

더글라스 크락포드가 정제한  
자바스크립트 언어의 정수!



더글라스 크락포드의

# 자바스크립트 핵심 가이드

더글라스 크락포드 저 | 김명신 역



*JavaScript  
The Good Parts*

더글라스 크락포드의

# 자바스크립트 핵심 가이드



J a v a S c r i p t : T h e G o o d P a r t s

## | 표지 설명 |



이 책의 표지에 있는 곤충은 플레인 타이거 나비입니다(Danaus chrysippus). 아시아 외부 지역에서는 아프리카 황제 나비로도 잘 알려져 있습니다. 이 나비는 밝은 오렌지 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 | 김태현

펴낸곳 | 한빛미디어(주)

주 소 | 서울시 마포구 서교동 480-26 한빛빌딩 2층

전 화 | (02)2128-8722~4(IT전문서팀), (02)336-7114(영업팀)

팩 스 | (02)336-7124

등 록 | 1999년 6월 24일 제10-1779호

초판 발행 | 2008년 9월 29일

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\(주\)](http://www.hanb.co.kr)의 홈페이지나 아래 이메일로 연락 주십시오. 잘못된 책은 구입하신 서점을 통해 교환해 드립니다.

| <http://www.hanb.co.kr>

ask@hanb.co.kr

©HANBIT Media, Inc., 2008.

Authorized translation of the English edition of *JavaScript: The Good Parts* ©2008 O'Reilly & Associates, Inc. This translation is published and sold by permission of O'Reilly & Associates, Inc., the owner of all rights to publish and sell the same.

이 책의 한국어판 저작권은 오라일리사와 독점 계약에 의해 한빛미디어(주)에 있습니다. 저작권법에 의해 한국 내에서 보호를 받는 저작물이므로 무단 전재와 무단 복제를 금합니다.



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 더글拉斯 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(주) 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 접할되었습니다. 자바스크립트는 놀라울 정도로 강력한 언어입니다. 기존의 언어들과 조금은 다른 특성들이 있지만, 그리 어렵지 않게 익힐 수 있습니다.

필자의 목적은 여러분이 자바스크립트식으로 생각할 수 있는 힘을 갖게 돋는 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 있지 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

<b>01 자바스크립트의 좋은 점들</b>	<b>11</b>
01   왜 자바스크립트인가?	13
02   자바스크립트 분석	14
03   예제 테스트를 위한 간단한 준비	17
<b>02 자바스크립트의 좋은 문법들</b>	<b>19</b>
01   공백 whitespace)	20
02   이름 Names)	21
03   숫자 Numbers)	23
04   문자열 Strings)	24
05   문장 Statements)	26
06   표현식 Expressions)	33
07   리터럴 Literals)	36
08   함수 Functions)	37
<b>03 객체</b>	<b>39</b>
01   객체 리터럴	40
02   속성값 읽기	41
03   속성값의 생성	42

04   참조	43
05   프로토타입(Prototype)	43
06   리플렉션(reflection)	45
07   열거(Enumeration)	46
08   삭제	47
09   최소한의 전역변수 사용	47

## 04 함수 49

01   함수 객체	49
02   함수 리터럴	50
03   호출	51
04   인수 배열(arguments)	57
05   반환	58
06   예외	58
07   기본 타입에 기능 추가	59
08   재귀적 호출	62
09   유효범위(Scope)	65
10   클로저(closure)	66
11   콜백	71
12   모듈	72
13   연속 호출(Cascade)	75
14   커링(Curry)	76
15   메모이제이션(memoization)	78

## 05 상속 81

01   의사 클래스 방식(Pseudoclassical)	82
02   객체를 기술하는 객체(Object Specifiers)	87
03   프로토타입 방식	88
04   함수를 사용한 방식	90
05   클래스 구성을 위한 부속품	96

<b>06 배열</b>	<b>99</b>
01   배열 리터럴	100
02   length 속성	101
03   삭제	103
04   열거	104
05   객체와 배열의 혼동	104
06   배열의 메소드	106
07   배열의 크기와 차원	108
<b>07 정규 표현식</b>	<b>111</b>
01   예제	113
02   정규 표현식 객체 생성	120
03   구성요소	122
<b>08 메소드</b>	<b>131</b>
<b>09 스타일</b>	<b>157</b>
<b>10 아름다운 속성에 대한 단상</b>	<b>163</b>
부록 A   나쁘지만 사용해야 하는 부분들(Awful parts)	167
부록 B   나쁜 점들	181
부록 C   JSLint	193
부록 D   구문 다이어그램	211
부록 E   JSON	223
찾아보기	237





## 자바스크립트의 좋은 점들

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

– 윌리엄 셰익스피어, 원저의 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, 나쁜 점은 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. DOM은 어떠한 언어로 다루든지 용이하지 않습니다. DOM은 서툴게 정의됐으며 그에 따라 그 구현도 제각각입니다. 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. 그러나 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)<sup>1</sup>입니다. 또한 주류가 된 첫 번째 람다(lambda)<sup>2</sup> 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. 역사 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\\_object](http://en.wikipedia.org/wiki/First-class_object)를 참고하세요.

2. 역사 주/ 람다는 Alonzo Church와 Stephen Cole Kleene의 람다 대수(Lambda calculus, [http://en.wikipedia.org/wiki/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/Lambda>를 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 속성을 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 노력을 분명 보상 받을 것입니다.

자바스크립트는 몇몇 핵심 개념들 때문에 더 많이 비난을 받고 있습니다. 좀 다른 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 Edition」(<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 보통 자바스크립트 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 어찌

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 편찮다는 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.



## 자바스크립트의 좋은 문법들

내가 잘 아는 것이군. 오래 전 문법책 예문에서 읽은 시구로군.

—윌리엄 셰익스피어, 타이터스 앤드로니커스



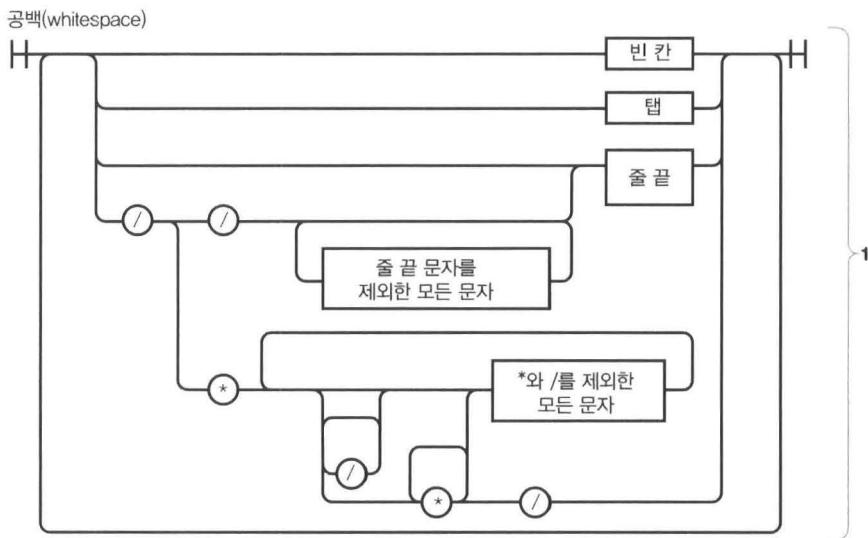
이번 장에서는 자바스크립트의 좋은 점들(good parts)에 해당하는 문법을 살펴봅니다. 이 문법들을 살펴봄으로써 언어가 어떻게 구성되었는지 빠르게 확인할 수 있습니다. 각 문법은 철도 다이어그램으로 설명합니다.

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 whitespace을 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.

