://msdn.microsoft.com/scripting/

Netscape Communications Corporation. (1998). *Client-Side JavaScript Reference 1.3*
[www dokument]. URL http://developer.netscape.com/docs/manuals/js/client/jsref

Netscape Communications Corporation. (1998). *Client-Side JavaScript Guide 1.3*
[www dokument]. URL http://developer.netscape.com/docs/manuals/js/client/jsguide

W3C. (1998). *Document Object Model Level 1 Specification.*
[www dokument]. URL http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001

Ordlista

Engelsk term	**Svensk översättning**
argument	argument
assignmentExpression	uttryck som tilldelar en variabel ett värde
condition	villkor
constructor	konstruktor
element	element, post
expression	uttryck

formalParameterList	en rad med identifierare:
	`syntax: identifier`
	`formalParameterList, identifier`

handler	hanteringsfunktion
identifier	identifierare, en referens till ett objekt
initializer	tilldelare av ett initialt värde `syntax: =assignmentExpression`
label	etikett
native code	kod tillhörande språket som är dolt för användaren.
pattern	mönster
property	egenskap
separator	skiljetecken (t ex kommatecken)
statement	sats
statementList	en rad med en eller flera satser
value	värde
variable	variabel
variableDeclaration	deklaration av en variabel: `syntax: identifier [initializer]`
variableDeclarationList	en rad med en eller flera variabeldeklarationer.

Uttryck i ECMAScript

Primary Expressions	
this	refererar till det aktuella objektet. I en funktion refereras i allmänhet till det kallande objektet. Ett annat vanligt användningsområde är i en objektkonstruktor för att referera till det aktuella objektet. ex. <INPUT TYPE="*text*" NAME="*name*" onChange="*test(*this*);*">
Left-Hand-Side Expressions	
new	skapar en instans av en användardefinierad objekttyp eller en av de inbyggda objekttyperna som har en konstruktor. **new** *constructor*[(*argument*)]
Postfix Expressions	
++	ökar en given variabel med 1. *variable* **++**
--	minskar en given variabel med 1. *variable* **--**
Unary operators	
delete	tar bort ett objekt, en egenskap eller ett element i en array.
void	utvärderar givet uttryck och returnerar "undefined". **void** [()]*expression*[)]
typeof	returnerar en sträng som visar uttryckets datatyp. **typeof** [()]*expression*[)]
++	ökar en given variabel med 1. **++** *variable*
--	minskar en given variabel med 1. **--** *variable*
+	konverterar operanden till ett tal. **+** *variable*
-	konverterar operanden till ett tal och negerar det. **-** *variable*
~	inverterar bitarna hos en given operand ~*variable* ex. ett binärt tal 00100100 ger 11011011
!	returnerar **false** om uttrycket kan konverteras till **true**, annars returneras **true**. !*expression*
Multiplicative operators	
*****	multiplicerar två tal. a * b
/	dividerar två tal (flyttalsdivision). a / b
%	utför en heltalsdivision och returnerar restvärdet. a % b ex. 12 % 5 ger 2

Additive operators	
+	adderar två tal. `a + b`
-	subtraherar två tal. `a – b`
Bitwise shift operators	

<<	skiftar ett tal ett givet antal bitar åt vänster.
	`a << b`
	`ex. 9 << 2 ger 36 (00001001 två steg vänster 00100100)`

>>	skiftar ett tal ett givet antal bitar åt höger och behåller förtecknet.
	`a >> b`
	`ex. 9 >> 2 ger 2`
	`–9 >> 2 ger –3`

>>>	skiftar ett tal ett givet antal bitar år höger.
	`a >>> b`
	`ex. 19 >>> 2 ger 4`

