더글라스 크락포드가 정제한
자바스크립트 언어의 정수!

더글라스 크락포드의

자바스크립트 핵심 가이드

더글라스 크락포드 저 | 김명신 역

JavaScript The Good Parts

한빛미디어 Hanbit Media, Inc. O'REILLY®

더글라스 크락포드의

자바스크립트 핵심 가이드

JavaScript: The Good Parts

| 표지 설명 |

이 책의 표지에 있는 곤충은 플레인 타이거 나비입니다(Danaus chrysippus). 아시아 외부 지역에서는 아프리카 황제 나비로도 잘 알려져 있습니다. 이 나비는 밝은 오랜지 색 날개 위에 여섯 개의 검은 점과 줄무늬가 특징인 중간 크기의 나비입니다.

플레인 타이거 나비의 아름다움은 매력적인 요소이지만 다른 한편으로는 죽음을 부르는 요소입니다. 유충 단계에서 이 나비는 새들에게 독이 되는 성분을 섭취합니다. 플레인 타이거의 화려한 색에 사로잡혀 이를 잡아먹은 새는 반복해서 토하게 되고 심지어는 죽을 수도 있습니다. 만약 이 나비를 잡아먹은 새가 살아남게 된다면 이 새는 다른 새들에게 우아하고 유유자적하며 지면 가까이 나는 이 곤충을 피하도록 알릴 것입니다.

더글라스 크락포드의 **자바스크립트 핵심 가이드**

지은이 | 더글라스 크락포드
옮긴이 | 김명신
펴낸이 | 김태헌
펴낸곳 | 한빛미디어(주)
주 소 | 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전 화 | IT전문서팀 02)325-0984, 영업1팀 02)336-7114
팩 스 | 02)336-7124
등 록 | 1999년 6월 24일 제10-1779호
초판 발행 | 2008년 9월 29일
2쇄 발행 | 2011년 9월 7일
정 가 | 22,000원
ISBN | 978-89-7914-598-4 93560

기 획 | 서형철

이 책에 대한 의견을 주시거나 오탈자 및 잘못된 내용의 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 연락 주십시오. 잘못된 책은 구입하신 서점에서 교환해 드립니다.

http://www.hanb.co.kr
ask@hanb.co.kr

Ajax의 등장으로 자바스크립트의 중요성은 이전보다 훨씬 강조되고 있습니다. 하지만 Ajax를 설명하는 책은 수없이 나왔음에도 자바스크립트의 본질에 대해 이야기하는 책은 거의 없었던 것이 사실입니다. 다행인 것은 최근 들어 그러한 책들이 하나 둘씩 선을 보이고 있다는 것이며 이 책도 그 중에 하나입니다.

자바스크립트계의 요다 스승으로 불리고 있는 더글라스 크락포드가 쓴 이 책은 자바스크립트에 대한 혜안을 제공합니다. 자바스크립트를 많이 접해 보지 않았거나 가볍게 여기고 있던 분이라면 '헉, 자바스크립트가 이렇게 심오했나!' 하는 생각을 할 것입니다. 이 책은 자바스크립트 제다이 기사로 인도하는 비급이라고 할 수 있습니다.

무공 비급들이 항상 그렇듯이 이 책의 내용은 쉽지 않습니다. 하지만 저자도 이 책에서 얘기하고 있는 것처럼 반복해서 읽고 온전히 이해한다면 그 노력은 결코 헛되지 않을 것입니다.

많은 제다이 기사 탄생을 기대해 봅니다.

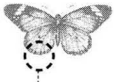
| 주의사항 |

이 책의 부록은 말 그대로 부록이 아니라 반드시 읽어야 하는 이 책의 주요 내용입니다. 꼭 부록까지 다 읽으세요.

| 감사의 글 |

제 행복의 근원인 우리 가족과 너무도 교정을 잘 해주신 한빛미디어㈜ 서형철 대리님께 감사드립니다.

2008년 9월 김명신



저저서문

만약에 여러분의 비위에 거슬린다면, 이것이 곧 저희의 소원이외다. 여러분의 비위를 거슬리려고 한 것은 아니고, 저희의 소원인 즉 서투른 솜씨를 보이자는 것이고 이것이 곧 저희들의 진정한 동기외다.

– 윌리엄 셰익스피어, 한여름 밤의 꿈

이 책은 자바스크립트 프로그래밍 언어에 관한 책입니다. 이 책의 목적은 자바스크립트를 우연한 기회에 접하였거나 이에 대해 호기심이 생겨 탐험하고자 하는 프로그래머들을 위해 쓰여졌습니다. 또한 이 책은 이전에 자바스크립트를 사용해 본적이 전혀 없으면서 이제 막 자세히 알아보려고 하는 프로그래머를 위해서도 집필되었습니다. 자바스크립트는 놀라울 정도로 강력한 언어입니다. 기존의 언어들과 조금은 다른 특성들이 있지만, 그리 어렵지 않게 익힐 수 있습니다.

또 다른 목적은 여러분이 자바스크립트식으로 생각할 수 있는 힘을 갖게 돕는 것입니다. 이 책을 통해서 자바스크립트라는 언어가 제공하는 기능들을 보여드릴 것이며, 기능들을 잘 조합해서 사용하는 방법을 찾는 길로 여러분을 안내할 것입니다. 이 책은 참고용으로 쓰여진 것이 아닙니다. 즉 언어에 대한 모든 명세와 특성을 망라하지 않으며 여러분이 후에 알 필요가 있을지도 모르는 모든 사항을 담고 있지도 않습니다. 그러한 내용들은 온라인 상에서 쉽게 찾을 수 있을 것입니다. 대신에 이 책은 정말로 중요한 사항만을 포함하고 있습니다.

이 책은 초급자를 위한 책이 아닙니다. 앞으로 언젠가 저는 JavaScript: The First Parts라는 제목으로 초급자를 위한 책을 쓰고 싶습니다. 하지만 지금 이 책은 아닙니다. 또한 Ajax나 웹 프로그래밍에 관한 책도 아닙니다. 이 책은 웹 개발자라면 반드시 숙지해야 될 자바스크립트에만 온전히 초점을 맞추고 있습니다.

이 책은 분량이 적지만 그 내용에 있어서는 어느 책보다도 깊이가 있습니다. 다양한 내용들이 잘 정제되어 있어서 조금 어려울 수도 있고, 이해가 잘 안 돼서 몇 번을 다시 읽게 될 수도 있습니다. 하지만 실망하지 않기 바랍니다. 그 노력의 결과는 분명히 달 것입니다.

| 감사의 글 |

수많은 오류를 적절히 지적한 검토자 분들에게 감사드립니다. 인생에서 자신의 실수를 적절히 지적해주는 현명한 분들과의 교류보다 더 나은 것은 거의 없다고 생각합니다. 특히 자신의 잘못이 여러 대중에게 알려지기 전에 미리 검토 받을 수 있다는 것은 더욱 좋은 일입니다. Steve Souders, Bill Scott, Julien LeComte, Stoyan Stefanov, Eric Miraglia, Elliotte Rusty Harold에게 진심으로 감사드립니다.

자바스크립트의 깊은 곳에 있는 좋은 점들을 찾도록 도와준 일렉트릭 커뮤니티 사와 스테이트 소프트웨어서 일하는 분들께 감사를 전합니다. 특히, Chip Morningstar, Randy Farmer, John La, Mark Miller, Scott Shattuck, Bill Edney에게는 더욱 감사드립니다.

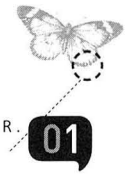
제가 몸담고 있는 야후!에도 감사드립니다. 저에게 이 책을 집필할 수 있는 시간을 주었으며 최상의 작업 공간을 제공했습니다. 아울러 과거와 현재의 Ajax Strike Force 팀 모든 분께도 감사드립니다. 또한 오라일리 미디어에도 감사드리며 특

히 모든 작업이 원활히 진행되도록 도와준 Mary Treseler, Simon St.Laurent, Sumita Mukherji에게 감사드립니다.

Lisa Drake 교수님에게 특별한 감사를 전합니다. 그리고 ECMAScript를 더 나은 언어로 만들기 위해 고군분투하고 있는 ECMA TC39 구성원 여러분에게도 감사하고 싶습니다.

마지막으로 세계에서 가장 많은 오해를 받고 있는 프로그래밍 언어의 설계자인 Brendan Eich에게 감사합니다. 만약 그가 없었다면 이 책은 나오지 못했을 것입니다.

목 차



C.H.A.P.T.E.R. 01

자바스크립트의 좋은 점들

... 나는 마력 같은 건 없는 사람이요. 본래 타고난 좋은 점(good parts)이 여성들에게 매력이 있는 거겠지요.

– 윌리엄 셰익스피어, 윈저의 명랑한 아낙네들

초보 프로그래머 시절에 필자는 새로운 프로그래밍 언어를 사용할 때 항상 언어의 모든 기능을 다 익혀서 사용하려고 노력했습니다. (여러분도 경험해서 알겠지만) 그렇게 하는 것이 실력을 과시하는 것이라고 생각했고 또한 그렇게 익혀서 사용한 기능들은 모두 다 제대로 동작할 것이라고 믿었습니다.

하지만 결국 알게 된 것은 그 기능들 중에 일부는 득보다 실이 많다는 것이었습니다. 일부 기능은 제대로 정의되지 않은 경우도 있었고, 다른 쪽으로 이식을 할 때 문제를 일으키는 기능들도 있었습니다. 그리고 어떤 요소들은 수정을 어렵게 하거나 가독성이 떨어지게 했고 심지어 어떤 것들은 오류가 발생할 확률이 높고 복잡한 방법으로 프로그래밍해야지만 사용할 수 있었습니다. 그뿐만 아니라 어떤 기능들은 언어의 구조에 문제가 있는 경우도 있었고, 언어를 고안한 사람이 실수를 한 경우도 있었습니다.

대부분의 프로그래밍 언어에는 좋은 점(good parts)과 나쁜 점(bad parts)이 있습니다. 오랜 프로그래밍 경험으로 언어의 좋은 점만을 사용하고 나쁜 점은 사용을 자제함으로써 좀더 좋은 프로그래머가 될 수 있다는 것을 알게 되었습니다. 결국, 요점은 어떻게 하면 나쁜 점을 피해서 좋은 결과를 얻느냐는 것입니다.

표준을 제정하는 위원회가 언어의 나쁜 점들을 제거하기는 어렵습니다. 왜냐하면 위원회가 나쁜 점을 제거하게 되는 경우 기존에 그 나쁜 점을 사용하여 개발된 프로그램들에 영향을 미치기 때문입니다. 그러므로 위원회는 보통 기존의 불완전함 위에 더 많은 기능을 추가하는 것 외에는 힘이 없습니다. 설상가상으로 새로 추가한 기능들이 기존 기능들과 조화를 이루지 못하는 경우도 있기 때문에, 언어의 나쁜 점이 가중되는 결과를 낳기도 합니다.

하지만 다행인 것은 언어의 모든 기능과 명세가 어떻든 간에 그 기능들 중에서 자신만의 부분집합을 지정할 수 있다는 것입니다. 우리는 전적으로 언어의 좋은 점만을 사용함으로써 좀더 나은 프로그램을 작성할 수 있습니다.

자바스크립트는 연구실에서 좀더 좋게 정련하는 과정을 거치지 못한 체 세상으로 나왔지만 출현하자마자 단기간에 브라우저 전체에 채택됐습니다. 그러나 기억할 점은 자바스크립트의 유명세는 프로그래밍 언어로서의 품질과 별개의 문제였다는 것입니다.

다행히도, 자바스크립트는 좋은 점(good parts)이 상당히 많습니다. 좋은 의도와 실수들 더미에 묻혀있는 아름답고 우아하며 매우 표현적인 특성이 있습니다. 자바스크립트의 진정한 속성들은 '자바스크립트는 별 볼일 없고 기능 없는 장난감에 불과하다'는 지배적인 의견 때문에 수년 동안 감춰져 있었습니다. 이 책을 집필하게 된 의도가 이 부분에 있습니다. 바로 동적 언어로서 두드러진 특성을 지닌 자바스크립트의 장점들을 부각하기 위해서입니다. 이 책에서는 자바스크립트의 진정한 본성을 드러내지 못하는 우아하지 않은 기능들을 모두 제거해 버렸습니다. 이 책에서 제시하는 정제된 언어의 부분집합은 보다 신뢰할 수 있고 읽기 편하며 유지가 용이한 좋은 언어적 특성을 나타낼 것입니다.