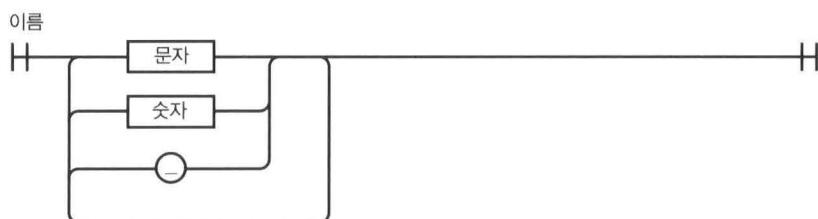
자바스크립트에서는 /\* \*/ 형태의 블록 주석과 // 형태의 한 줄 주석을 사용할 수 있습니다. 주석은 프로그램의 가독성을 높이기 위해 충분히 사용되는 것이 좋습니다. 주석을 달 때는 항상 코드에 대해 정확히 설명해야 합니다. 쓸모 없는 주석은 없느니만 못합니다.

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 /a*/.match(5);
*/
```

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

## 02 | 이름(Names)



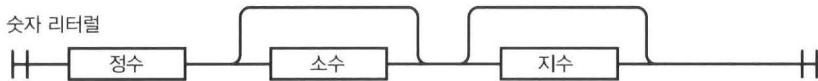
이름(name)은 하나의 문자나 그 뒤를 이어서 하나 이상의 문자, 숫자, \_가 붙는 문자열로 문장, 변수, 매개변수, 속성명, 연산자, 라벨 등에 사용합니다. 다음에 나오는 예약어들은 이름(name)이 될 수 없습니다.