Relational operators	
<	returnerar **true** om den vänstra operanden är mindre än den högra. *`variable1 < variable2`*
>	retrunerar **true** om den vänstra operanden är större än den högra. *`variable1 > variable2`*
<=	returnerar **true** om den vänstra operanden är mindre än eller lika med den högra operanden. *`variable1 <= variable2`*
>=	returnerar **true** om den vänstra operanden är större än eller lika med den högra operanden. *`variable1 >= variable2`*
Equality operators	
==	returnerar **true** om operanderna är lika. (Om operanderna är av olika typ försöker ECMAScript konvertera dem till lämplig typ för jämförelse.) *`variable1 == variable2`*
!=	returnerar **true** om operanderna inte är lika. (Om operanderna är av olika typ försöker ECMAScript konvertera dem till lämplig typ för jämförelse.) *`variable1 != variable2`*
Binary bitwise operators	

&	returnerar en “etta” för alla bit-positioner där motsvarande bitar i operanderna är båda “ettor”.
	`a & b`
	`ex. 14 & 9 ger 8 (1110 & 1001 = 1000)`

^	returnerar en “etta” för alla bit-positioner där motsvarande bitar i operanderna är någon av dem, men inte båda, “ettor”.
	`a ^ b`
	`ex. 14 ^ 9 ger 7 (1110 ^ 1001 = 0111)`

|	returnerar en "etta" för alla bit-postitioner där motsvarande bitar i operanderna är någon av dem, eller båda, "ettor". `a	b` `ex. 14 | 9 ger 15 (1110 | 1001 = 1111)`
Binary logical operators	

&&	returnerar *expression1* om det kan konverteras till **false**, annars returneras *expression2*. Uttrycket är motsvarigheten till **AND**.
	
	expression1 **&&** *expression2*
	`ex.`
	`true && true ger true`
	`true && false ger false`
	`false && true ger false`
	`false && false ger false`

||	returnerar *expression1* om det kan konverteras till **true**, annars returneras *expression2*. Uttrycket är motsvarigheten till **OR**.
	
	expression1 `||` *expression2*
	`ex.`
	`true || true ger true`
	`true || false ger true`
	`false || true ger true`
	`false || false ger false`

Conditional operator	
?:	returnerar *expression1* om villkoret är sant, annars returneras *expression2*. Operatorn fungerar som en enklare form av if-sats. *condition* **?** *expression1*: *expression2*
Assignment operators	
=	`x` **=** `y` tilldelar `x` värdet av `y`.
***=**	`x` ***=** `y` är kort för `x = x * y`
/=	`x` **/=** `y` är kort för `x = x / y`
%=	`x` **%=** `y` är kort för `x = x % y`
+=	`x` **+=** `y` är kort för `x = x + y`
-=	`x` **-=** `y` är kort för `x = x - y`
<<=	`x` **<<=** `y` är kort för `x = x << y`
>>=	`x` **>>=** `y` är kort för `x = x >> y`
>>>=	`x` **>>>=** `y` är kort för `x = x >>> y`
&=	`x` **&=** `y` är kort för `x = x & y`
^=	`x` **^=** `y` är kort för `x = x ^ y`
|=	`x` **|=** `y` är kort för `x = x | y`

Comma operator	
`,`	utvärderar både *`expression1`* och *`expression2`* och returnerar värdet av *`expression2`*. *`expression1, expression2`*

Egenskaper och metoder i ECMAScripts inbyggda objekt

Det globala objektet

Egenskaper

NaN	är ett värde som representerar "Not-A-Number", dvs ett icke-numeriskt värde.
Infinity	är ett värde som är större än något annat numeriskt värde inkluderat sig själv.