이 책에는 자바스크립트의 모든 것이 담겨 있지 않습니다. 그래서 간간히 피해야 할 나쁜 점들을 언급하기는 하지만 전적으로 자바스크립트의 좋은 점에만 초점을 맞출 것입니다. 여기서 기술하는 부분집합은 크던 작던 규모에 상관 없이 신뢰할 수 있고 가독성이 좋은 프로그램을 작성하는데 사용할 수 있습니다. 또한 좋은 점(good parts)에만 초점을 맞춤으로써 언어를 익히는 시간도 단축할 수 있고, 프로그램의 견고함도 높일 수 있으며 종이도 아낄 수 있습니다.

아마 좋은 점(good parts)만을 배움으로써 얻을 수 있는 최대의 이점은 나쁜 점을 잊어야 하는 수고를 덜 수 있다는 것입니다. 나쁜 패턴을 잊어버리는 것은 쉽지 않습니다. 잊어버리려고 시도하면 대부분은 심한 저항에 부딪히게 됩니다. 때때로 프로그래밍 언어를 보다 쉽게 가르치기 위해서 부분집합만을 사용하는 경우가 있습니다. 하지만 이 책에서는 그런 초보용 부분집합이 아니라 전문가를 위한 자바스크립트의 부분집합을 다룹니다.

01 | 왜 자바스크립트인가?

자바스크립트는 웹 브라우저의 언어이기 때문에 세계에서 가장 유명하고 중요한 언어 중 하나입니다. 하지만 그와 동시에 가장 무시 당하고 있는 언어 중에 하나이기도 합니다. 브라우저의 API라고 할 수 있는 DOM(Document Object Model)은 아주 형편없는데 거기에 편승하여 자바스크립트도 부당한 비난을 받고 있습니다. DOM을 어떠한 언어로 다루든지 용이하지 않습니다. DOM은 서툴게 정의됐으며 그에 따라 그 구현도 제 각각입니다. 이 책에서는 DOM에 관해 아주 적은 부분만을 언급합니다. 아마 DOM에 관한 좋은 점만을 모아 놓은 책을 집필하는 것은 매우 어려운 도전일 것입니다.

자바스크립트는 여타 다른 언어들과 다른 점이 많기 때문에 매우 저 평가돼 있습니다. 만약 여러분이 다른 언어들에 익숙한 상태인데 오로지 자바스크립트만 지원하는 환경에서 프로그래밍을 해야 하는 상황에 처한다면, 당연히 자바스크립트만을 사용해야 할 것이며 이로 인해 무척 짜증이 날 것입니다. 이러한 상황에 처하는

대부분의 사람은 일단 자바스크립트를 알아보려는 노력조차 하지 않습니다. 그러다가 자신들이 대신 사용했으면 하는 언어와 매우 다르다는 점에 놀라고 차별하기 시작합니다.

자바스크립트의 놀라운 점은 언어 자체에 대해 많이 모르거나 심지어는 프로그래밍에 대해 잘 모르더라도 원하는 작업을 할 수 있다는 것입니다. 이를 볼 때 자바스크립트는 놀라운 표현력을 가진 언어입니다. 하지만 여러분이 무엇을 하고 있는지 알고 있을 때 자바스크립트는 더 큰 힘을 발휘합니다. 프로그래밍은 어려운 비즈니스기 때문에 결코 무지한 가운데 진행해서는 안됩니다.

02 | 자바스크립트 분석

자바스크립트는 몇몇의 매우 좋은 아이디어와 몇몇의 나쁜 (그것도 아주) 아이디어에 기초하여 만들어졌습니다.

매우 좋은 아이디어에는 함수, 느슨한 타입 체크, 동적 객체, 표현적인 객체 리터럴 표기법 등이 있습니다. 그리고 대표할 만한 나쁜 아이디어는 프로그래밍 모델이 전역변수에 기초하고 있다는 것입니다.

자바스크립트의 함수는 어휘적 유효범위를 가진 일급 객체(first-class object)[1] 입니다. 또한 주류가 된 첫 번째 람다(lambda)[2] 언어며, 좀더 깊이 들어가면, 이름처럼 자바에 가깝기보다는 Lisp 언어 그리고 Scheme 언어와 더 많은 공통점이 있습니다. 자바스크립트는 C의 옷을 입은 Lisp이라고 할 수 있습니다. 놀라울 정도로 강력한 언어적 특징은 바로 이러한 특성에서 기인한 것입니다.

1. 역자 주/ 언어상에 제약이 없는 객체를 일급 객체라고 합니다. 즉 변수에 대입되거나 인수로 넘길 수도 있고, 반환값으로 사용하거나 연산 등에 사용하는데 전혀 제약이 없는 객체입니다. 자바스크립트에서는 함수가 일급 객체이므로 이 모든 것이 다 가능합니다. 영문이지만 보다 자세한 내용은 http://en.wikipedia.org/wiki/First-class_object를 참고하세요.
2. 역자 주/ 람다는 Alonzo Church와 Stephen Cole Kleene의 람다 대수(Lambda calculus, http://en.wikipedia.org/wiki/Lambda_calculus 참조)에서 유래한 것으로 익명 함수나 클로저 등을 정의하기 위한 표현식을 의미합니다. 더 자세한 사항은 Lisp이나 파이썬에서 람다 부분을 찾아보기 바랍니다. 영문이지만 자세한 내용은 http://en.wikipedia.org/wiki/Lambda를 참고하세요.

오늘날 프로그래밍 언어 대부분은 강력한 데이터 타입 체크를 요구합니다. 이러한 경향은 강력한 타입 체크를 해야 컴파일러가 컴파일 시간에 가능한 많은 오류를 찾을 수 있다는 이론에 근거한 것입니다. 오류를 가능한 빨리 찾아서 수정을 하면 추후에 오류 발생으로 생기는 비용을 더 많이 줄일 수 있다는 생각입니다. 자바스크립트는 느슨한 타입 체크 언어입니다. 그래서 자바스크립트 컴파일러는 타입 오류를 찾을 수 없습니다. 이러한 점은 강력한 타입 체크를 하는 언어를 사용하다가 자바스크립트로 넘어온 사람들에게는 놀라운 일일 수 있습니다. 하지만 강력한 타입 체크가 중요한 오류들을 효율적으로 제거하지 못한다는 것이 판명됐습니다. 그리고 경험에 의하면 강력한 타입 체크가 오류를 알려주기는 하지만 실제로 우려하는 오류들은 알려주지 못합니다. 반면에 느슨한 타입 체크에서 발견한 것은 오류 찾기의 어려움이 아니라 자유로움입니다. 느슨한 타입 체크를 하는 언어를 사용하면 복잡한 클래스 계층을 구성할 필요도 없으며, 원하는 대로 동작하도록 타입 캐스팅과 씨름할 필요도 없습니다.

자바스크립트는 매우 강력한 객체 리터럴 표기법이 있습니다. 이 표기법을 사용하면 단순히 필요한 요소들을 열거하는 방법으로 객체를 만들 수 있습니다. 이러한 표기법은 유명한 데이터 교환 형식인 JSON에도 영감을 주었습니다(JSON은 부록 E에서 자세히 알아봅니다).

자바스크립트에서 논란의 대상이 되는 기능은 프로토타입에 의한 상속입니다. 자바스크립트는 클래스가 필요 없는 객체 시스템이 있어서 특정 객체에 있는 속성들을 다른 객체에 직접 상속할 수 있습니다. 이러한 특성은 매우 강력하지만 클래스 기반의 언어에 익숙한 프로그래머에게는 친숙하지 않은 점입니다. 만약, 여러분이 클래스 기반의 설계 패턴을 그대로 자바스크립트에 사용하려고 한다면 아마 좌절감을 느낄 것입니다. 하지만 자바스크립트가 제공하는 프로토타입의 특성을 배운다면 이러한 노력은 분명 보상 받을 것입니다.

자바스크립트는 몇몇 핵심 개념들 때문에 더 많이 비난을 받고 있습니다. 좀 다르긴 해도 핵심 개념의 대부분은 좋은 선택이었습니다. 하지만 특별히 잘못 선택된 점이 하나 있는데 그것은 전역변수에 근간을 두고 있다는 것입니다. 모든 컴파일

단위에 있는 최상위 레벨의 변수들은 모두 전역객체(global object)라 불리는 공용 이름 공간(namespace)에 위치하게 됩니다. 전역변수는 나쁘기 때문에 이러한 개념은 상당히 좋지 않다고 볼 수 있습니다. 그런데 이것이 자바스크립트의 근간을 이루는 것 중에 하나입니다. 그나마 다행인 것은 자바스크립트에는 이러한 문제를 완화시킬 수 있는 방법이 있다는 것입니다.

때때로 어쩔 수 없이 사용하는 자바스크립트의 나쁜 점들이 있습니다. 이렇게 피할 수 없는 필요악 같은 부분들은 이 책에서 나올 때마다 바로 언급할 것입니다. 또한 이런 부분들을 부록 A에 정리했습니다. 피할 수 없는 나쁜 부분들이 있기는 하지만, 우리는 이 책으로 인해 부록 B에서 알려주고 있는 꼭 피해야 할 자바스크립트의 나쁜 점들은 대부분 피하는데 성공할 것입니다. 만약 여러분이 이 책에서 다루지 않은 더 많은 나쁜 점을 그 사용법까지 자세히 알고 싶다면 여타 다른 자바스크립트 책을 만나보기 바랍니다.

자바스크립트(또는 JScript)를 정의하고 있는 표준은 「The ECMAScript Programming Language third Edtion」(http://www.ecmainternational.org/publications/files/ecma-st/ECMA-262.pdf)입니다. ECMAScript는 나쁜 부분을 포함하기 때문에 이 책에서는 언어 전체를 기술하지 않습니다. 즉 전체를 총망라하여 기술하지 않고 위험한 부분들을 피해서 기술하고 있습니다.

부록 C에서는 JSLint라는 프로그래밍 툴을 설명합니다. 이 툴은 자바스크립트 프로그램을 분석하여 포함된 나쁜 점들을 알려주는 자바스크립트 파서입니다. JSLint는 [illegible]스크립트 개발에서 부족하기 쉬운 엄격한 수준의 분석 결과를 제공합니다. 그러므로 이 툴의 분석에서 나쁜 점이 보고되지 않았다면 자바스크립트의 좋은 점만을 사용해서 프로그래밍됐다고 확신할 수 있습니다.

자바스크립트는 모순이 많은 언어입니다. 자바스크립트에는 많은 오류가 있기 때문에, '도대체 왜 자바스크립트를 사용해야 하지?'하고 생각할 수 있습니다. 이에 대한 답은 두 가지가 있습니다. 그 중 첫 번째는 선택의 여지가 없다는 것입니다. 웹은 애플리케이션 개발에 있어 중요한 플랫폼이며, 자바스크립트는 모든 브라우저에서 사용할 수 있는 유일한 언어입니다. 자바가 이 환경에서 실패한 것은 어쩌

면 불행한 일이라고도 볼 수 있습니다. 만약 자바가 실패하지 않았다면 강력한 타입 체크를 하는 클래스 기반의 언어를 원하는 이들에게는 좋은 선택이 됐을 것입니다. 하지만 자바는 실패했고 자바스크립트는 살아남아 번성했으며, 이는 자바스크립트가 나름대로 옳았다는 하나의 증거이기도 합니다.

두 번째 답은 많은 부족한 면이 있기는 하지만 자바스크립트는 실제로 꽤 괜찮다는 것입니다. 자바스크립트는 매우 경량화돼 있으며 표현적(expressive)입니다. 그리고 일단 언어 사용의 요령을 익히고 나면 함수형 프로그래밍(functional programming)이 꽤 재미있다는 것을 알게 됩니다.

하지만 자바스크립트라는 언어를 잘 사용하기 위해서는 이 언어의 한계점을 잘 알아야 합니다. 이를 위해 이 책에서는 자바스크립트를 용감하게 조각내서 좋은 점만을 제공할 것입니다. 자바스크립트의 좋은 점들(good parts)은 나쁜 점들을 보상하고도 남을 정도로 충분한 가치가 있습니다.