```
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
```

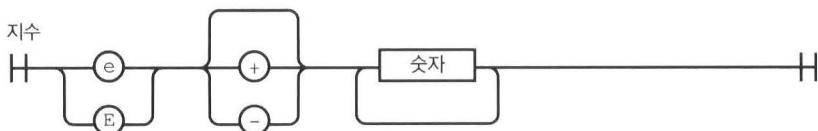
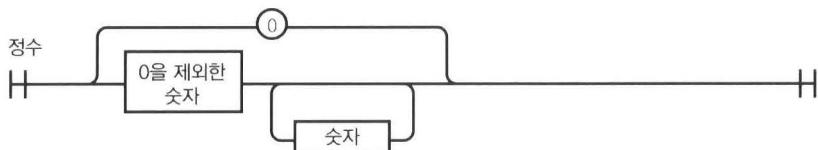
(o) 목록에는 예약어에 꼭 포함할 것 같은 undefined, NaN, Infinity 등을 없습니다.)

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 소수 점 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.

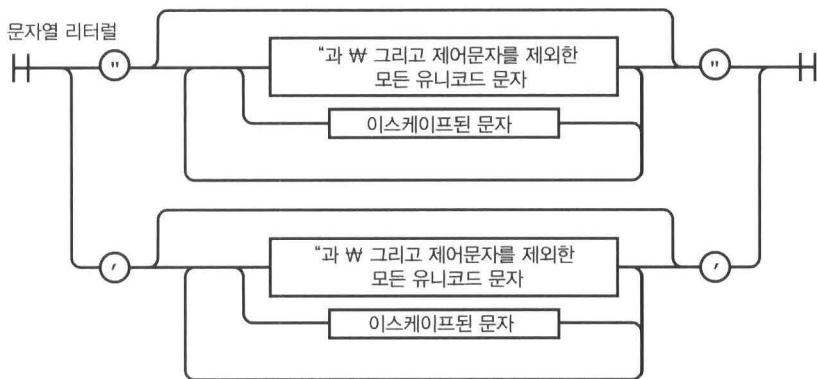
음수는 수 앞에 -를 붙이면 됩니다.

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

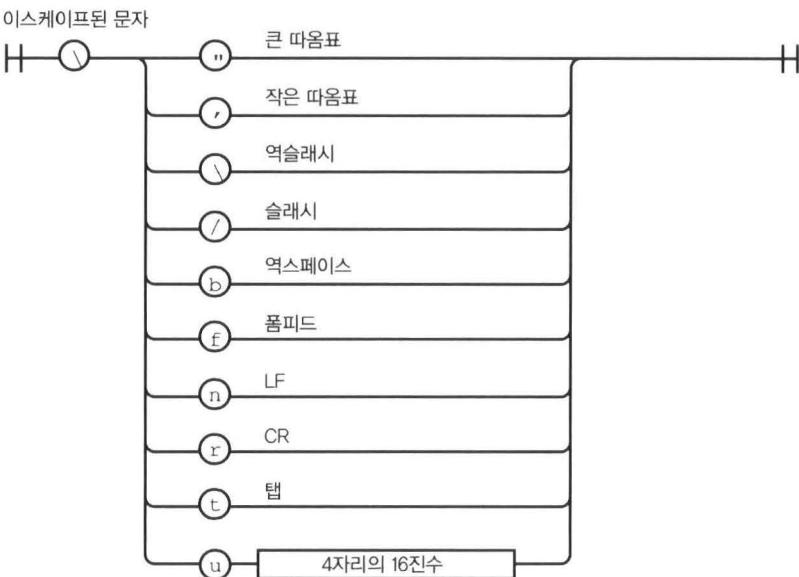
`1.79769313486231570e+308` 보다 큰 값은 `Infinity`로 나타납니다.

숫자는 메소드가 있습니다(8장 참조). 자바스크립트의 `Math`라는 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.

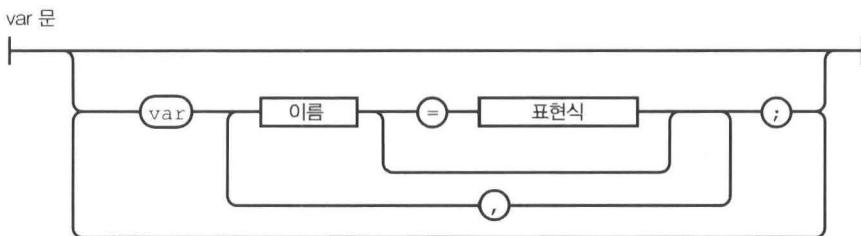
문자열은 변하지 않습니다(immutable). 일단 문자열이 한번 만들어지면, 이 문자열은 결코 변하지 않습니다. 하지만 여러 문자열을 + 연산자로 연결하여 새로운 문자열을 만들 수 있습니다. 분리된 문자열이 + 연산자로 연결되든 그냥 문자열 하나든 문자들의 순서가 같으면 같은 문자열입니다. 그래서 다음의 예는 참(true)입니다.

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

문자열은 메소드가 있습니다(8장 참조).

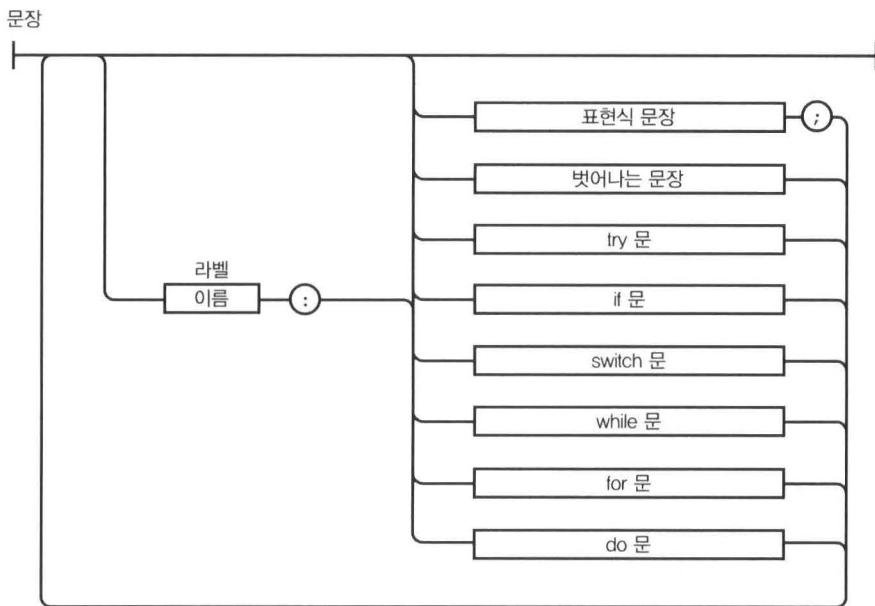
```
'cat'.toUpperCase() === 'CAT'
```

## 05 | 문장(Statements)

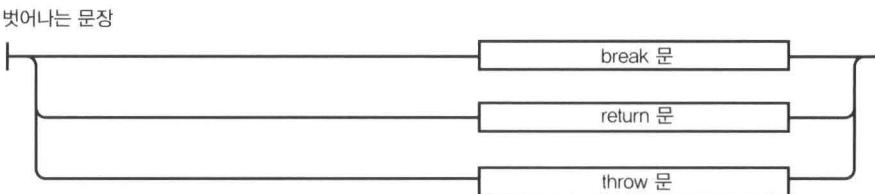


하나의 컴파일 단위에는 실행을 위한 문장들이 포함돼 있습니다. 웹 브라우저에서 각각의 `<script>` 태그는 컴파일되어 즉시 실행되는 하나의 컴파일 단위입니다. 링커(linker)가 없기 때문에 자바스크립트는 모든 문장을 공통적인 전역 이름 공간(namespace)에 한 데 몰아 넣습니다. 전역변수에 관한 좀더 자세한 부분은 부록 A에서 설명합니다.

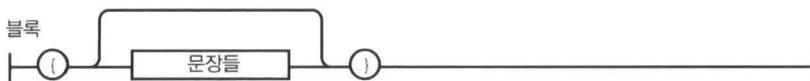
`var` 문은 함수 내부에서 사용될 때 함수의 private 변수를 정의합니다.



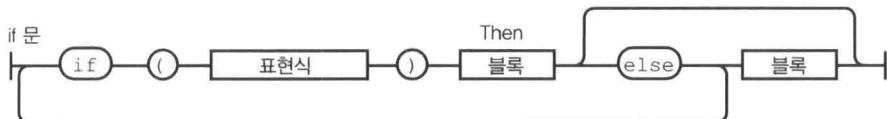
switch 문, while 문, for 문, do 문에는 break 문에서 사용할 수 있는 라벨을 선택적으로 지정할 수 있습니다.



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.

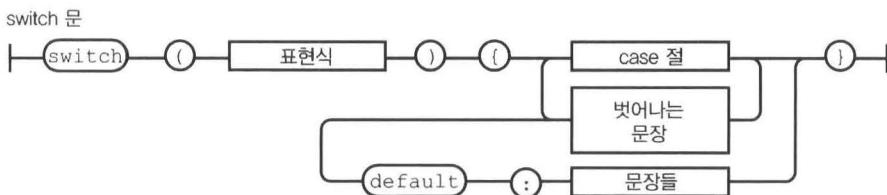


if 문은 표현식의 값에 따라 프로그램의 흐름을 변경합니다. then 블록은 표현식이 참(true)일 때 실행하며, 표현식이 거짓인 경우에는 else 블록을 실행합니다(물론 else 블록은 선택적입니다).

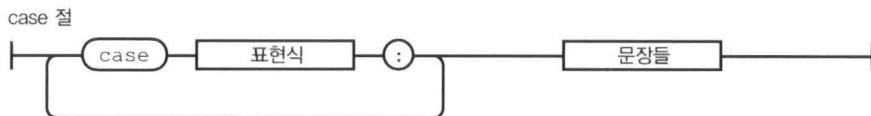
다음은 거짓에 해당하는 값들입니다.

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

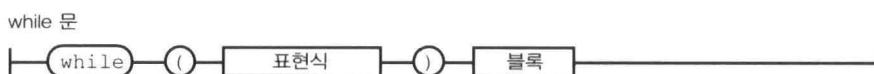
이 외의 모든 값은 참입니다. 예를 들어 true, 문자열 'false', 모든 객체 등은 모두 참입니다.



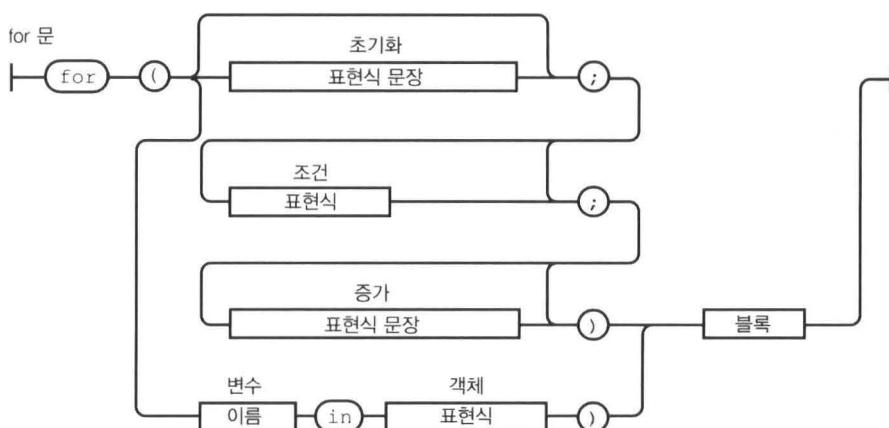
switch 문은 다중 분기를 수행합니다. 이 문장은 표현식과 모든 case 문의 표현식이 같은지를 비교합니다. 표현식의 결과값은 숫자일 수도 있고 문자열일 수도 있습니다. 일치하는 표현식을 찾으면 해당 case 절에 있는 문장들을 실행합니다. 만약 일치하는 표현식을 찾지 못하는 경우에는 default 절의 문장들을 실행합니다 (default는 선택사항입니다).



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.

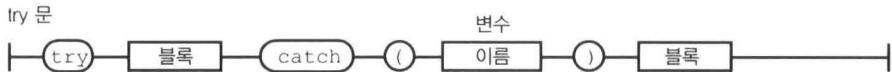
이 for in 형식은 보통 다음과 같이 `object.hasOwnProperty(변수)` 메소드로 속성 이름이 실제로 객체의 속성인지 아니면 프로토타입 체인(prototype chain) 상에 있는 것인지를 확인하는 것이 필요합니다.

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

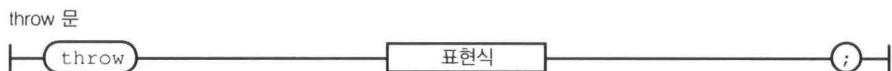
do 문



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

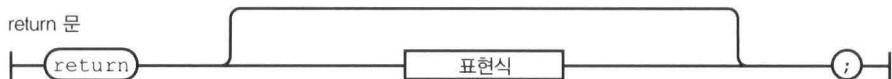


try 문은 블록을 실행하면서 블록 내에서 발생하는 예외 상황을 포착합니다. catch 절은 예외 객체를 받는 새로운 변수를 정의합니다.



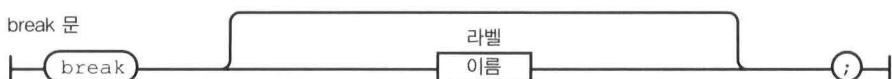
throw 문은 예외를 발생합니다. 만약 throw 문이 try 블록 안에 있으면 실행의 제어는 catch 절로 이동합니다. 만약 throw 문이 일반적인 함수 내에 있다면 함수 호출은 중단되고 함수를 호출한 try 문의 catch 절로 실행 흐름이 이동합니다.

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.