Metoder

eval (*x*)	utvärderar ECMAScript-kod och exekverar den. Funktionen tillåter dynamisk exekvering av källkoden. *x* = ECMAScript-kod i form av ett objekt av typen **String.**
parseInt (*string*, [*radix*])	returnerar ett heltal (integer) konverterat från en sträng. Om strängen inte kan konverteras till ett heltal returneras NaN (Not a Number). *string* = Den sträng som skall konverteras till ett heltal. *radix* = Ett tal mellan 2 och 36 som indikerar basen på det tal som skall konverteras. Om radix ej anges, antas strängar som börjar med "0x" vara hexdecimala och strängar som börjar med "0" oktala. Alla andra strängar antas vara decimala.
parseFloat (*string*)	returnerar ett flyttal (float) konverterat från en sträng. Om strängen inte kan konverteras till ett heltal returneras NaN. *string* = Den sträng som skall konverteras till ett flyttal.
escape (*string*)	returnerar ett nytt objekt av typen String i Unicode-format. Alla mellanslag, punkter, accenterade bokstäver och andra icke-ASCII-tecken ersätts med kodning av typ %xx, där xx är det hexdecimala tal som representerar tecknet. Syftet med funktionen är att returnera en sträng som kan läsas av alla typer av datorer. *string* = Den sträng som skall kodas.

unescape (*string*)	avkodar en sträng som kodats enligt escape-metoden ovan och returnerar ett objekt av typen String. *string* = Den sträng som skall avkodas.
isNaN (*number*)	returnerar ett värde av typen boolean, **true** om *number* är **NaN**, annars **false**.
isFinite (*number*)	returnerar ett värde av typen boolean, **true** om *number* är ett värde annat än **NaN**, **negative infinity** eller **positive infinity**, annars **false**.

Function

Egenskaper

arguments	returnerar en array som motsvarar argumenten i en aktuell funktion.
arguments.callee	returnerar koden för den funktion som exekveras.
arguments.length	returnerar antalet argument som skickats med till aktuell funktion.
length	specificerar antal argument som förväntas av funktionen.

Array

Egenskaper

length	återger antal element i en array.

Metoder

join ([*separator*])	returnerar en sträng bestående av alla element i arrayen separerade med ett givet tecken. Om inget argument lämnas, används kommatecken.
reverse ()	kastar om ordningen i en array. Det sista elementet blir det första och det första elementet det sista.
sort ([*compareFunction*])	sorterar elementen i en array. *compareFunction* = specificerar en funktion som definierar sorterings-ordningen.

String

Egenskaper

length	återger strängens längd.

Metoder

charAt (*index*)	returnerar en specificerat tecken från en sträng.
charCodeAt (*index*)	returnerar Unicode-värdet för ett givet tecken.
indexOf (searchString [, fromIndex])	returnerar index-positionen för den första sökträffen i strängen. Vid negativt sökresultat returneras –1.

fromCharCode(char0, ..., charN)	returnerar en sträng som skapats av Unicode-värden.
lastIndexOf (searchString [, fromIndex])	returnerar index-postitionen för den sista sökträffen i strängen. Vid negativt sökresultat returneras –1.
split (*separator*)	delar ett strängobjekt och returnerar en array av delsträngar.
substring (*start*[, *end*])	returnerar en del av ett strängobjekt.
toLowerCase ()	omvandlar alla tecken till gemener och returnerar strängen.
toUpperCase ()	omvandlar alla tecken till versaler och returnerar strängen.

Number

Egenskaper

MAX_VALUE	representerar det maximala numeriska värdet i ECMAScript (ca 1,79E+308). Värden över maxvärdet representeras av **infinity**.
MIN_VALUE	representerar det minsta positiva numeriska värdet i ECMAScript, dvs värdet närmast 0 (ca 5e-324).
NaN	representerar värdet Not-A-Number, dvs inte ett nummer.
NEGATIVE_INFINITY	representerar det numeriska värdet **negative infinity**, dvs negativ oändlighet.
POSITIVE_INFINITY	representerar det numeriska värdet **infinity**, dvs oändlighet.