03 | 예제 테스트를 위한 간단한 준비

웹 브라우저와 텍스트 에디터만 있으면 자바스크립트 프로그램을 실행시킬 모든 준비가 끝난 것입니다. 먼저 다음 내용을 program.html 파일로 저장합니다.

```
<html><body><pre><script src="program.js">
</script></pre></body></html>
```


그런 다음 같은 폴더에 다음의 내용을 program.js 파일로 저장합니다.

```
document.writeln('Hello, world!');
```

이제 결과를 보기 위해 program.html 파일을 브라우저로 불러와서 결과를 확인합니다.

다음에 나오는 코드는 이 책 전체에서 새로운 메소드를 정의할 때 두루 쓰이는 method라는 메소드입니다.

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
```

일단 눈여겨보기 바랍니다. 이 메소드에 대한 자세한 내용은 4장에서 살펴봅니다.

C.H.A.P.T.E.R. 02

자바스크립트의 좋은 문법들

내가 잘 아는 것이군. 오래 전 문법책 예문에서 읽은 시구로군.
-윌리엄 셰익스피어, 타이터스 앤드로니커스

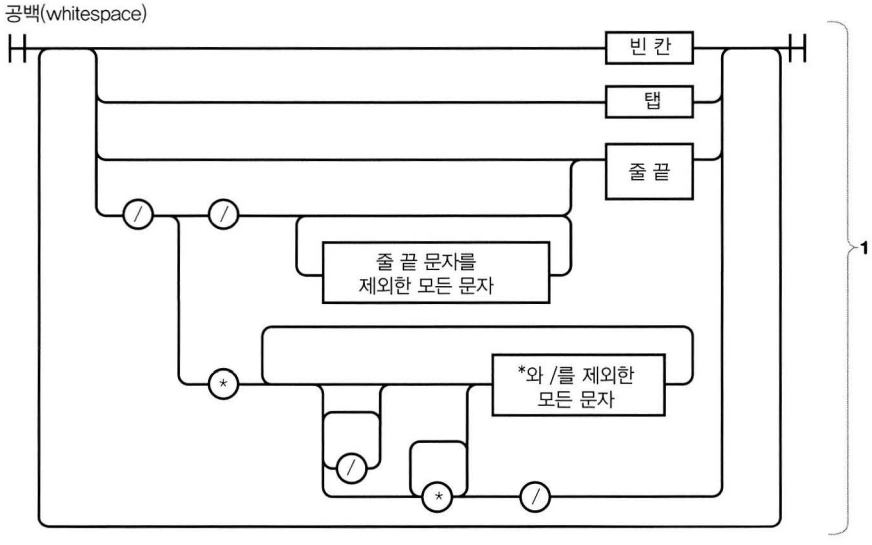
이번 장에서는 자바스크립트의 좋은 점들(good parts)에 해당하는 문법을 살펴봅니다. 이 문법들을 살펴봄으로써 언어가 어떻게 구성되었는지 빠르게 확인할 수 있습니다. 각 문법은 철도 다이어그램으로 설명합니다.

철도 다이어그램을 해석하는 방법은 간단합니다.

- 왼쪽에서 시작해서 트랙을 따라 오른쪽 끝으로 이동합니다.
- 오른쪽으로 가다 보면 둥근 도형 안의 리터럴이나 사각형 안에 있는 설명 또는 규칙을 만납니다.
- 트랙을 따라가는 것이라면 어떠한 경로라도 유효합니다.
- 트랙을 따라가지 않는 경로는 모두 무효입니다.
- 양 끝에 세로 막대 하나를 가진 철도 다이어그램은 한 쌍의 토큰 사이에 공백(whitespace)을 허용한다는 뜻이며, 세로 막대 두 개가 있는 경우는 그렇지 않다는 뜻입니다.

좋은 점(good parts)만을 설명하는 이번 장의 문법은 자바스크립트 전체 문법보다 훨씬 간단합니다.

01 | 공백(whitespace)



공백은 문자를 구분하는 형태나 주석의 형태를 취할 수 있습니다(즉, 주석 역시 공백입니다). 보통 공백은 별로 중요하지 않지만, 때때로 공백을 사용하지 않으면 하나의 토큰으로 묶여버리는 문자들을 분리하기 위해서 필요합니다. 다음은 그 예입니다.


```
var that = this;
```

1. 역자 주/ '줄 끝'이라는 표현이 많이 나올 것입니다. 줄 끝(줄 끝 문자)은 우리가 흔히 생각하는 CR이나 LF 등을 의미합니다. 그래서 '줄 끝'이라는 표현이 나오면 이런 의미로 생각하면 됩니다. 참고로 ECMAScript 명세에서 정의하고 있는 줄 종결자(Line Terminators)에는 CR, LF 외에도 LS(Line Separator), PS(Paragraph Separator)가 있습니다.

var와 that 사이에 있는 빈 칸은 제거할 수 없습니다. 하지만 다른 빈 칸들은 제거해도 상관 없습니다.

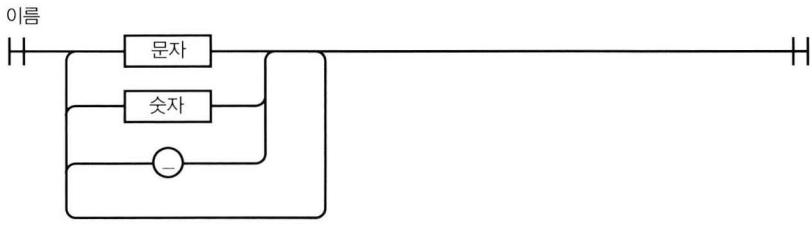
자바스크립트에서는 /* */ 형태의 블록 주석과 // 형태의 한 줄 주석을 사용할 수 있습니다. 주석은 프로그램의 가독성을 높이기 위해 충분히 사용되는 것이 좋습니다. 주석을 달 때는 항상 코드에 대해 정확히 설명해야 합니다. 쓸모 없는 주석은 없느니만 못합니다.

블록 주석을 달 때 사용하는 /* */는 PL/I이라는 언어에서 비롯된 것입니다. PL/I은 이 이상한 한 쌍을 주석을 위한 기호로 삼았는데 그 이유는 PL/I 프로그램에서는 문자열 리터럴을 제외하고는 이러한 조합이 거의 나타나지 않기 때문입니다. 하지만 자바스크립트에서는 이러한 조합이 정규 표현식 리터럴에서도 나타날 수 있습니다. 그러므로 블록 주석 방법은 안전하지 않다고 볼 수 있습니다. 예를 들어 다음과 같은 코드를 블록 주석으로 주석화하면 구문 오류가 발생합니다.

```
/*
    var rm_a =/a*/.match(5);
*/
```

그러므로 가능하면 /* */를 사용하는 대신 //를 사용할 것을 권합니다. 이 책에서는 //만을 사용하고 있습니다.

02 | 이름(Names)



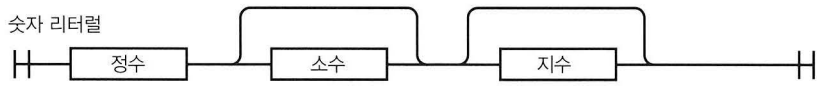
이름(name)은 하나의 문자나 그 뒤를 이어서 하나 이상의 문자, 숫자, _가 붙는 문자열로 문장, 변수, 매개변수, 속성명, 연산자, 라벨 등에 사용합니다. 다음에 나오는 예약어들은 이름(name)이 될 수 없습니다.

```
abstract
boolean break byte
case catch char class const continue
debugger default delete do double
else enum export extends
false final finally float for function
goto
if implements import in instanceof int interface
long
native new null
package private protected public
return
short static super switch synchronized
this throw throws transient true try typeof
var volatile void
while with
```

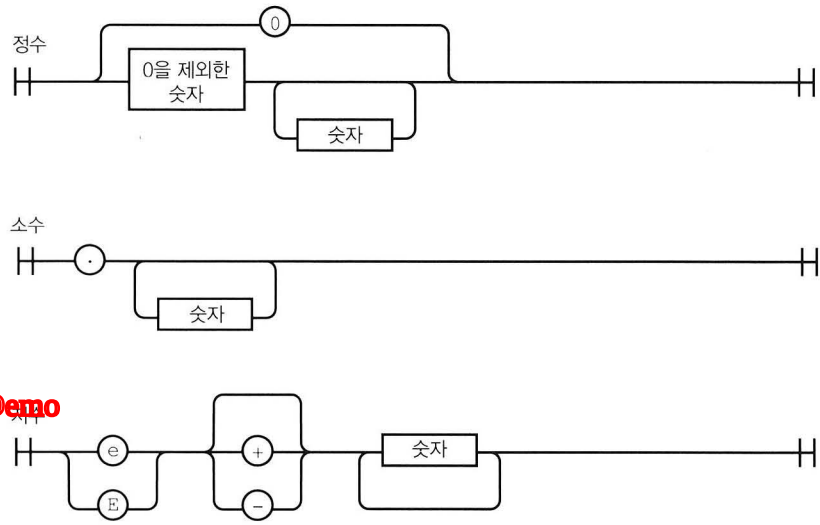
(이 목록에는 예약어에 꼭 포함할 것 같은 undefined, NaN, Infinity 등은 없습니다.)

여기에 열거된 예약어 대부분은 실제로 언어에서 사용하지 않습니다. 예약어는 변수 이름이나 매개변수 이름으로 사용할 수 없습니다. 게다가 예약어는 객체 리터럴의 속성명이나 객체의 속성을 나타낼 때 사용하는 마침표 다음에 사용할 수 없습니다.

03 | 숫자(Numbers)



자바스크립트는 숫자형이 하나만 있습니다. 내부적으로 숫자는 64비트 부동 소수점 형식을 지닙니다. 이는 자바의 double 형과 같습니다. 대부분의 다른 언어와는 달리 자바스크립트에는 정수와 실수의 구분이 없습니다. 즉 1과 1.0은 같은 값입니다. 이러한 특성은 매우 편리합니다. short 형을 사용해서 오버플로우가 발생하는 일 등이 전혀 없으며, 단지 알아야 할 것은 숫자형이라는 것뿐입니다. 결국 숫자형 때문에 발생하는 오류 대부분을 피할 수 있습니다.



숫자 리터럴이 지수 부분을 포함하는 경우 이 숫자 리터럴의 값은 e 앞의 값에다 e 뒤의 값만큼 10을 제곱한 값의 곱이 됩니다. 그래서 100은 1e2와 같습니다.

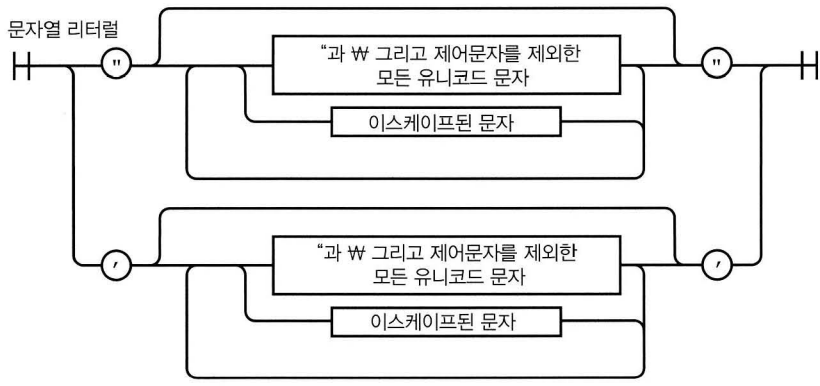
음수는 수 앞에 -를 붙이면 됩니다.

NaN은 수치 연산을 해서 정상적인 값을 얻지 못할 때의 값입니다. NaN은 그 자신을 포함해서 어떤 값하고도 같지 않습니다. 그러므로 NaN인지 확인하려면 비교 구문이 아니라 isNaN()이라는 함수를 사용합니다.

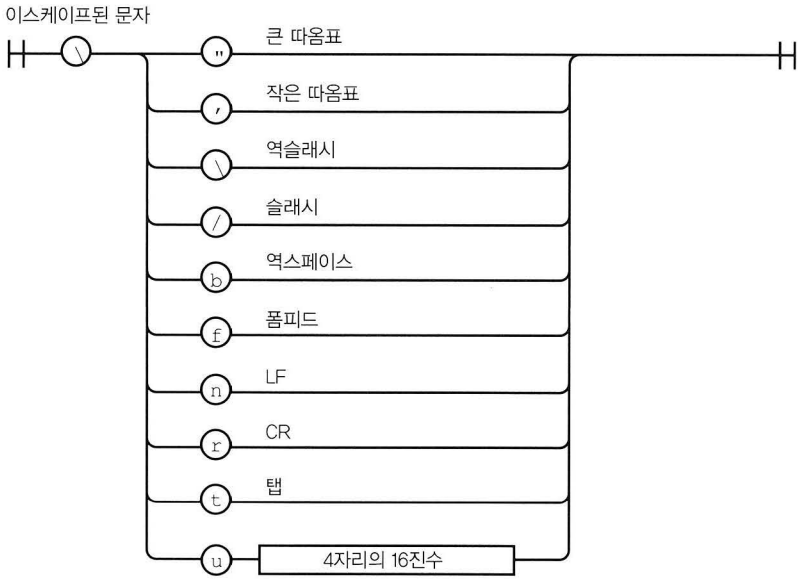
1.7976931348623157e+308 보다 큰 값은 Infinity로 나타납니다.

숫자는 메소드가 있습니다(8장 참조). 자바스크립트의 Math라는 객체에 수치 계산을 위한 메소드가 있습니다. 예를 들어 Math.floor(number) 메소드는 수를 정수로 변환할 때 사용합니다.

04 | 문자열(Strings)



문자열은 작은 따옴표나 큰 따옴표로 묶어서 나타내며, 따옴표 안에는 문자 0개 이상을 포함합니다. \(백슬래시)는 이스케이프 문자입니다. 자바스크립트는 유니코드가 16비트 문자 셋이었을 때 개발했기 때문에 자바스크립트 내의 모든 문자는 16비트 유니코드입니다.



자바스크립트에는 문자 타입이 없습니다. 그러므로 문자 하나를 나타내기 위해서는 문자 하나만을 포함하는 문자열을 사용해야 합니다.

이스케이프 시퀀스로 \나 따옴표, 제어문자처럼 일반 문자가 아닌 특별한 문자를 문자열에 삽입할 수 있습니다. \u로 시작하는 표기법은 유니코드 숫자값으로 문자를 나타낼 수 있습니다.