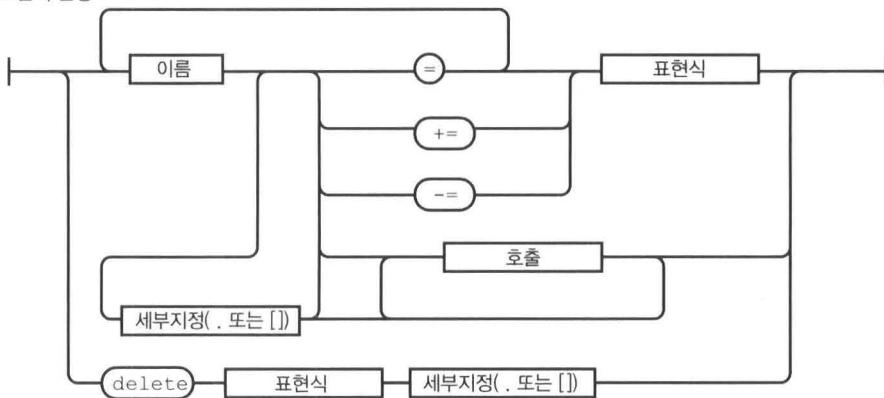
자바스크립트는 return과 표현식 부분 사이에 줄 바꿈을 허용하지 않습니다.



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.

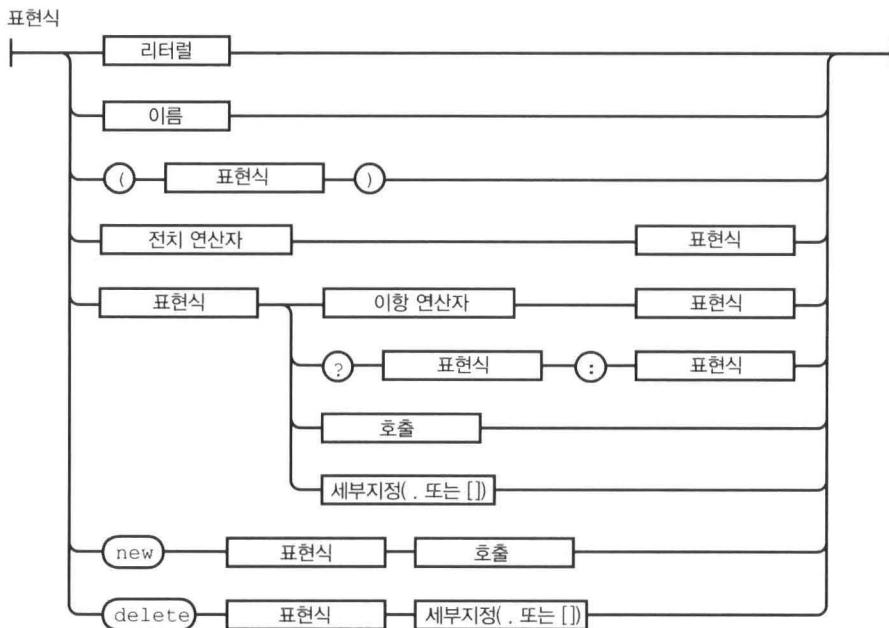
자바스크립트는 break와 라벨 사이에 줄 바꿈을 허용하지 않습니다.

표현식 문장



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 [] )

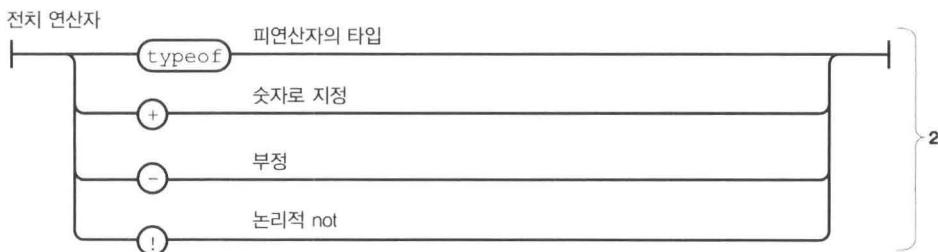
? 삼항 연산자는 3개의 피연산자를 취합니다. 첫 번째 피연산자가 참이면, 두 번째 피연산자가 값이 되지만, 첫 번째 피연산자가 거짓인 경우에는 세 번째 피연산자가 값이 됩니다.

[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

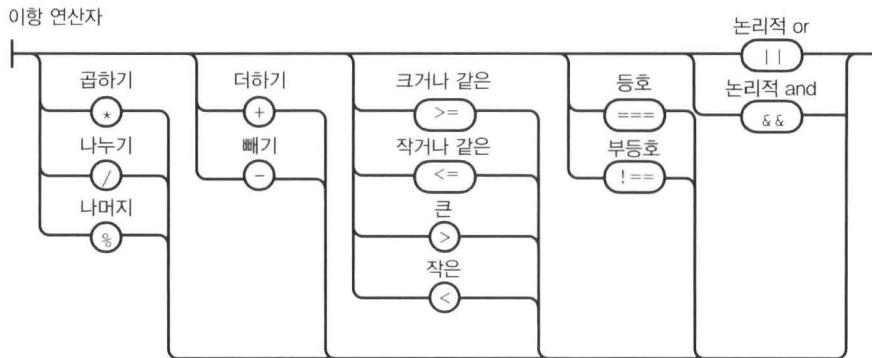
. [] ()	세부지정이나 호출
delete new typeof + - !	단항 연산자
* / %	곱하기, 나누기, 나머지
+ -	더하기/연결, 빼기
)= (<= > <	같지 않음 비교
== !=	동등
&&	논리적 and
	논리적 or
? :	삼항



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.

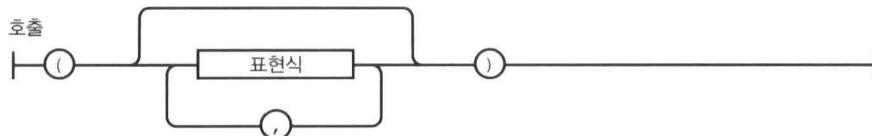


+ 연산자는 수를 더하거나 문자열을 연결합니다. 더하기 연산을 원하는 경우에는 반드시 피연산자 두 개가 모두 숫자여야 합니다.

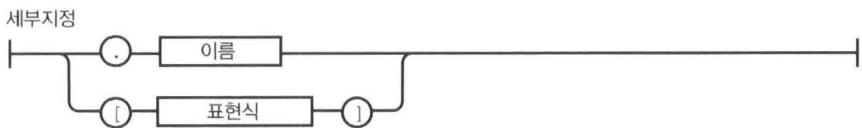
/ 연산자는 피연산자 두 개가 모두 정수형이어도 결과값이 실수일 수 있습니다.

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

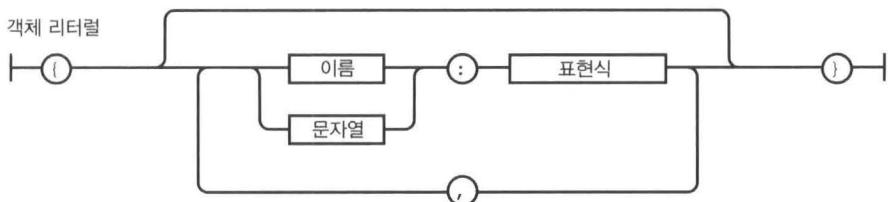
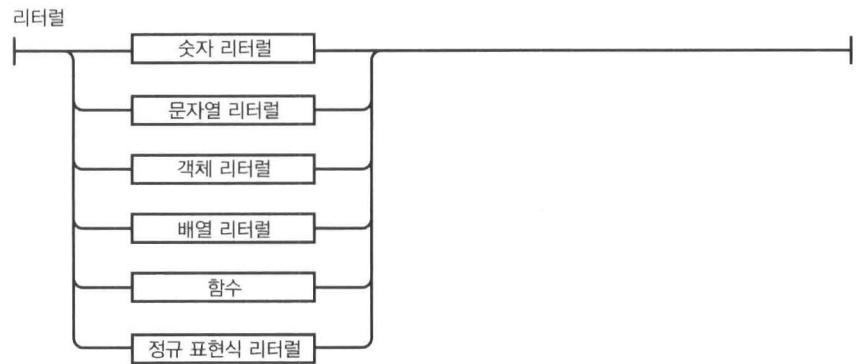
|| 연산자는 &&와 반대로 첫 번째 피연산자가 참이면 이 값을 취하고 그렇지 않으면 두 번째 피연산자를 결과값으로 취합니다.



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



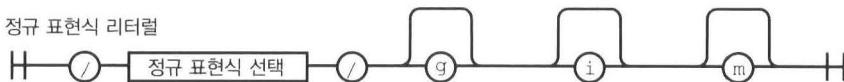
## 07 | 리터럴(Literals)



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.

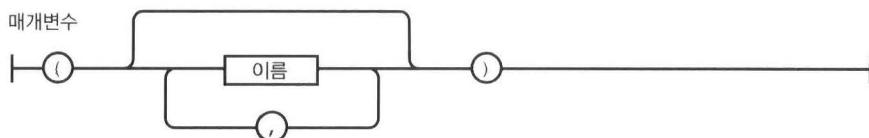


배열 리터럴은 새로운 배열을 생성할 때 편리한 표기법입니다. 배열은 6장에서 보다 자세하게 살펴봅니다.



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 매개 변수 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 모우고 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.<sup>1</sup>

---

1. 역사 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 생성에 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.<sup>2</sup>

## 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. 역사 주/ 예제 코드에 나오는 `stooge`와 `another_stooge`로 프로토타입 개념을 설명하면 다음과 같습니다. `another_stooge`의 프로토타입은 `stooge`고 `stooge`는 `another_stooge`의 프로토타입 객체가 됩니다. 그리고 `another_stooge`가 자신의 프로토타입인 `stooge`에 연결돼 있고 또 `stooge`는 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 hasOwnProperty 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.prototype`에 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 결과는 구분됩니다. 이렇게 복잡하게 얹힌 구조는 다음 장에서 살펴봅니다.<sup>1</sup>

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. 역사 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 `undefined`로 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.<sup>2</sup>

## 03 | 호출

함수를 호출하면 현재 함수의 실행을 잠시 중단하고 제어를 매개변수와 함께 호출한 함수로 넘깁니다. 모든 함수는 명시되어 있는 매개변수에 더해서 `this`와 `arguments`라는 추가적인 매개변수 두 개를 받게 됩니다. `this`라는 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 `this`라는 추가적인 매개변수를 다르게 초기화합니다.

---

2. 역사 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 돋지 못한다는 것입니다. 왜냐하면, 내부 함수는 메소드가 객체 접근을 위해 사용하는 this에, 자신의 this를 바인딩하지 않고 영뚱한 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