Math

Egenskaper

E	Euler's konstant. Basen för naturliga logaritmer (ca 2,718).
LN10	Naturliga logaritmen av 10 (ca 2,302).
LN2	Naturliga logaritmen av 2 (ca 0,693).
LOG10E	10-logaritmen av E (ca 0,434).
LOG2E	2-logaritmen av E (ca 1,442).
PI	pi = Cirkelns omkrets dividerat med dess diameter (ca 3,14159).
SQRT1_2	Kvadratroten ur 0,5 (ca 0,707).
SQRT2	Kvadratroten ur 2 (ca 1.414).

Metoder

abs (*x*)	returnerar absolutvärdet av x.
acos (*x*)	returnerar arcus cosinus av x.
asin (*x*)	returnerar arcus sinus av x.
atan (*x*)	returnerar arcus tangens av x.
atan2 (*y*, *x*)	returnerar arcus tangens av kvoten y/x.
ceil (*x*)	returnerar det lägsta heltalet som är större än eller lika med x.
cos (*x*)	returnerar cosinus av x.
exp (*x*)	returnerar E (Euler's konstant) upphöjt med x.

floor (*x*)	returnerar den största heltalet som är mindre än eller lika med x.
log (*x*)	returnerar den naturliga logaritmen av x.
max (*x, y*)	returnerar den största av två tal x och y.
min (*x, y*)	returnerar den minsta av två tal x och y.
pow (*x, y*)	returnerar resultatet av x upphöjt till y.
random ()	returnerar ett slumpmässigt tal mellan 0 och 1.
round (*x*)	returnerar värdet av x avrundat till närmaste heltal.
sin (*x*)	returnerar sinus av x.
sqrt (*x*)	returnerar kvadratroten ur x.
tan	returnerar tangens av x.

Date

Metoder

getTime ()	returnerar tiden i objektet enligt lokal tid.
getYear[6] ()	returnerar året (två siffror) enligt lokal tid.
getFullYear ()	returnerar året enligt lokal tid.
getUTCFullYear ()	returnerar året enligt universal tid.
getMonth ()	returnerar månaden enligt lokal tid.
getUTCMonth ()	returnerar minuter enligt universal tid.
getDate ()	returnerar dagen i månaden enligt lokal tid.
getUTCDate ()	returnerar dagen i månaden enligt universal tid.
getDay ()	returnerar veckodagen enligt lokal tid.
getUTCDay ()	returnerar veckodagen enligt universal tid.
getHours ()	returnerar timmarna enligt lokal tid.
getUTCHours ()	returnerar timmarna enligt universal tid.
getMinutes ()	returnerar minuterna enligt lokal tid.
getUTCMinutes ()	returnerar minuterna enligt universal tid.
getSeconds ()	returnerar sekunderna enligt lokal tid.
getUTCSeconds ()	returnerar sekunderna enligt universal tid.
getMilliseconds ()	returnerar millisekunderna enligt lokal tid.
getUTCMilliseconds ()	returnerar millisekunderna enligt universal tid.
getTimezoneOffset ()	returnerar tidsskillnaden i minuter för den lokala tidszonen.
setTime (*time*)	sätter tidsvärdet i objektet enligt lokal tid.
setMilliseconds (*ms*)	sätter millisekunderna enligt lokal tid.
setUTCMilliseconds (*ms*)	sätter millisekunderna enligt universal tid.
setSeconds (*sec* [, *ms*])	sätter sekunderna enligt lokal tid.
setUTCSeconds (*sec* [, *ms*])	sätter sekunderna enligt universal tid.
setMinutes (*min* [, *sec* [, *ms*]])	sätter minuterna enligt lokal tid.
setUTCMinutes (*min* [, *sec* [, *ms*]])	sätter minuterna enligt universal tid.
setHours (*hour* [, *min* [, *sec* [, *ms*]]])	sätter timmarna enligt lokal tid.