```
"A" === "\u0041"
```

문자열은 length라는 속성이 있습니다. "seven".length의 값은 5입니다.

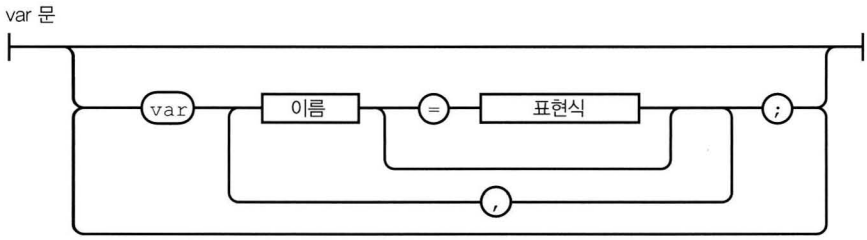
문자열은 변하지 않습니다(immutable). 일단 문자열이 한번 만들어지면, 이 문자열은 결코 변하지 않습니다. 하지만 여러 문자열을 + 연산자로 연결하여 새로운 문자열을 만들 수 있습니다. 분리된 문자열이 + 연산자로 연결되든 그냥 문자열 하나든 문자들의 순서가 같으면 같은 문자열입니다. 그래서 다음의 예는 참(true)입니다.

```
'c' + 'a' + 't' === 'cat'
```

문자열은 메소드가 있습니다(8장 참조).

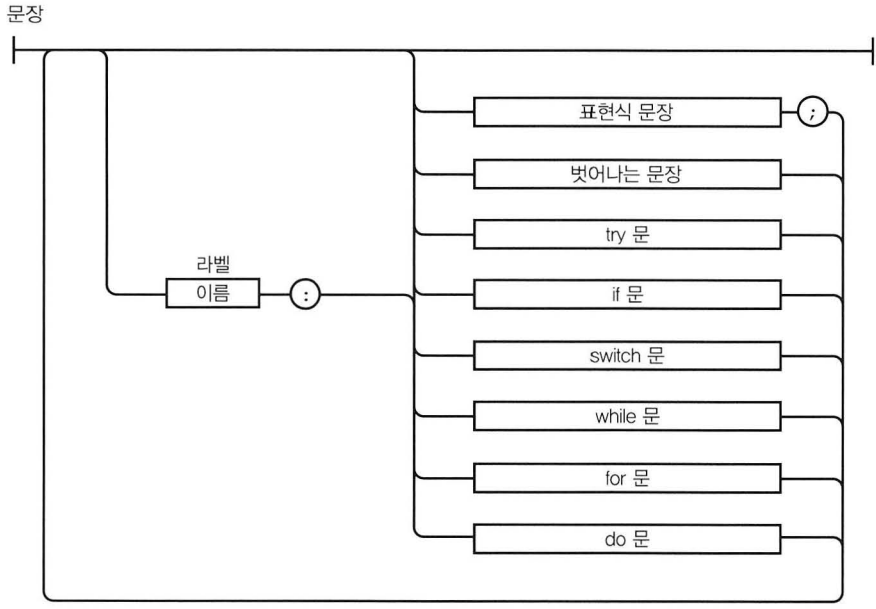
```
'cat'.toUpperCase( ) === 'CAT'
```

05 | 문장(Statements)

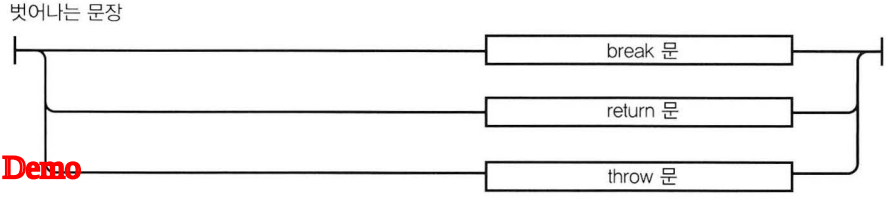


하나의 컴파일 단위에는 실행을 위한 문장들이 포함돼 있습니다. 웹 브라우저에서 각각의 〈script〉 태그는 컴파일되어 즉시 실행되는 하나의 컴파일 단위입니다. 링커(linker)가 없기 때문에 자바스크립트는 모든 문장을 공통적인 전역 이름 공간(namespace)에 한 데 몰아 넣습니다. 전역변수에 관한 좀더 자세한 부분은 부록 A에서 설명합니다.

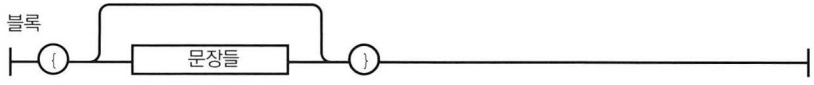
var 문은 함수 내부에서 사용될 때 함수의 private 변수를 정의합니다.



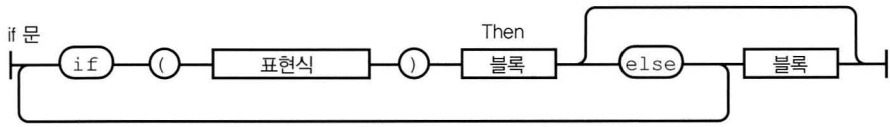
switch 문, while 문, for 문, do 문에는 break 문에서 사용할 수 있는 라벨을 선택적으로 지정할 수 있습니다.



문장들은 대개 위에서 아래로 순서대로 실행합니다. 이러한 실행 순서는 조건문(if, switch)이나 반복문(while, for, do) 또는 실행 흐름을 벗어나는 문장(break, return, throw)이나 함수 호출로 변경할 수 있습니다.



블록은 중괄호로 쌓인 문장들의 집합입니다. 다른 언어들과 달리 자바스크립트에서 블록은 새로운 유효범위(scope)를 생성하지 않습니다. 이러한 이유로 변수는 블록 안에서가 아니라 함수의 첫 부분에서 정의해야 합니다.

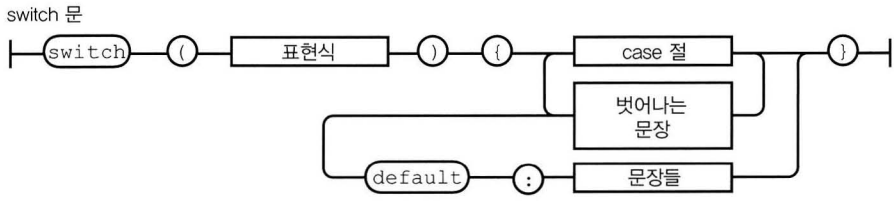


if 문은 표현식의 값에 따라 프로그램의 흐름을 변경합니다. then 블록은 표현식이 참(true)일 때 실행하며, 표현식이 거짓인 경우에는 else 블록을 실행합니다(물론 else 블록은 선택적입니다).

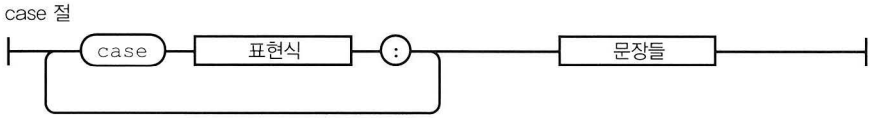
다음은 거짓에 해당하는 값들입니다.

- false
- null
- undefined
- 빈 문자열 ' '
- 숫자 0
- NaN

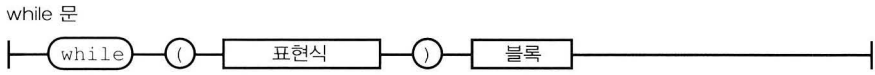
이 외의 모든 값은 참입니다. 예를 들어 true, 문자열 'false', 모든 객체 등은 모두 참입니다.



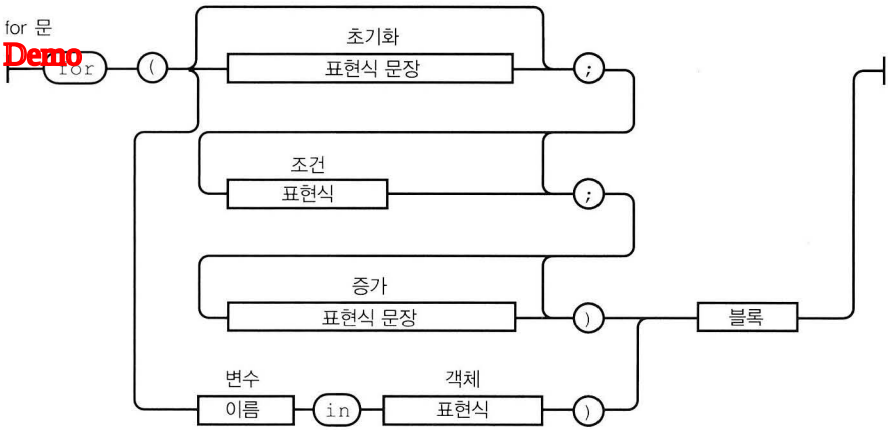
switch 문은 다중 분기를 수행합니다. 이 문장은 표현식과 모든 case 문의 표현식이 같은지를 비교합니다. 표현식의 결과값은 숫자일수도 있고 문자열일수도 있습니다. 일치하는 표현식을 찾으면 해당 case 절에 있는 문장들을 실행합니다. 만약 일치하는 표현식을 찾지 못하는 경우에는 default 절의 문장들을 실행합니다(default는 선택사항입니다).



case 절은 하나 이상의 case 문을 포함합니다. case 절의 표현식은 꼭 상수일 필요가 없습니다. case 절의 문장 마지막에는 다음 case 절로 넘어가지 않게 실행 흐름을 벗어나는 문장을 사용해야 합니다. 이를 위해 break 문을 사용할 수 있습니다.



while 문은 단순한 반복 수행 문장입니다. 표현식이 참인 동안은 블록을 반복해서 실행하며, 표현식이 거짓이면 반복 수행은 끝납니다.



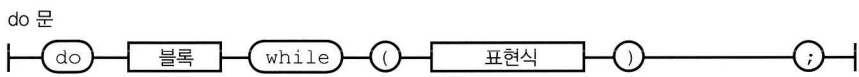
for 문은 좀더 복잡한 반복문입니다. 이 실행문은 두 가지 형식으로 사용합니다.