```



## 생성자 호출 패턴

자바스크립트는 프로토타입에 의해서 상속이 이루어지는 언어입니다. 이 말은 객체가 자신의 속성을 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 프로토타입적 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. 역사 주/ 예제의 경우 위의 예부터 계속 이어져야지만 6이라는 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.

`new`라는 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
```

`new`라는 전치 연산자와 함께 사용하도록 만든 함수를 생성자(constructor)라고 합니다. 일반적으로 생성자는 이니셜을 대문자로 표기하여 이름을 지정합니다.<sup>4</sup> 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 예제는 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 해 주는 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.

예를 들어 다음과 같이 method라는 메소드를 Function.prototype에 추가하면 이후 모든 함수에서 이 메소드를 사용할 수 있습니다.

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

이와 같이 method라는 메소드를 Function.prototype에 추가함으로써 앞으로는 Function.prototype에 메소드를 추가할 때 prototype이라는 속성 이름을 사용할 필요가 없습니다. 이로 인해 코드를 다소 보기 안 좋게 하는 부분(.prototype)이 사라집니다. (위 코드처럼 추가하던 것을 아래의 코드처럼 .prototype 부분 없이 깔끔하게 사용할 수 있습니다.)

자바스크립트에는 따로 구분된 정수형이 없어서 때때로 숫자형에서 정수 부분만 추출해야 하는 경우가 생깁니다. 그런데 이러한 작업을 위해 자바스크립트가 제공하는 방법은 용이하지 않습니다. 이러한 문제를 다음의 예처럼 Number.prototype에 integer라는 메소드를 추가해서 해결할 수 있습니다. 이 메소드는 숫자의 부호에 따라 Math.ceil이나 Math.floor를 사용합니다.

```
Number.method('integer', function () {  
    return Math[this < 0 ? 'ceiling' : 'floor'](this);  
});  
  
document.writeln((-10 / 3).integer()); // -3
```

자바스크립트에는 문자열의 양 끝에 있는 빈 칸을 지우는 메소드가 없습니다. 이러한 부실함을 다음과 같이 간단하게 보완할 수 있습니다.

```
String.method('trim', function () {
    return this.replace(/^\\s+|\\s+$/g, '');
});

document.writeln('"' + "    neat    ".trim() + '"');
```

trim() 메소드는 정규 표현식을 사용하는데, 정규 표현식은 7장에서 자세히 살펴봅니다.

이러한 방법으로 기본적인 타입에 기능을 추가함으로써 언어의 능력을 배가시킬 수 있습니다. 자바스크립트의 프로토타입에 의한 상속이라는 동적인 본성 덕분에 새로운 메소드를 추가하면 관련된 값들에는 바로 새로운 메소드들이 추가됩니다. 이러한 특성은 해당 값이 새로운 메소드가 추가되기 전에 생성됐더라도 관계 없이 적용됩니다.

기본 타입의 프로토타입은 public 구조입니다. 그러므로 라이브러리들을 섞어서 사용할 때는 주의할 필요가 있습니다. 한 가지 방어적인 방법은 존재하지 않는 메소드만 추가하는 것입니다.

```
// 조건에 따라 메소드를 추가.
```

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

## 08 | 재귀적 호출

재귀 함수는 직접 또는 간접적으로 자신을 호출하는 함수입니다. 재귀적 호출은 어떤 문제가 유사한 하위 문제로 나뉘어지고 각각의 문제를 같은 해결 방법으로 처리할 수 있을 때 사용할 수 있는 강력한 프로그래밍 기법입니다. 일반적으로 재귀 함수는 하위 문제를 처리하기 위해 자신을 호출합니다.

하노이의 탑은 유명한 퍼즐입니다. 이 퍼즐에는 3개의 기둥과 가운데 구멍이 있는 다양한 지름의 원반이 있습니다. 먼저 시작 기둥에 원반들을 지름이 큰 것에서부터 작은 것으로 차례로 쌓습니다. 목표는 한 번에 원반 하나를 다른 기둥으로 옮기면서 최종적으로 목적 기둥에 원래의 순서대로 쌓는 것입니다. 여기서 한가지 규칙은 절대로 큰 원반이 작은 원반 위에 쌓여서는 안 된다는 것입니다. 이 퍼즐은 재귀적 호출을 위한 전형적인 문제입니다.