[6] getYear och setYear rekommenderas inte av ECMA och kräver ej implementering.

setUTCHours (*hour* [, *min* [, *sec* [, *ms*]]])	sätter timmarna enligt universal tid.
setDate (*date*)	sätter dagen i månaden enligt lokal tid.
setUTCDate (*date*)	sätter dagen i månaden enligt universal tid.
setMonth (*mon* [, *date*])	sätter månaden enligt lokal tid.
setUTCMonth (*mon* [, *date*])	sätter månaden enligt universal tid.
setFullYear (*year* [, *mon* [, *date*]])	sätter året enligt lokal tid.
setUTCFullYear (*year* [, *mon* [, *date*]])	sätter året enligt universal tid.
setYear (*year*)	sätter året (två siffror) enligt lokal tid.
toLocaleString ()	konverterar datumobjektet till en sträng enligt lokalt tidsformat.
toUTCString ()	konverterar datumobjektet till en sträng enligt det universala tidsformatet.
toGMTString ()	konverterar datumobjektet till en sträng enligt GMT-formatet på Internet.
UTC (*year*, *month*, *day* [, *hour*, *min*, *sec*, *ms*])	returnerar, i ett datumobjekt, millisekunder sedan den 1 januari 1970 00:00:00 enligt universal tid.
parse (*dateString*)	returnerar antal millisekunder sedan den 1 januari 1970 00:00:00 enligt lokal tid baserat på en sträng (t ex ”Dec 24, 1999”)

Ytterligare funktionalitet i JavaScripts inbyggda objekt

Array

Nya metoder

concat (*array2, array3,...,arrayN*))	returnerar en ny array bestående av en kombination av två eller fler arrayer.

slice (*start*, [*end*])	returnerar en ny array bestående av en given del av en array.
	`start =`
	`end =`

pop ()	tar bort den sista posten i en array och returnerar posten.
push (*element1,element2,...,elementN*)	lägger till ett eller flera element i slutet på en array och returnerar den nya längden på arrayen.
shift ()	tar bort den första posten i en array och returnerar posten.
splice (*index, howMany,* [*element1*][*,..., elementN*]))	ändrar innehållet i en array genom att valfritt lägga till nya poster samtidigt som gamla poster tas bort.
unshift (*element1,...,elementN*)	lägger till en eller flera poster i början av en array och returnerar arrayens nya längd.

String

Nya metoder

anchor (*anchorstring*)	placerar HTML-taggen <A> runt ett strängobjekt. *anchorstring* = den text som placeras i A-tag-attributet **NAME.**
big ()	placerar HTML-taggen <BIG> runt ett strängobjekt.
blink ()	placerar HTML-taggen <BLINK> runt ett strängobjekt.
bold ()	placerar HTML-taggen <B> runt ett strängobjekt.
concat (*string2, string3* [*, ..., stringN*])	returnerar ett objekt av typen **String** innehållande en sammanslagning av två eller fler givna strängar.
fixed ()	placerar HTML-taggen <TT> runt ett strängobjekt.
fontcolor (*colorval*)	placerar HTML-taggen <FONT> runt ett strängobjekt. *colorval* = den färg som placeras i FONT-tag-attributet **COLOR**.

fontsize (*intSize*)	placerar HTML-taggen <FONT> runt ett strängobjekt. *intSize* = den storlek som placeras i FONT-tag-attributet **SIZE**.
italics ()	placerar HTML-taggen <I> runt ett strängobjekt.
link (*linkstring*)	placerar HTML-taggen <A> runt ett strängobjekt. *linkstring* = den text som placeras i A-tag-attributet **HREF.**
match (*rgExp*)	returnerar en array innehållande resultatet av en sökning på en sträng med ett objekt av typen **Regular Expression** (se nedan).
replace (*rgExp*, *replaceText*)	returnerar en kopia av en given sträng med vissa delar av texten utbytta med hjälp av ett objekt av typen **Regular Expression**.
search (*rgExp*)	returnerar positionen för den första delsträngen som matchar en sökning med hjälp av **Regular Expression**.
slice (*start*[, *end*])	returnerar en del av en sträng.
small ()	placerar HTML-taggen <SMALL> runt ett strängobjekt.
split ([*separator*][,*limit*])	delar ett strängobjekt och returnerar en array av delsträngar. Metoden finns i ECMA-262, men JavaScript tillför argumentet **limit**. *limit* = sätter en gräns för antal träffar.
strike ()	placerar HTML-taggen <STRIKE> runt ett strängobjekt.
sub ()	placerar HTML-taggen <SUB> runt ett strängobjekt.
substr (*start*[,*length*]))	returnerar en delsträng från ett givet tecken med en given längd.
sup ()	placerar HTML-taggen <SUP> runt ett strängobjekt.