일반적인 형식은 초기화, 조건, 증가라는 세가지 절로 제어하는 구조입니다(다이어그램에서 볼 수 있는 것처럼 이 세가지 절은 모두 선택적입니다). 먼저 초기화를 실행합니다. 보통 이 부분에서 반복 횟수를 제어하는 변수를 초기화합니다. 그 다음에 조건 부분이 만족하는지를 검사합니다. 이 검사의 전형적인 형태는 반복 횟수를 제어하는 변수가 조건에 만족하는지를 확인하는 것입니다. 만약 조건 부분이 생략되면 참으로 간주합니다. 조건 부분의 검사 결과가 거짓이면 반복을 종료합니다. 조건을 만족해서 블록을 한 번 실행하면 증가 부분을 실행하고 다시 조건 검사를 반복합니다.

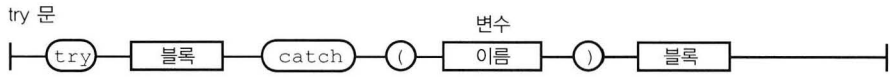
또 다른 for 문의 형식은 객체의 속성 이름(또는 키)을 열거하는 것입니다(이러한 형식을 for in이라고 부릅니다). 각각의 반복 실행마다 객체에 있는 각각의 속성 이름을 변수에 할당합니다.

이 for in 형식은 보통 다음과 같이 object.hasOwnProperty(변수) 메소드로 속성 이름이 실제로 객체의 속성인지 아니면 프로토타입 체인(prototype chain) 상에 있는 것인지를 확인하는 것이 필요합니다.

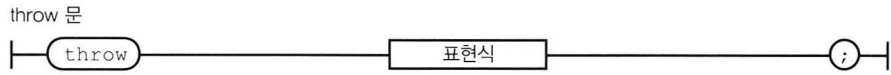
```
for (myvar in obj) {
    if (obj.hasOwnProperty(myvar)) {
        ...
    }
```

do 문은 표현식이 블록을 실행하기 전이 아니라 실행한 후에 검사된다는 점만 빼면 while 문과 같습니다. 이러한 특성 때문에 do 문은 적어도 한 번 블록을 실행합니다.

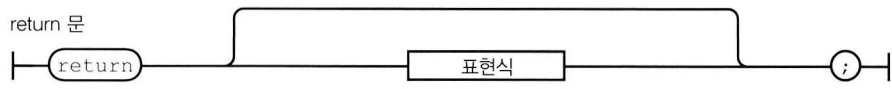


try 문은 블록을 실행하면서 블록 내에서 발생하는 예외 상황을 포착합니다. catch 절은 예외 객체를 받는 새로운 변수를 정의합니다.



throw 문은 예외를 발생합니다. 만약 throw 문이 try 블록 안에 있으면 실행의 제어는 catch 절로 이동합니다. 만약 throw 문이 일반적인 함수 내에 있다면 함수 호출은 중단되고 함수를 호출한 try 문의 catch 절로 실행 흐름이 이동합니다.

표현식 부분은 보통 name과 message라는 속성이 있는 객체 리터럴입니다. 예외를 포착한 곳에서는 이 객체의 정보로 무엇을 수행할지 결정할 수 있습니다.

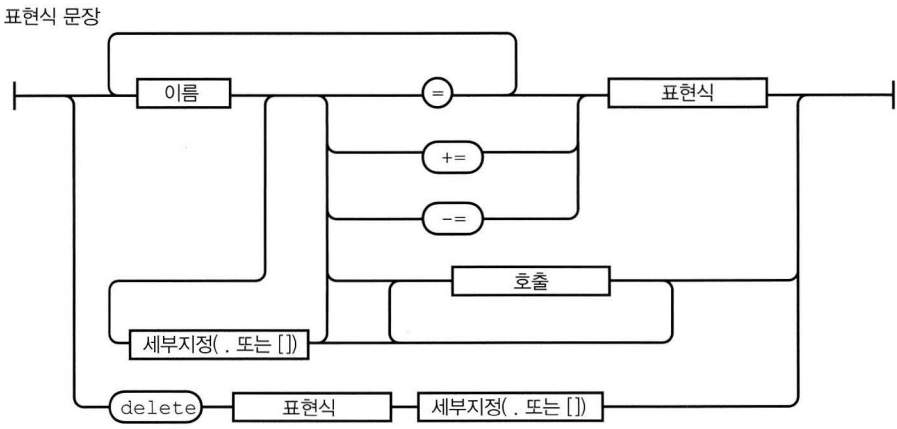


return 문은 함수에서 호출한 곳으로 되돌아가는 역할을 합니다. 또한 이 문장은 반환값을 지정합니다. 표현식이 지정되지 않으면 undefined를 반환합니다.

자바스크립트는 return과 표현식 부분 사이에 줄 바꿈을 허용하지 않습니다.

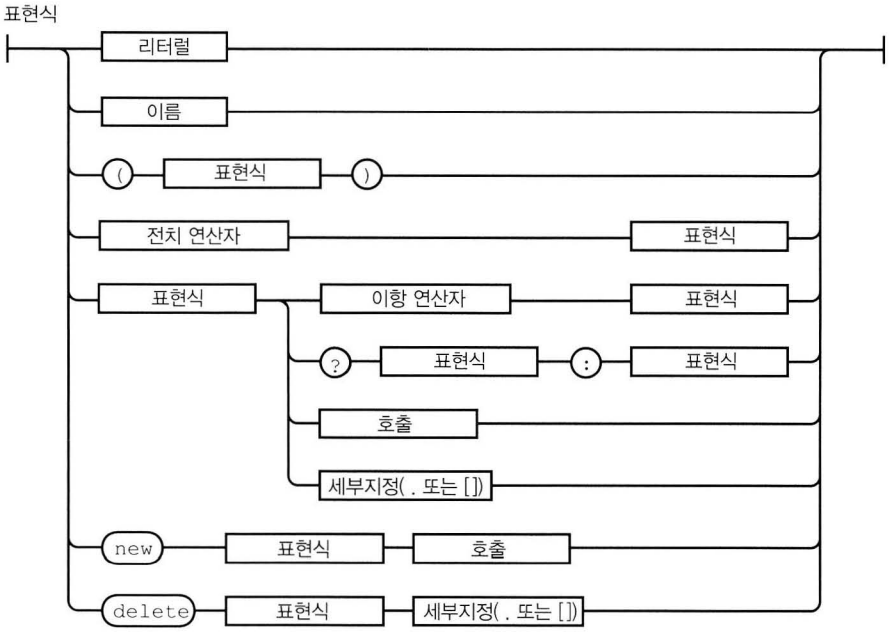
break 문은 반복문이나 switch 문에서 흐름을 벗어나게 하는 역할을 합니다. 이 문장은 라벨 이름을 취할 수 있는 데 라벨이 주어지면 라벨이 붙은 문장의 끝으로 이동합니다.

자바스크립트는 break와 라벨 사이에 줄 바꿈을 허용하지 않습니다.



표현식 문장은 값을 하나 이상의 변수나 객체의 속성에 할당하거나 메소드를 호출하고 객체의 속성을 삭제할 수 있습니다. = 연산자는 할당하는 데 사용합니다. 할당 연산자를 동등 연산자인 ===와 혼동해서는 안 됩니다. += 연산자는 더하거나 연결합니다.

06 | 표현식(Expressions)



가장 간단한 표현식은 리터럴 값(문자열이나 숫자), 변수, 내장값들(true, false, null, undefined, NaN, Infinity 등), new 키워드에 의한 호출 표현식, delete 키워드 다음에 나오는 세부지정 표현식, 괄호로 쌓인 표현식, 전치 연산자 다음에 이어지는 표현식 등입니다. 그리고 그 외에도 다음과 같은 표현식이 있습니다.

- 이항 연산자의 표현식
- ? 삼항 연산자의 표현식
- 호출
- 세부지정(. 또는 [])

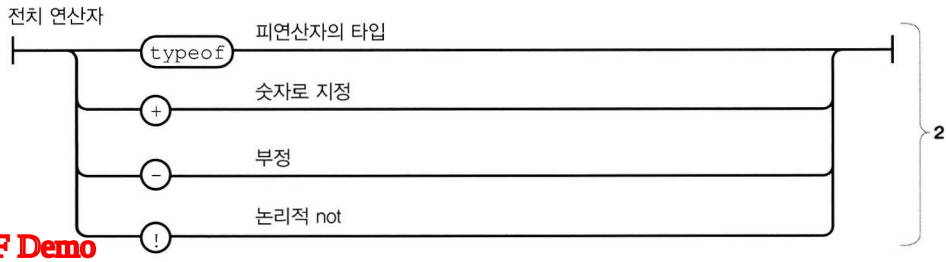
? 삼항 연산자는 3개의 피연산자를 취합니다. 첫 번째 피연산자가 참이면, 두 번째 피연산자가 값이 되지만, 첫 번째 피연산자가 거짓인 경우에는 세 번째 피연산자가 값이 됩니다.

[표 2-1] 목록에서 상위에 있는 연산자일수록 우선 순위가 높습니다. 괄호를 사용하면 일반적인 우선 순위를 변경하여 높은 우선 순위를 갖게 할 수 있습니다. 다음은 그 예입니다.