```
var hanoi = function (disc, src, aux, dst) {
    if (disc > 0) {
        hanoi(disc - 1, src, dst, aux);
        document.writeln('Move disc ' + disc +
            ' from ' + src + ' to ' + dst);
        hanoi(disc - 1, aux, src, dst);
    }
};

hanoi(3, 'Src', 'Aux', 'Dst');
```

이 프로그램을 실행하면 3개의 원반에 대한 다음과 같은 해결방법을 볼 수 있습니다.

```
Move disc 1 from Src to Dst
Move disc 2 from Src to Aux
Move disc 1 from Dst to Aux
Move disc 3 from Src to Dst
```

```
Move disc 1 from Aux to Src
Move disc 2 from Aux to Dst
Move disc 1 from Src to Dst
```

Hanoi 함수는 필요한 경우 보조 기둥을 사용하며 원반들을 목적지 기둥으로 옮깁니다. 이 함수는 전체적인 문제 하나를 세부 문제 세 개로 분리합니다. 먼저 위쪽에 있는 원반들을 보조 기둥으로 옮겨서 바닥에 있는 원반을 드러나게 합니다. 이렇게 하면 바닥에 있는 원반을 목적지 기둥으로 옮길 수 있습니다. 마지막으로 보조 기둥에 있는 원반을 목적지 기둥으로 옮깁니다. 보조 기둥에 있는 남은 원반들의 이동은 자신을 재귀적으로 호출하는 방식으로 수행됩니다.

hanoi 함수는 옮겨야 할 원반의 수와 사용할 수 있는 기둥 세 개를 넘겨 받습니다. hanoi는 자신을 재귀적으로 호출할 때 현재 작업하고 있는 원반의 위에 있는 원반을 처리합니다. 이러한 작업을 반복하면 결국 원반이 없는 상태에서 함수가 호출됩니다. 이런 경우 아무 것도 하지 않게 되는데 이렇게 함으로써 재귀적 호출이 무한대로 일어나지 않게 됩니다.

재귀 함수는 웹 브라우저의 DOM(Document Object Model) 같은 트리 구조를 다루는데 매우 효과적입니다. 즉 각각의 재귀적 호출이 트리 구조의 항목 하나에 대해 작동하면 효율적으로 트리 구조를 다룰 수 있습니다.

```
// 주어진 노드부터 HTML 소스 순으로 DOM 트리의
// 모든 노드를 방문하는 walk_the_DOM 함수 정의.
// 이 함수는 차례로 각각의 노드를 넘기면서 함수를 호출.
// walk_the_DOM은 각각의 자식 노드들을 처리하기 위해서
// 자신을 호출함.
```

```
var walk_the_DOM = function walk(node, func) {
    func(node);
    node = node.firstChild;
    while (node) {
        walk(node, func);
        node = node.nextSibling;
```

```

        }
    };

    // getElementsByAttribute 함수 정의.
    // 이 함수는 어트리뷰트 이름(att)과 일치하는 값(value, 이 값은 넘기지 않아도
    // 되는 옵션임)을 인수로 받음.
    // 이 함수는 노드에서 어트리뷰트 이름을 찾는 함수를 전달하면서
    // walk_the_DOM을 호출.
    // 일치하는 노드는 results 배열에 저장됨.

var getElementsByAttribute = function (att, value) {
    var results = [];

    walk_the_DOM(document.body, function (node) {
        var actual = node.nodeType === 1 && node.getAttribute(att);
        if (typeof actual === 'string' &&
            (actual === value || typeof value !== 'string')) {
            results.push(node);
        }
    });
    return results;
};

```

몇몇 언어에서는 꼬리 재귀(tail recursion) 최적화를 제공합니다. 꼬리 재귀 최적화라는 것은 함수가 자신을 재귀적으로 호출하는 것을 반환하는 방법으로 진행되는 재귀적 호출(꼬리 재귀)일 경우 이를 개선하여 속도를 매우 빠르게 향상시키는 반복 실행으로 대체하는 것입니다(다음의 예 참조). 불행하게도 현재 자바스크립트는 꼬리 재귀 최적화를 제공하지 않습니다. 자신을 매우 깊은 단계까지 호출하는 함수는 반환 스택의 과다 사용으로 제대로 실행되지 않을 수 있습니다.

```

// 꼬리 재귀를 하는 계승(factorial) 함수를 만듭니다.
// 호출 자체의 결과를 반환하기 때문에 꼬리 재귀입니다.

// 현재 자바스크립트는 이러한 유형에 대해 최적화를 제공하지 않습니다.

```

```
var factorial = function factorial(i, a) {
    a = a || 1;
    if (i < 2) {
        return a;
    }
    return factorial(i - 1, a * i);
};

document.writeln(factorial(4));      // 24
```

## 09 | 유효범위(Scope)

프로그래밍 언어에서 유효범위는 변수와 매개변수의 접근성과 생존 기간을 제어합니다. 유효범위는 이름들이 충돌하는 문제를 덜어주고 자동으로 메모리를 관리하기 때문에 프로그래머에게는 중요한 개념입니다.

```
var foo = function () {
    var a = 3, b = 5;

    var bar = function () {
        var b = 7, c = 11;

        // 이 시점에서 a는 3, b는 7, c는 11

        a += b + c;
    }

    // 이 시점에서 a는 21, b는 7, c는 11

    bar();
}

// 이 시점에서 a는 3, b는 5, c는 정의되지 않음.

// 이 시점에서 a는 21, b는 5

};
```

C 언어 유형의 구문을 가진 모든 언어는 블록 유효범위가 있습니다. 블록(중괄호로 묶인 문장들의 집합) 내에서 정의된 모든 변수는 블록의 바깥쪽에서는 접근할 수 없습니다. 블록 내에서 정의된 변수는 블록의 실행이 끝나면 해제됩니다. 이러한 구조는 좋은 구조입니다.

자바스크립트의 블록 구문은 마치 블록 유효범위를 지원하는 것처럼 보이지만 불행히도 블록 유효범위가 없습니다. 이러한 혼란은 오류의 원인이 될 여지가 충분합니다.

자바스크립트는 함수 유효범위가 있습니다. 즉 함수 내에서 정의된 매개변수와 변수는 함수 외부에서는 유효하지 않습니다. 반면에 이렇게 내부에서 정의된 변수는 함수 어느 곳에서도 접근할 수 있습니다.

오늘날 대부분의 언어에서는 변수를 가능한 늦게, 즉 처음 사용하기 바로 전에 선언해서 사용할 것을 권하고 있습니다. 하지만 자바스크립트에서는 블록 유효범위를 지원하지 않기 때문에 이러한 권고가 적용되지 않습니다. 대신에 자바스크립트에서는 함수에서 사용하는 모든 변수를 함수 첫 부분에서 선언하는 것이 최선의 방법입니다.

## 10 | 클로저(closure)

유효범위에 관한 좋은 소식 하나는 내부 함수에서 자신을 포함하고 있는 외부 함수의 매개변수와 변수들을 접근할 수 있다는 것입니다(this와 arguments는 예외입니다). 이러한 특성은 매우 유용합니다.

내부 함수에서 외부 함수의 변수에 접근할 수 있는 예는 앞서 재귀적 호출에서 살펴봤던 getElementsByTagName 함수에서 볼 수 있습니다. 이 함수에는 results라는 변수가 선언돼 있는데 이 results라는 변수를 walk\_the\_DOM의 인수로 넘긴 (내부) 함수에서 접근하고 있습니다.

이러한 특성과 관련하여 더 흥미로운 경우는 외부 함수보다 내부 함수가 더 오래 유지될 때입니다.

03. 호출 절에서 value라는 속성과 increment라는 메소드를 가진 myObject를 살펴봤습니다. 이제 myObject 객체에서 허락되지 않은 경우에는 value 속성의 값을 변경할 수 없게 하고 싶다고 가정해 보겠습니다.

myObject를 객체 리터럴로 초기화하는 대신에 다음에 나오는 코드처럼 객체 리터럴을 반환하는 함수를 호출하여 초기화합니다. 이렇게 하면 increment와 getValue를 통해 value라는 변수에 접근할 수 있지만 함수 유효범위 때문에 프로그램의 나머지 부분에서는 접근할 수가 없습니다.