Function

Nya egenskaper

arguments.caller (Netscapes referensmaterial rekommenderar att egenskapen inte används i JavaScript 1.3)	specificerar en referens till den funktion som anropade den aktuella funktionen. Är endast definierad medan den aktuella funktionen exekveras.
arity	specificerar antal argument som förväntas av funktionen.

Nya metoder

apply (*thisArg*[, *argArray*])	tillåter ett objekt att ärva ett annat objekts metod.
call (*thisArg*[, *arg1*[, *arg2*[, ...]]])	tillåter exekvering av ett annat objekts metod.

Object

Nya metoder

watch (*property*, *handler*)	övervakar och väntar på att en egenskap skall tilldelas ett värde och exekverar då en funktion.
unwatch (*property*)	tar bort en ”watchpoint” som har satts av metoden watch.
toSource ()	returnerar en sträng som representerar källkoden för ett objekt. Metoden används främst internt av JavaScript.

RegExp

Egenskaper

global	är **true** om ”**g**”-flaggan användes, annars **false**.
ignoreCase	är **true** om ”**i**”-flaggan användes, annars **false**.
lastIndex	specificerar det index på vilken nästa sökning skall starta.
source	returnerar texten i ett **regular expression**.

Metoder

test ([*str*])	returnerar **true** om sökningen innehåller en lyckad matchning, annars **false**.
compile (*pattern* [, *flags*])	kompilerar ett **regular expression** till ett internt format i JavaScript.
exec ([*str*])	exekverar en sökning i en given sträng. Returnerar en resultatarray.

Egenskaper tillhörande det fördefinierade objektet **RegExp**

$1, ..., $9	returnerar någon av de nio senaste memorerade delsträngarna som matchat ett **regular expression**.
input	returnerar den sträng vilken ett **regular expression** matchades mot.
lastMatch	returnerar den senaste matchningen.
lastParen	returnerar den senaste matchningen från en parentes (en delsträng).
leftContext	returnerar den delsträng som föregår den senaste matchningen.
multiline	är **true** om sökningen innehåller multipla rader, **false** om sökninngen avbryts vid radbrytningar.

`rightContext`	returnerar den delsträng som följer den senaste matchningen.

Ytterligare funktionalitet i JScripts inbyggda objekt

Array

Nya metoder

concat (*array2*)	returnerar en ny array bestående av en kombination av två arrayer.
slice (*start*, [*end*])	returnerar en del av en array.

String

Nya metoder

anchor (*anchorstring*)	placerar HTML-taggen <A> runt ett strängobjekt. *anchorstring* = den text som placeras i A-tag-attributet **NAME.**
big ()	placerar HTML-taggen <BIG> runt ett strängobjekt.
blink ()	placerar HTML-taggen <BLINK> runt ett strängobjekt.
bold ()	placerar HTML-taggen <B> runt ett strängobjekt.
concat (*string2*)	returnerar ett objekt av typen **String** innehållande en sammanslagning av två givna strängar.
fixed ()	placerar HTML-taggen <TT> runt ett strängobjekt.
fontcolor (*colorval*)	placerar HTML-taggen <FONT> runt ett strängobjekt. *colorval* = den färg som placeras i FONT-tag-attributet **COLOR**.
fontsize (*intSize*)	placerar HTML-taggen <FONT> runt ett strängobjekt. *intSize* = den storlek som placeras i FONT-tag-attributet **SIZE**.
italics ()	placerar HTML-taggen <I> runt ett strängobjekt.
link (*linkstring*)	placerar HTML-taggen <A> runt ett strängobjekt. *linkstring* = den text som placeras i A-tag-attributet **HREF.**
match (*rgExp*)	returnerar en array innehållande resultatet av en sökning på en sträng med ett objekt av typen **Regular Expression** (se nedan).