```
2 + 3 * 5 === 17
(2 + 3) * 5 === 25
```

[표 2-1] 연산자 우선 순위

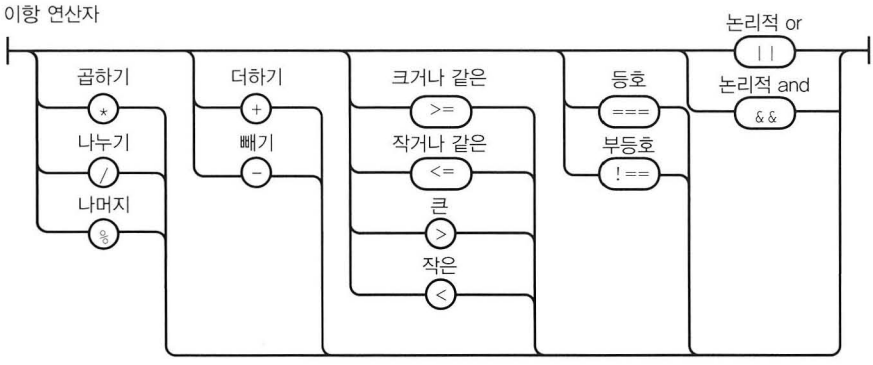
. [] ( )	세부지정이나 호출
delete new typeof + - !	단항 연산자
* / %	곱하기, 나누기, 나머지
+ -	더하기/연결, 빼기
>= <= > <	같지 않음 비교
=== !==	동등
&&	논리적 and
||	논리적 or
?:	삼항



typeof 연산자의 결과값에는 number, string, boolean, undefined, function, object 등이 있습니다. 피연산자가 배열이나 null이면 결과는 모두 object인데 이는 약간 문제가 있습니다. 이 문제를 포함하여 typeof 연산자에 대한 자세한 부분은 6장과 부록 A에서 살펴봅니다.

2. 역자 주/ + 는 피연산자가 숫자일 경우에는 크게 의미가 없거나 양수를 명확히 나타내지만 피연산자가 숫자가 아닐 경우에는 이를 숫자로 변환하는 역할을 합니다. 숫자로 변환할 수 없을 경우에는 NaN을 반환합니다. -는 피연산자를 부정합니다. 즉 피연산자가 음수이면 양수로, 양수이면 음수로 변환합니다.

! 연산자의 피연산자 값이 참이면 결과값은 거짓이며 그 반대면 결과값은 참입니다.

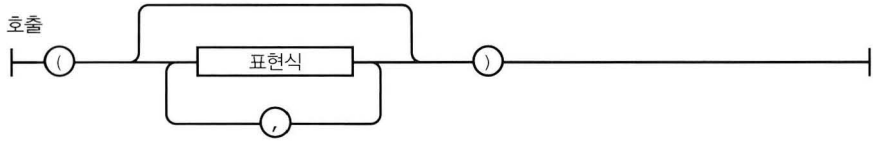


+ 연산자는 수를 더하거나 문자열을 연결합니다. 더하기 연산을 원하는 경우에는 반드시 피연산자 두 개가 모두 숫자여야 합니다.

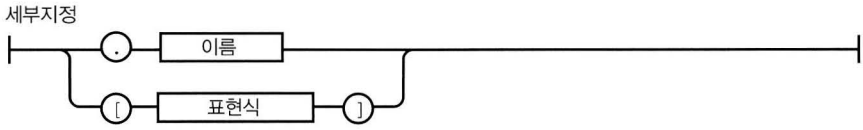
/ 연산자는 피연산자 두 개가 모두 정수형이어도 결과값이 실수일 수 있습니다.

&& 연산자는 첫 번째 피연산자가 거짓일 경우 첫 번째 피연산자의 값을 취하고, 그렇지 않은 경우에는 두 번째 피연산자 값을 결과값으로 취합니다.

|| 연산자는 &&와 반대로 첫 번째 피연산자가 참이면 이 값을 취하고 그렇지 않으면 두 번째 피연산자를 결과값으로 취합니다.

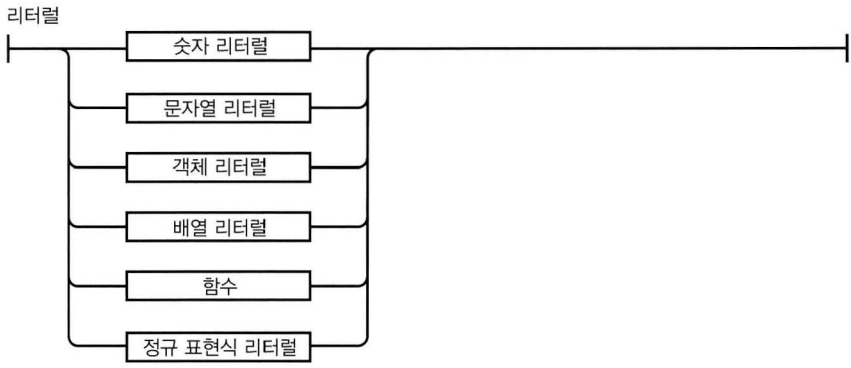


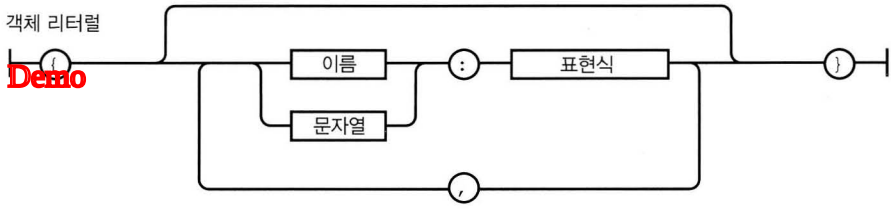
호출은 함수를 실행합니다. 호출 연산자는 함수 이름 뒤에 이어지는 한 쌍의 괄호입니다. 괄호 내에는 함수에 전달하는 인수를 포함할 수 있습니다. 함수는 4장에서 자세히 살펴봅니다.



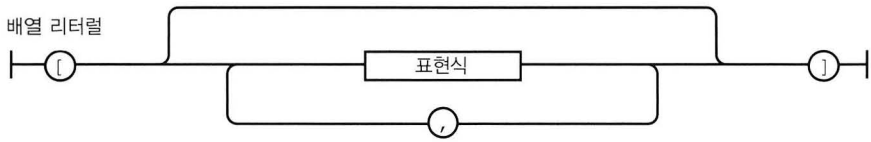
세부지정은 객체의 속성이나 배열의 구성요소를 지정할 때 사용합니다. 세부지정은 다음 장에서 자세히 살펴봅니다.

07 | 리터럴(Literals)

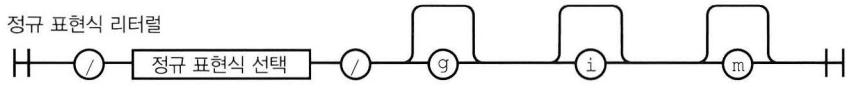




객체 리터럴은 새로운 객체를 생성할 때 편리한 표기법입니다. 속성명은 이름이나 문자열로 지정할 수 있습니다. 속성명은 변수가 아니라 리터럴 이름이기 때문에 객체의 속성명은 반드시 컴파일 시에 알려져야 합니다. 속성의 값은 표현식입니다. 객체 리터럴은 다음 장에서 자세히 살펴봅니다.

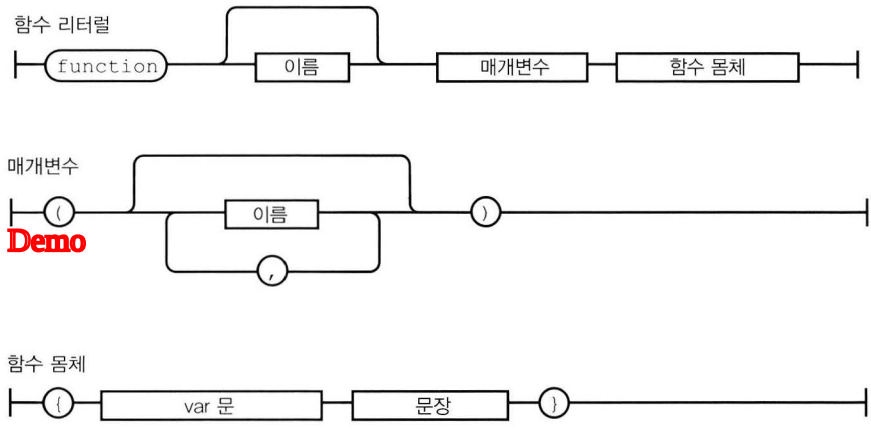


배열 리터럴은 새로운 배열을 생성할 때 편리한 표기법입니다. 배열은 6장에서 보다 자세하게 살펴봅니다.



정규 표현식은 7장에서 자세히 살펴봅니다.

08 | 함수(Functions)



함수 리터럴은 함수값을 정의합니다. 함수 리터럴은 이름을 가질 수 있는데 이 이름은 자신을 재귀적으로 호출할 때 사용할 수 있습니다. 또한 함수 리터럴은 매개변수 목록을 가질 수 있는데 이 매개변수는 함수 호출 시 넘어온 인수로 초기화되는 변수입니다. 함수 몸체는 변수 정의와 문장들을 포함합니다. 함수의 자세한 내용은 4장에서 살펴봅니다.

C.H.A.P.T.E.R. 03

객체

하찮은 것(object)에 사랑은 눈을 감으니까요
– 윌리엄 셰익스피어, 베로나의 두 신사

자바스크립트에서 단순한 데이터 타입은 숫자, 문자열, 불리언(true/false), null, undefined가 있습니다. 이들을 제외한 다른 값들은 모두 객체입니다. 숫자와 문자열 그리고 불리언은 메소드가 있기 때문에 유사 객체라고 할 수 있습니다. 하지만 이들은 값이 한번 정해지면 변경할 수가 없습니다(immutable). 자바스크립트의 객체는 변형 가능한 속성들의 집합이라고 할 수 있습니다. 자바스크립트에서는 배열, 함수, 정규 표현식 등과 (당연히) 객체 모두가 객체입니다.

객체는 이름과 값이 있는 속성들을 포함하는 컨테이너라고 할 수 있습니다. 속성의 이름은 문자열이면 모두 가능합니다. 여기에는 빈 문자열도 포함합니다. 속성의 값은 undefined를 제외한 자바스크립트의 모든 값이 사용될 수 있습니다.

자바스크립트의 객체는 클래스가 필요 없습니다(class-free). 새로운 속성의 이름이나 값에 어떠한 제약 사항이 없습니다. 객체는 데이터를 한 곳에 모으고 구조화하는데 유용합니다. 객체 하나는 다른 객체를 포함할 수 있기 때문에, 그래프나 트리 같은 자료구조를 쉽게 표현할 수 있습니다.

자바스크립트에는 객체 하나에 있는 속성들을 다른 객체에 상속하게 해주는 프로토타입(prototype) 연결 특성이 있습니다. 이 특성을 잘 활용하면, 객체를 초기화하는 시간과 메모리 사용을 줄일 수 있습니다.

01 | 객체 리터럴

객체 리터럴은 새로운 객체를 생성할 때 매우 편리한 표기법을 제공합니다. 이 표기법은 아무 것도 없거나 하나 이상의 이름/값 쌍들을 둘러싸는 중괄호이며, 표현식이 있을 수 있는 곳이라면 어디라도 위치할 수 있습니다.

```
var empty_object = {};

var stooge = {
    "first-name": "Jerome",
    "last-name": "Howard"
};
```

속성(property)의 이름은 어떤 문자열이라도 가능합니다 여기에는 빈 문자열도 포함합니다. 속성 이름에 사용한 따옴표는 속성 이름이 자바스크립트에서 사용할 수 있는 유효한 이름이고 예약어가 아닐 경우에는 생략할 수 있습니다. 그러므로 "first-name"이라는 속성명은 반드시 따옴표를 사용해야 하지만, first_name은 사용해도 되고 안 해도 됩니다. 쉼표(,)는 "속성 이름": "값" 쌍들을 구분하는데 사용합니다.

속성의 값은 어떠한 표현식도 가능합니다. 여기에는 다음의 예처럼 객체 리터럴도 가능합니다(중첩된 객체).

```
var flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

02 | 속성값 읽기

객체에 속한 속성의 값은 속성 이름을 대괄호([])로 둘러싼 형태로 읽을 수 있습니다. 속성 이름이 유효한 자바스크립트 이름이고 예약어가 아닐 경우에는 마침표(.) 표기법을 대신 사용할 수 있습니다. 마침표 표기법은 보다 간단하고 읽기가 편하기 때문에 보통 더 선호합니다.

```
stooge["first-name"]      // "Joe"
flight.departure.IATA     // "SYD"
```


객체에 존재하지 않는 속성을 읽으려고 하면 undefined를 반환합니다.

```
stooge["middle-name"]     // undefined
flight.status             // undefined
stooge["FIRST-NAME"]      // undefined
```

|| 연산자를 사용하여 다음과 같이 기본값을 지정할 수 있습니다.

```
var middle = stooge["middle-name"] || "(none)";
var status = flight.status || "unknown";
```

존재하지 않는 속성, 즉 undefined의 속성을 참조하려 할 때 TypeError 예외가 발생합니다. 이런 상황을 방지하기 위해서 다음과 같이 && 연산자를 사용할 수 있습니다.

```
flight.equipment                              // undefined
flight.equipment.model                        // throw "TypeError"
flight.equipment && flight.equipment.model // undefined
```

03 | 속성값의 갱신

객체의 값은 할당에 의해 갱신합니다. 만약 할당하는 표현식에서 속성 이름이 이미 객체 안에 존재하면 해당 속성의 값만 교체합니다.

```
stooge['first-name'] = 'Jerome';
```


이와 반대로 속성이 객체 내에 존재하지 않는 경우에는 해당 속성을 객체에 추가합니다.

```
stooge['middle-name'] = 'Lester';
stooge.nickname = 'Curly';
flight.equipment = {
    model: 'Boeing 777'
};
flight.status = 'overdue';
```

04 | 참조

객체는 참조 방식으로 전달됩니다. 결코 복사되지 않습니다.

```
var x = stooge;
x.nickname = 'Curly';
var nick = stooge.nickname;
    // x와 stooge가 모두 같은 객체를 참조하기 때문에,
    // 변수 nick의 값은 'Curly'.

var a = {}, b = {}, c = {};
    // a, b, c는 각각 다른 빈 객체를 참조
a = b = c = {};
    // a, b, c는 모두 같은 빈 객체를 참조
```

05 | 프로토타입(Prototype)

모든 객체는 속성을 상속하는 프로토타입 객체에 연결돼 있습니다. 객체 리터럴로 생성되는 모든 객체는 자바스크립트의 표준 객체인 Object의 속성인 prototype (Object.prototype) 객체에 연결됩니다.