```
var myObject = function () {
    var value = 0;

    return {
        increment: function (inc) {
            value += typeof inc === 'number' ? inc : 1;
        },
        getValue: function () {
            return value;
        }
    };
}();
```

코드를 잘 살펴보면 myObject에 함수를 할당한 것이 아니라 함수를 호출한 결과를 할당하고 있습니다. 맨 마지막 줄에 있는 ()를 주목할 필요가 있습니다. 함수는 메소드 두 개를 가진 객체를 반환하며 이 두 메소드는 계속해서 value라는 변수에 접근할 수 있습니다.

03절의 생성자 호출 패턴에서 살펴봤던 Quo 생성자는 status라는 속성과 get\_status라는 메소드를 가진 객체를 생성합니다. 하지만 status라는 변수를 바로 접근할 수 있기 때문에 getter 역할을 하는 get\_status는 별 쓸모가 없어 보입니다. get\_status가 쓸모가 있으려면 status 속성이 private이어야 할 것입니다. 그러면 이제 그렇게 되도록 quo라는 함수를 정의해 보겠습니다.

```

// quo라는 함수를 생성.
// 이 함수는 get_status라는 메소드와
// status라는 private 속성을 가진 객체를 반환.

var quo = function (status) {
    return {
        get_status: function () {
            return status;
        }
    };
};

// quo의 인스턴스를 생성.

var myQuo = quo("amazed");

document.writeln(myQuo.get_status());

```

quo 함수는 new 키워드 없이 사용하게 설계됐습니다. 그래서 이름을 대문자로 표기하지 않았습니다. quo를 호출하면 get\_status 메소드가 있는 객체를 반환합니다. 이 객체에 대한 참조는 myQuo에 저장됩니다. get\_status 메소드는 quo가 이미 반환된 뒤에도 quo의 status에 계속해서 접근할 수 있는 권한을 가지게 됩니다. get\_status는 status 매개변수의 복사본에 접근할 수 있는 권한을 갖는 것이 아니라 매개변수 그 자체에 대한 접근 권한을 갖습니다. 이러한 것이 가능한 것은 함수가 자신이 생성된 함수, 즉 자신을 내포하는 함수의 문맥(context)에 접근할 수 있기 때문입니다. 이러한 것을 클로저(closure)라고 부릅니다.

좀더 유용한 예제를 살펴보겠습니다.

```

// DOM 노드의 색을 노란색으로 지정하고 흰색으로 사라지게 하는
// 함수 정의.

var fade = function (node) {
    var level = 1;
    var step = function () {
        var hex = level.toString(16);

```

```

        node.style.backgroundColor = '#FFFF' + hex + hex;
        if (level < 15) {
            level += 1;
            setTimeout(step, 100);
        }
    };
    setTimeout(step, 100);
};

fade(document.body);

```

document.body(HTML <body> 태그에 의해서 만들어지는 노드)를 넘기면서 fade를 호출합니다. fade는 level의 값을 1로 설정합니다. 그리고 step이라는 함수를 정의하고, setTimeout을 호출하여 이 함수를 100밀리 초 후에 실행하게 합니다. 여기 까지 수행을 한 후 fade는 종료합니다.

약 10분의 1초 뒤에 step 함수가 호출됩니다. 이 함수는 fade의 level 값을 16진수로 변환한 후 이를 이용하여 fade의 매개변수인 node의 배경색을 변경합니다. 그리고 나서 fade의 level 값을 살펴본 후 아직 배경색이 흰색이 되지 않았으면 level의 값을 증가시킨 후, setTimeout을 사용하여 같은 작업을 반복하게 합니다.

이제 다시 step 함수가 호출되면 이번에는 fade 내의 변수 level의 값이 2입니다. fade 함수는 이미 반환됐지만 함수 안의 변수는 이를 필요로 하는 내부 함수가 하나 이상 존재하는 경우 계속 유지됩니다.

내부 함수가 외부 함수에 있는 변수의 복사본이 아니라 실제 변수에 접근한다는 것을 이해해야 합니다. 그렇지 않으면 다음과 같은 문제가 발생할 수 있습니다.

```

// 나쁜 예제.

// 잘못된 방법으로 노드 배열에 이벤트 핸들러 함수를 할당하는 함수 정의.
// 노드를 클릭하면 해당 노드가 몇 번째 노드인지지를 경고창으로 알려주는 것이
// 함수의 목적.
// 하지만 항상 전체 노드의 수만을 보여줌.

```

```
var add_the_handlers = function (nodes) {
    var i;
    for (i = 0; i < nodes.length; i += 1) {
        nodes[i].onclick = function (e) {
            alert(i);
        };
    }
};

// 나쁜 예제 끝.
```

add\_the\_handlers 함수는 각각의 핸들러에 유일한 번호(i)를 전달하도록 고안됐습니다. 하지만 이러한 의도대로 동작하지 않는데 그 이유는 핸들러 함수가 받는 i가 함수가 만들어지는 시점의 i가 아니라 그냥 변수 i에 연결되기 때문입니다.

```
// 더 나은 예제.

// 올바른 방법으로 노드 배열에 이벤트 핸들러 함수를 할당하는 함수 정의.
// 노드를 클릭하면 해당 노드가 몇 번째 노드인지를 경고창으로 알려줌.

var add_the_handlers = function (nodes) {
    var i;
    for (i = 0; i < nodes.length; i += 1) {
        nodes[i].onclick = function (i) {
            return function (e) {
                alert(i);
            };
        }(i);
    }
};
```

이제 onclick에 함수를 할당하는 대신에 새로 함수를 정의하고 여기에 i를 넘기면서 곧바로 실행시켰습니다. 실행된 함수는 add\_the\_handlers에 정의된 i가 아니라 넘겨받은 i의 값을 이벤트 핸들러 함수에 연결하여 반환합니다. 이 반환되는 이벤트 핸들러 함수는 onclick에 할당합니다.

## 11 | 콜백

함수는 비연속적인 이벤트를 다루는 것을 좀더 쉽게 할 수 있는 방법을 제공합니다. 예를 들어 사용자와 상호작용으로 시작해서 서버로 요청을 하고 마지막으로 요청에 대한 응답을 보여주는 일련의 작업 흐름이 있다고 가정해 보겠습니다. 이러한 작업을 처리하는 가장 고지식한 방법은 다음과 같을 것입니다.

```
request = prepare_the_request();
response = send_request_synchronously(request);
display(response);
```

이러한 방법으로 작업을 해결할 때의 문제는 동기화된 요청을 하기 때문에 서버로부터 응답이 올 때까지 클라이언트는 꼼짝없이 멈춰서 기다려야 한다는 것입니다. 만약 네트워크나 서버가 느리다면 이 애플리케이션은 응답성에 있어서 이해할 수 없을 만큼 최악일 것입니다.

이런 작업을 처리하는 좋은 방법은 서버로 요청을 비동기식으로 하고 서버의 응답이 왔을 때 호출되는 콜백 함수를 제공하는 것입니다. 비동기식 함수는 서버의 응답을 기다리지 않고 그 즉시 반환되기 때문에 클라이언트는 멈춤 상태로 빠지지 않습니다.