replace (*rgExp, replaceText*)	returnerar en kopia av en given sträng med vissa delar av texten utbytta med hjälp av ett objekt av typen **Regular Expression**.
search (*rgExp*)	returnerar positionen för den första delsträngen som matchar en sökning med hjälp av **Regular Expression**.
slice (*start*[, *end*])	returnerar en del av en sträng.
small ()	placerar HTML-taggen <SMALL> runt ett strängobjekt.
strike ()	placerar HTML-taggen <STRIKE> runt ett strängobjekt.
sub ()	placerar HTML-taggen <SUB> runt ett strängobjekt.
substr (*start*[,*length*]))	returnerar en delsträng från ett givet tecken med en given längd.
sup ()	placerar HTML-taggen <SUP> runt ett strängobjekt.

Function

Nya egenskaper

caller	returnerar en referens till den funktion som anropade den aktuella funktionen. Är endast definierad medan den aktuella funktionen exekveras.

Date

Nya Metoder

getVarDate ()	returnerar "VT-DATE"-värdet i ett Date-objekt. Används vid interaktion med ActiveX-objekt, och andra objekt som accepterar och returnerar värden i "VT-DATE"-format

Globala funktioner

JScript tillhandahåller även fyra ytterligare funktioner, med åtkomst på global nivå. De syftar till att ge information om det skriptspråk som används.

ScriptEngine ()	returnerar en sträng som visar det skriptspråk som används ("JScript", "VBA", eller "VBScript)
ScriptEngineBuildVersion ()	returnerar tillverkningsversionsnumret på den skriptmotor som används.
ScriptEngineMajorVersion ()	returnerar huvudversionsnumret på den skriptmotor som används
ScriptEngineMinorVersion ()	returnerar delversionsnumret på den skriptmotor som används.

RegExp

Egenskaper

lastIndex	specificerar det index på vilken nästa sökning skall starta.
source	returnerar texten i ett regular expression utan ”/” ”**g**” och ”**i**”.

Metoder

test ([*str*])	returnerar **true** om sökningen innehåller en lyckad matchning, annars **false**.
compile (*pattern* [, *flags*])	kompilerar ett **regular expression** till ett internt format i JScript.
exec ([*str*])	

VBArray

Egenskaper tillhörande det fördefinierade objektet **RegExp**

$1, ..., $9	returnerar någon av de nio senaste memorerade delsträngarna som matchat ett **Regular Expression**.
input	returnerar den sträng vilken ett **Regular Expression** matchades mot.
lastIndex	returnerar indexet för första tecknet i senaste lyckade matchningen.

Metoder

dimensions ()	returnerar antalet dimensioner i en VBArray.
getItem (dimension1[, dimension2, ...], dimensionN)	returnerar en given post i arrayen.
lbound (dimension)	returnerar det lägsta indexvärdet som används i en given dimension av en VBArray.
toArray ()	returnerar en JScript-Array konverterad från en VBArray.
ubound (dimension)	returnerar det högsta indexvärdet som används i en given dimension av en VBArray.

Enumerator

Metoder

atEnd ()	returnerar ett värde av typen Boolean som anger om enumeratorn är vid slutet av samlingen.
item ()	returnerar den aktuella posten i samlingen.
moveFirst ()	återställer den aktuella posten till den första posten i samlingen.
moveNext ()	flyttar den aktuella posten till nästa post i samlingen.