객체를 생성할 때는 해당 객체의 프로토타입이 될 객체를 선택할 수 있습니다. 이를 위해 자바스크립트가 제공하는 메커니즘은 좀 까다롭고 복잡하지만 조금만 신경을 쓰면 매우 단순화할 수 있습니다. 이제 Object 객체에 create라는 메소드를 추가할 것입니다. create는 넘겨받은 객체를 프로토타입으로 하는 새로운 객체를 생성하는 메소드입니다. 다음에 나오는 create 메소드의 코드를 보면 함수를 사용하는 부분이 나오는데 함수에 관한 자세한 내용은 다음 장에서 살펴봅니다.[1]

1. 역자 주/ create 메소드는 이후로도 자주 언급하기 때문에 잘 기억해두기 바랍니다.

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        var F = function () {};
        F.prototype = o;
        return new F();
    };
}
var another_stooge = Object.create(stooge);
```

프로토타입 연결은 값의 갱신에 영향을 받지 않습니다. 즉 객체를 변경하더라도 객체의 프로토타입에는 영향을 미치지 않습니다.

```
another_stooge['first-name'] = 'Harry';
another_stooge['middle-name'] = 'Moses';
another_stooge.nickname = 'Moe';
```

프로토타입 연결은 오로지 객체의 속성을 읽을 때만 사용합니다. 객체에 있는 특정 속성의 값을 읽으려고 하는데 해당 속성이 객체에 없는 경우 자바스크립트는 이 속성을 프로토타입 객체에서 찾으려고 합니다. 이러한 시도는 프로토타입 체인(prototype chain)의 가장 마지막에 있는 Object.prototype까지 계속해서 이어집니다. 만약 찾으려는 속성이 프로토타입 체인 어디에도 존재하지 않는 경우 undefined를 반환합니다. 이러한 일련의 내부 동작을 위임(delegation)이라고 합니다.

프로토타입 관계는 동적 관계입니다. 만약 프로토타입에 새로운 속성이 추가되면, 해당 프로토타입을 근간으로 하는 객체들에는 즉각적으로 이 속성이 나타납니다.

```
stooge.profession = 'actor';
another_stooge.profession    // 'actor'
```

프로토타입 체인(prototype chain)과 관련된 내용은 6장에서도 살펴봅니다.[2]

06 | 리플렉션(reflection)

객체에 어떤 속성들이 있는지는 특정 속성을 접근해서 반환하는 값을 보면 쉽게 알 수 있습니다. 이때 typeof 연산자는 속성의 타입을 살펴보는데 매우 유용합니다.

```
typeof flight.number       // 'number'
typeof flight.status       // 'string'
typeof flight.arrival      // 'object'
typeof flight.manifest     // 'undefined'
```

때때로 해당 객체의 속성이 아니라 프로토타입 체인 상에 있는 속성을 반환할 수 있기 때문에 주의할 필요가 있습니다.

```
typeof flight.toString     // 'function'
typeof flight.constructor  // 'function'
```

리플렉션을 할 때 원하지 않는 속성을 배제하기 위한 두 가지 방법이 있습니다. 첫 번째 방법은 함수값을 배제하는 방법입니다. 일반적으로 리플렉션을 할 때는 데이터에 관심이 있기 때문에 함수가 반환되는 경우를 염두에 두고 있다가 배제시키면 원하지 않는 속성을 배제할 수 있습니다.

또 다른 방법은 객체에 특정 속성이 있는지를 확인하여 true/false 값을 반환하는 hasOwnProperty 메소드를 사용하는 것입니다. hasOwnProperty 메소드는 프로토타입 체인을 바라보지 않습니다.

2. 역자 주/ 예제 코드에 나오는 stooge와 another_stooge로 프로토타입 개념을 설명하면 다음과 같습니다. another_stooge의 프로토타입은 stooge고 stooge는 another_stooge의 프로토타입 객체가 됩니다. 그리고 another_stooge가 자신의 프로토타입인 stooge에 연결돼 있고 또 stooge는 다른 객체에 연결돼 있고 하는 식으로 특정 객체의 프로토타입 연결 전체를 프로토타입 체인으로 보면 됩니다.

```
flight.hasOwnProperty('number')          // true
flight.hasOwnProperty('constructor')     // false
```

07 | 열거(Enumeration)

for in 구문을 사용하면 객체에 있는 모든 속성의 이름을 열거할 수 있습니다. 이러한 열거 방법에는 함수나 프로토타입에 있는 속성 등 모든 속성이 포함되기 때문에 원하지 않는 것들을 걸러낼 필요가 있습니다. 가장 일반적인 필터링 방법은 hasOwnProeprty 메소드와 함수를 배제하기 위한 typeof를 사용하는 것입니다. 다음은 그 예입니다.

```
var name;
for (name in another_stooge) {
    if (typeof another_stooge[name] !== 'function') {
        document.writeln(name + ': ' + another_stooge[name]);
    }
}
```

for in 구문을 사용하면 속성들이 이름순으로 나온다는 보장이 없습니다. 그러므로 만약 특정 순으로 속성 이름들이 열거되기를 원한다면 for in 구문을 사용하지 말고, 다음의 예처럼 속성이 열거되기 원하는 순서를 특정 배열로 지정하고 이 배열을 이용하여 객체의 속성을 열거할 수 있습니다.


```
var i;
var properties = [
    'first-name',
    'middle-name',
    'last-name',
    'profession'
];
for (i = 0; i < properties.length; i += 1) {
    document.writeln(properties[i] + ': ' +
            another_stooge[properties[i]]);
}
```

이렇게 하면 프로토타입 체인에 있는 속성들이 나오지 않을까 염려할 필요도 없으며 원하는 순서대로 속성들을 리플렉션할 수 있습니다.

08 | 삭제

delete 연산자를 사용하면 객체의 속성을 삭제할 수 있습니다. delete 연산자는 해당 속성이 객체에 있을 경우에 삭제를 하며 프로토타입 연결 상에 있는 객체들은 접근하지 않습니다.

객체에서 특정 속성을 삭제했는데 같은 이름의 속성이 프로토타입 체인에 있는 경우 프로토타입의 속성이 나타납니다. 다음의 예제를 보기 바랍니다.

```
another_stooge.nickname     // 'Moe'

// another_stooge에서 nickname을 제거하면
// 프로토타입에 있는 nickname이 나타남.

delete another_stooge.nickname;

another_stooge.nickname     // 'Curly'
```


09 | 최소한의 전역변수 사용

자바스크립트에서는 전역변수 사용이 매우 쉽습니다. 불행히도 전역변수는 프로그램의 유연성을 약화하기 때문에 가능하면 피하는 것이 좋습니다.

전역변수 사용을 최소화하는 방법 한 가지는 애플리케이션에서 전역변수 사용을 위해 다음과 같이 전역변수 하나를 만드는 것입니다.

```
var MYAPP = {};
```

이제 이 변수를 다른 전역변수를 위한 컨테이너로 사용합니다.

```
MYAPP.stooge = {
    "first-name": "Joe",
    "last-name": "Howard"
};

MYAPP.flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

이러한 방법으로 애플리케이션에 필요한 전역변수를 이름 하나로 관리하면 다른 애플리케이션이나 위젯 또는 라이브러리들과 연동할 때 발생하는 문제점을 최소화할 수 있습니다. 또한 MYAPP.stooge가 명시적으로 전역변수라는 것을 나타내기 때문에 프로그램의 가독성도 높입니다. 다음 장에서는 정보은닉을 위해 클로저(closure) 사용 방법을 살펴볼 것인데, 이 방법은 전역변수 사용을 줄이는 효과적인 방법 중에 하나입니다.

C.H.A.P.T.E.R. 04

함수

허나 죄란 어느 것이나 범행되기 전부터 처벌되기로 정해있는 것.
... 나의 직무(function)상 안 될 말이요.

– *윌리엄 셰익스피어, 以尺報尺*

자바스크립트에서 가장 좋은 점 중에 하나는 함수의 구현 부분입니다. 자바스크립트에서 함수는 거의 대부분 제대로 된 특성들로 이루어져 있습니다. 하지만 예상할 수 있는 것처럼 모든 부분이 그런 것은 아닙니다.

함수는 실행 문장들의 집합을 감싸고 있습니다. 함수는 자바스크립트에서 모듈화의 근간입니다. 함수는 코드의 재사용이나 정보의 구성 및 은닉 등에 사용하고, 객체의 행위를 지정하는데도 사용합니다. 일반적으로 프로그래밍 기술은 요구사항의 집합을 함수와 자료구조의 집합으로 변환하는 것입니다.

01 | 함수 객체

자바스크립트에서 함수는 객체입니다. 객체는 앞서도 설명한 것처럼 프로토타입 객체로 숨겨진 연결을 갖는 이름/값 쌍들의 집합체입니다. 객체 중에서 객체 리터럴로 생성되는 객체는 Object.prototype에 연결됩니다. 반면에 함수 객체는

Function.prototype에 연결됩니다(Function은 Object.prototype에 연결됩니다). 또한 모든 함수는 숨겨져 있는 두 개의 추가적인 속성이 있는데, 이 속성들은 함수의 문맥(context)과 함수의 행위를 구현하는 코드(code)입니다.

또한 모든 함수 객체는 prototype이라는 속성이 있습니다. 이 속성의 값은 함수 자체를 값으로 갖는 constructor라는 속성이 있는 객체입니다. 이는 Function.prototype으로 숨겨진 연결과는 구분됩니다. 이렇게 복잡하게 얽힌 구조는 다음 장에서 살펴봅니다.[1]

함수는 객체이기 때문에 다른 값들처럼 사용할 수 있습니다. 함수는 변수나 객체, 배열 등에 저장되며, 다른 함수에 전달하는 인수로도 사용하고, 함수의 반환값으로도 사용합니다.

함수를 다른 객체와 구분 짓는 특징은 호출할 수 있다는 것입니다.

02 | 함수 리터럴

함수 객체는 함수 리터럴로 생성할 수 있습니다.

```
// add라는 변수를 생성하고 두 수를 더하는 함수를
// 이 변수에 저장.
 var add = function (a, b) {
return a + b;
};
```


함수 리터럴에는 네 가지 부분이 있습니다. 첫 번째 부분은 function이라는 예약어입니다.

1. 역자 주/ 뒤에 다시 언급하지만 함수의 prototype 속성의 값을 객체 리터럴로 나타내면 다음과 같습니다.

```
{
    constructor: this
}
```

두 번째 부분은 선택사항으로 함수의 이름입니다. 함수의 이름은 함수를 재귀적으로 호출할 때 사용하며, 디버거나 개발 툴에서 함수를 구분할 때도 사용합니다. 앞선 예처럼 함수의 이름이 주어지지 않은 경우 익명함수(anonymous)라고 부릅니다.

세 번째 부분은 괄호로 둘러싸인 함수의 매개변수 집합입니다. 괄호 안에 아예 없거나 하나 이상의 매개변수를 쉼표로 분리해서 열거합니다. 이 매개변수들은 함수 내에서 변수로 정의합니다. 일반적인 변수들을 undefined로 초기화하는 것과는 달리 매개변수는 함수를 호출할 때 넘겨진 인수로 초기화합니다.

네 번째 부분은 중괄호로 둘러싸인 문장들의 집합입니다. 이러한 문장들은 함수의 몸체(body)이며 함수를 호출했을 때 실행합니다.

함수 리터럴은 표현식이 나올 수 있는 곳이면 어디든지 위치할 수 있습니다. 함수는 다른 함수 내에서도 정의할 수 있습니다. 물론 이러한 내부 함수도 매개변수와 변수를 가질 수 있으며 자신을 포함하고 있는 함수의 매개변수와 변수에도 접근할 수 있습니다. 함수 리터럴로 생성한 함수 객체는 외부 문맥으로의 연결이 있는데 이를 클로저(closure)라고 합니다. 클로저는 강력한 표현력의 근원입니다.[2]

03 | 호출

함수를 호출하면 현재 함수의 실행을 잠시 중단하고 제어를 매개변수와 함께 호출한 함수로 넘깁니다. 모든 함수는 명시되어 있는 매개변수에 더해서 this와 arguments라는 추가적인 매개변수 두 개를 받게 됩니다. this라는 매개변수는 객체지향 프로그래밍 관점에서 매우 중요하며, 이 매개변수의 값은 호출하는 패턴에 의해 결정됩니다. 자바스크립트에는 함수를 호출하는데 메소드 호출 패턴, 함수 호출 패턴, 생성자 호출 패턴, apply 호출 패턴이라는 네 가지 패턴이 있습니다. 각각의 패턴에 따라 this라는 추가적인 매개변수를 다르게 초기화합니다.

2. 역자 주/ 함수 리터럴과 관련해서 부록 B의 09. 함수 문장 vs 함수 표현식 절을 참조하세요.

함수를 호출하는 호출 연산자는 함수를 나타내는 표현식 뒤에 이어지는 한 쌍의 괄호입니다. 괄호 안에는 표현식을 포함하지 않거나, 하나나 또는 쉼표로 구분해서 둘 이상의 표현식을 포함합니다. 각각의 표현식은 인수값 하나를 산출합니다. 각각의 인수값을 함수의 매개변수에 각각 할당합니다. 함수를 호출할 때 넘기는 인수의 개수와 매개변수의 개수가 일치하지 않아도 실행시간 오류는 발생하지 않습니다. 만약 인수가 더 많을 경우 매개변수 수보다 초과하는 인수는 무시합니다. 그리고 인수가 매개변수 수보다 적은 경우에는 남는 매개변수에 undefined를 할당합니다. 인수에 대한 타입 체크는 없습니다. 어떠한 값이 넘어오든지 그대로 매개변수에 할당합니다.

메소드 호출 패턴

함수를 객체의 속성에 저장하는 경우 이 함수를 메소드라고 부릅니다. 메소드를 호출할 때, this는 메소드를 포함하고 있는 객체에 바인딩됩니다(즉, this는 객체 자체가 됩니다). 호출되는 표현식이 세부지정(마침표나 [])을 포함하고 있으면 이 방법이 메소드 호출 패턴입니다.

```
// value와 increment 메소드가 있는 myObject 생성.
// increment 메소드의 매개변수는 선택적.
// 인수가 숫자가 아니면 1이 기본값으로 사용됨.

var myObject = {
    value: 0,
    increment: function (inc) {
        this.value += typeof inc === 'number' ? inc : 1;
    }
};

myObject.increment( );
document.writeln(myObject.value);     // 1

myObject.increment(2);
document.writeln(myObject.value);     // 3
```

메소드는 자신을 포함하는 객체의 속성들에 접근하기 위해서 this를 사용할 수 있습니다. 즉 this를 사용해서 객체의 값을 읽거나 변경할 수 있습니다. this와 객체의 바인딩은 호출 시에 일어납니다. 이렇게 매우 늦은 바인딩은 this를 효율적으로 사용하는 함수를 만들 수 있습니다. 자신의 객체 문맥을 this로 얻는 메소드를 퍼블릭(public) 메소드라고 부릅니다.

함수 호출 패턴

함수가 객체의 속성이 아닌 경우에는 함수로서 호출합니다.

```
var sum = add(3, 4);    // 합은 7
```

함수를 이 패턴으로 호출할 때 this는 전역객체에 바인딩됩니다. 이런 특성은 언어 설계 단계에서의 실수입니다. 만약 언어를 바르게 설계했다면, 내부 함수를 호출할 때 이 함수의 this는 외부 함수의 this 변수에 바인딩되어야 합니다. 이러한 오류의 결과는 메소드가 내부 함수를 사용하여 자신의 작업을 돕지 못한다는 것입니다. 왜냐하면, 내부 함수는 메소드가 객체 접근을 위해 사용하는 this에, 자신의 this를 바인딩하지 않고 엉뚱한 값(전역객체)에 연결하기 때문입니다. 다행히도 이러한 문제를 해결하기 위한 쉬운 대안이 있습니다. 그 대안은 메소드에서 변수를 정의한 후 여기에 this를 할당하고, 내부 함수는 이 변수를 통해서 메소드의 this에 접근하는 방법입니다. 관례상 이 변수의 이름을 that이라고 하면 다음의 예와 같이 구현할 수 있습니다.


```
// myObject에 double 메소드를 추가

myObject.double = function ( ) {
    var that = this;     // 대안

    var helper = function ( ) {
        that.value = add(that.value, that.value);
```

```
        };

        helper( );      // helper를 함수로 호출
    };

    // double을 메소드로 호출

    myObject.double( );
    document.writeln(myObject.getValue( ));     // 6
```

3

생성자 호출 패턴

자바스크립트는 프로토타입에 의해서 상속이 이루어지는 언어입니다. 이 말은 객체가 자신의 속성들을 다른 객체에 바로 상속할 수 있다는 뜻입니다. 자바스크립트는 클래스가 없습니다.

이러한 특성은 현존하는 언어들의 경향과는 조금 다른 급진적인 것입니다. 오늘날 대부분의 언어는 클래스를 기반으로 하고 있습니다. 프로토타입에 의한 상속은 매우 표현적이지만 널리 알려져 있지 않습니다. 자바스크립트 자체도 자신의 프로토타입적 본성에 확신이 없었던지, 클래스 기반의 언어들을 생각나게 하는 객체 생성 문법을 제공합니다. 클래스 기반의 프로그래밍에 익숙한 프로그래머들에게 프로토타입에 의한 상속은 받아들여지지 못했고, 클래스를 사용하는 듯한 구문은 자바스크립트의 진정한 프로타입적 속성을 애매하게 만들었습니다. 이는 양쪽에게 모두 최악의 결과라고 할 수 있습니다.

함수를 new라는 전치 연산자와 함께 호출하면, 호출한 함수의 prototype 속성의 값에 연결되는 (숨겨진) 링크를 갖는 객체가 생성되고, 이 새로운 객체는 this에 바인딩됩니다.

3. 역자 주/ 예제의 경우 위의 예부터 계속 이어져야지만 60이라는 결과를 얻을 수 있습니다. 또한 myObject에서 value 속성의 getter라고 할 수 있는 myObject.getValue()도 정의돼 있지 않습니다. 이 책에 나와있는 예제들이 이렇게 앞의 내용과 이어지는 가운데 해당 부분만 보여주는 경우가 많으므로 주의할 필요가 있습니다. 이 책의 사이트에서 예제를 내려받아 확인해보세요.

new라는 전치 연산자는 return 문장의 동작을 변경합니다. 이 내용은 뒤에서 더 자세히 살펴봅니다.

```
// Quo라는 생성자 함수를 생성.
// 이 함수는 status라는 속성을 가진 객체를 생성함.

var Quo = function (string) {
    this.status = string;
};

// Quo의 모든 인스턴스에 get_status라는 public 메소드를 줌.

Quo.prototype.get_status = function ( ) {
    return this.status;
};

// Quo의 인스턴스 생성

var myQuo = new Quo("confused");

document.writeln(myQuo.get_status( )); // confused
```

new라는 전치 연산자와 함께 사용하도록 만든 함수를 생성자(constructor)라고 합니다. 일반적으로 생성자는 이니셜을 대문자로 표기하여 이름을 지정합니다.[4] 생성자를 new 없이 호출하면 컴파일 시간이나 실행시간에 어떠한 경고도 없어서 알 수 없는 결과를 초래합니다. 그러므로 대문자 표기법을 사용하여 해당 함수가 생성자라고 구분하는 것은 매우 중요합니다.

생성자 함수를 사용하는 스타일은 권장 사항이 아닙니다. 다음 장에서 좀더 나은 대안을 살펴봅니다.

4. 역자 주/ 보통 파스칼 표기법이라고 일컬어지는 표기법입니다.

apply 호출 패턴

자바스크립트는 함수형 객체지향 언어이기 때문에, 함수는 메소드를 가질 수 있습니다.

apply 메소드는 함수를 호출할 때 사용할 인수들의 배열을 받아들입니다. 또한 이 메소드는 this의 값을 선택할 수 있도록 해줍니다. apply 메소드에는 매개변수 두 개가 있습니다. 첫 번째는 this에 묶이게 될 값이며, 두 번째는 매개변수들의 배열입니다.

```
// 숫자 두 개를 가진 배열을 만들고 이를 더함.

var array = [3, 4];
var sum = add.apply(null, array);
// 합은 7

// status라는 속성을 가진 객체를 만듦.

var statusObject = {
    status: 'A-OK'
};

// statusObject는 Quo.prototype을 상속받지 않지만,
// Quo에 있는 get_status 메소드가 statusObject를 대상으로
// 실행되도록 호출할 수 있음.

var status = Quo.prototype.get_status.apply(statusObject);
// status는 'A-OK'
```

04 | 인수 배열(arguments)

함수를 호출할 때 추가적인 매개변수로 arguments라는 배열을 사용할 수 있습니다. 이 배열은 함수를 호출할 때 전달된 모든 인수를 접근할 수 있게 합니다. 여기에는 매개변수 개수보다 더 많이 전달된 인수들도 모두 포함합니다. 이 arguments라는 매개변수는 매개변수의 개수를 정확히 정해놓지 않고, 넘어오는 인수의 개수에 맞춰서 동작하는 함수를 만들 수 있게 합니다.

```
// 여러 작업을 수행하는 함수를 만듦.

// 함수 내부에 있는 sum이라는 변수는
// 외부에 있는 sum 변수에 영향을 미치지 않는 것에 주목.
// 함수는 오로지 내부의 sum에만 영향을 미침.

var sum = function ( ) {
    var i, sum = 0;
    for (i = 0; i < arguments.length; i += 1) {
        sum += arguments[i];
    }
    return sum;
};

document.writeln(sum(4, 8, 15, 16, 23, 42)); // 108
```


이 패턴은 그다지 유용한 패턴은 아닙니다. 6장에는 이와 유사한 메소드를 배열에 추가하는 것을 살펴봅니다.

설계상의 문제로 arguments는 실제 배열은 아닙니다. arguments는 배열 같은 객체입니다. 왜냐하면 arguments는 length라는 속성이 있지만 모든 배열이 가지는 메소드들은 없습니다. 이 장의 마지막에서 이러한 설계상의 오류로 인한 결과를 보게 될 것입니다.

05 | 반환

함수를 호출하면 첫 번째 문장부터 실행해서, 함수의 몸체를 닫는 }를 만나면 끝납니다. 함수가 끝나면 프로그램의 제어가 함수를 호출한 부분으로 반환됩니다.

return 문은 함수의 끝에 도달하기 전에 제어를 반환할 수 있습니다. return 문을 실행하면 함수는 나머지 부분을 실행하지 않고 그 즉시 반환됩니다.

함수는 항상 값을 반환합니다. 반환값이 지정되지 않은 경우에는 undefined가 반환됩니다.

함수를 new라는 전치 연산자와 함께 실행하고 반환값이 객체가 아닌 경우 반환값은 this(새로운 객체)가 됩니다.

06 | 예외

자바스크립트는 예외를 다룰 수 있는 메커니즘을 제공합니다. 예외는 정상적인 프로그램의 흐름을 방해하는 비정상적인 사고입니다(완전히 예측 불가능한 것은 아닙니다). 이러한 사고가 발생하면 프로그램은 예외를 발생합니다.

```
var add = function (a, b) {
    if (typeof a !== 'number' || typeof b !== 'number') {
        throw {
            name: 'TypeError',
            message: 'add needs numbers'
        };
    }
    return a + b;
}
```

throw 문은 함수의 실행을 중단합니다. throw 문은 어떤 예외인지 알 수 있게 해주는 name 속성과 예외에 대해 설명하는 message 속성을 가진 예외 객체를 반환해야 합니다. 물론 이 반환 객체에 필요한 속성이 더 있는 경우 추가할 수 있습니다.

예외 객체는 try 문의 catch 절에 전달됩니다.

```
// 새로운 add 함수를 잘못된 방법으로 호출하는
// try_it 함수 작성

var try_it = function ( ) {
    try {
        add("seven");
    } catch (e) {
        document.writeln(e.name + ': ' + e.message);
    }
}

try_it( );
```

try 블록 내에서 예외가 발생하면, 제어는 catch 블록으로 이동합니다.

try 문은 모든 예외를 포착하는 하나의 catch 블록만을 갖습니다. 만약 예외 상황에 따라 그에 맞게 대처하고 싶은 경우에는 예외 객체의 name 속성을 확인하여 그에 맞게 처리하면 됩니다.

07 | 기본 타입에 기능 추가

자바스크립트는 언어의 기본 타입에 기능을 추가하는 것을 허용합니다. 앞선 3장에서 Object.prototype에 메소드를 추가하여 모든 객체에서 이 메소드를 사용 가능하게 하는 것을 보았습니다. 이러한 작업은 함수, 배열, 문자열, 숫자, 정규 표현식, 불리언에 모두 유효합니다.