```
request = prepare_the_request();
send_request_asynchronously(request, function (response) {
    display(response);
});
```

`send_request_asynchronously` 함수에 함수를 매개변수로 전달하여 서버로부터 응답이 왔을 때 호출되게 합니다.<sup>5</sup>

## 12 | 모듈

함수와 클로저를 사용해서 모듈을 만들 수 있습니다. 모듈은 내부의 상태나 구현 내용은 숨기고 인터페이스만 제공하는 함수나 객체입니다. 모듈을 만들기 위해서 함수를 사용하면 전역변수 사용을 거의 대부분 제거할 수 있기 때문에 결국 자바스크립트의 최대 약점 중 하나를 보완할 수 있습니다.

예를 들어 `String` 객체에 `deentityify` 메소드를 추가한다고 가정해 보겠습니다. 이 메소드는 문자열에서 HTML 엔티티들을 찾고 이들을 그에 상응하는 문자로 대체하는 기능을 합니다. `deentityify`는 엔티티의 이름과 상응하는 문자들이 담긴 객체를 사용해야 합니다. 그런데 이 객체를 어디에 간직해야 할까요? 간단히 생각해서 이 객체를 전역변수로 지정할 수 있습니다. 하지만 전역변수는 사용해서는 안 되는 나쁜 것입니다. 또 다른 방법으로 이 객체를 함수 자체 내에 정의할 수도 있습니다. 하지만 이 역시도 함수가 호출될 때마다 매번 객체 리터럴을 객체화(evaluate)하는 실행시간 비용(runtime cost) 부담이 있습니다. 이상적인 해결 방법은 이 객체를 클로저에 두고 이 객체에 HTML 엔티티를 추가할 수 있는 메소드를 따로 두는 것입니다.

```
String.method('deentityify', function () {  
    // 엔티티 테이블. 엔티티 이름을 문자에 대응시킴.
```

5. 역사 주/ 저자의 의도는 콜백 함수라는 함수 호출의 한 형태를 보여주기 위해 간단히 개념적으로만 설명한 것 같습니다. 그래서 좀더 실질적인 예제를 제시하고 있지 않습니다. 물론 서론에서 저자가 밝혔듯이 순수한 자바스크립트의 기능에만 초점을 맞추려고 했기 때문에 브라우저에 특화된 비동기식 호출 예제를 제시하지 않은 것 같습니다. 콜백 함수의 구체적인 예는 인터넷이나 다른 자바스크립트 책에서 (흔히 Ajax라고 일컬어지는) XMLHttpRequest를 사용하여 비동기적으로 서버에 요청하는 예제를 찾아보면 됩니다.

```

var entity = {
    quot: '',
    lt: '<',
    gt: '>'
};

// deentityify 메소드 반환.

return function () {

    // 여기가 deentityify 메소드.
    // 문자열의 replace 메소드를 호출하여 &로 시작하고 ;로 끝나는
    // 부분을 찾고 &와 ; 사이의 문자열이 엔티티 테이블에 있으면
    // 해당 문자로 엔티티를 대체. 정규 표현식 사용(7장 참조).

    return this.replace(/&([^\&;]+);/g,
        function (a, b) {
            var r = entity[b];
            return typeof r === 'string' ? r : a;
        }
    );
}
();

```

코드의 마지막 줄을 잘 보기 바랍니다. () 연산자를 사용하여 방금 막 정의한 함수를 바로 호출하는 것을 볼 수 있습니다. 이 호출로 deentityify 메소드가 되는 함수를 생성해서 반환합니다.

```

document.writeln(
    '<'.deentityify() + '> // <">

```

모듈 패턴은 바인딩과 private을 위해 함수의 유효범위와 클로저를 이용합니다. 이 예제에서는 deentityify 메소드만이 엔티티들을 담고 있는 데이터 구조인 entity 객체에 접근할 수 있습니다.

모듈의 일반적인 패턴은 private 변수와 함수를 정의하는 함수입니다. 클로저를 통해 private 변수와 함수에 접근할 수 있는 권한이 있는 함수를 생성하고 이 함수를 반환하거나 접근 가능한 장소에 이를 저장하는 것입니다.

모듈 패턴을 사용하면 전역변수 사용을 없앨 수 있습니다. 이 패턴은 정보은닉과 그 외 다른 좋은 설계 방식을 따를 수 있게 하고, 애플리케이션이나 다른 싱글톤(singleton) 패턴들을 효과적으로 캡슐화할 수 있게 합니다.

모듈 패턴은 또한 안전한 객체를 생성하는데도 사용할 수 있습니다. 이제 시리얼 번호를 생성하는 객체를 만든다고 가정해 보겠습니다.

```
var serial_maker = function () {
    // 유일한 문자열을 생성하는 객체 생성.
    // 유일한 문자열은 접두어와 연속된 숫자 두 부분으로 구성됨.
    // 객체에는 접두어와 연속된 숫자를 설정하는 메소드와
    // 유일한 문자열을 생성하는 gensym 메소드가 있음.

    var prefix = '';
    var seq = 0;
    return {
        set_prefix: function (p) {
            prefix = String(p);
        },
        set_seq: function (s) {
            seq = s;
        },
        gensym: function () {
            var result = prefix + seq;
            seq += 1;
            return result;
        }
    };
};

var seqr = serial_maker();
seqr.set_prefix('Q');
```

```
sequer.set_seq(1000);
var unique = sequer.gensym(); // unique는 "Q1000"
```

메소드가 this나 앞서 살펴본 that(03절의 함수 호출 패턴 예제 참조)을 사용하지 않기 때문에 sequer 내부의 변수를 접근할 수 있는 방법은 없습니다. 즉 해당 변수를 다루도록 정의된 메소드를 제외하고는 prefix나 seq의 값을 얻거나 변경할 수 있는 방법은 없습니다. sequer 객체는 변형될 수 있기 때문에 가지고 있는 메소드들은 다른 메소드로 대체될 수 있지만 그렇다고 해서 대체된 메소드들이 숨겨져 있는 prefix와 seq를 접근할 수는 없습니다. sequer은 단순히 함수들의 집합처럼 보이지만, 이 함수들만이 숨겨진 변수들을 사용하거나 수정할 수 있는 권한이 있습니다.

sequer.gensym을 써드 파티 함수에 넘기면 유일한 문자열을 생성할 수 있지만 써드 파티 함수가 prefix나 seq를 변경할 수 없습니다.

## 13 | 연속 호출(Cascade)

일부 메소드는 반환값이 없습니다. 예를 들어 객체의 상태를 변경하거나 설정하는 메소드들은 일반적으로 반환값이 없습니다. 만약 이러한 메소드들이 undefined 대신에 this를 반환한다면 연속 호출이 가능합니다. 연속 호출을 사용하면 같은 객체에 대해 문장 하나로 연속되는 많은 메소드를 호출할 수 있습니다. 연속 호출을 가능하게 하는 Ajax 라이브러리를 사용하면 다음과 같은 스타일의 프로그래밍이 가능합니다.

```
getElement('myBoxDiv').
move(350, 150).
width(100).
height(100).
color('red').
border('10px outset').
padding('4px').
appendText("Please stand by").
```

```

on('mousedown', function (m) {
    this.startDrag(m, this.getNinth(m));
}).
on('mousemove', 'drag').
on('mouseup', 'stopDrag').
later(2000, function () {
    this.
        color('yellow').
        setHTML("What hath God wraught?").
        slide(400, 40, 200, 200);
}).
tip('This box is resizeable');

```

이 예제에서 getElement 함수는 id가 myBoxDiv인 DOM 엘리먼트에 여러 기능을 추가한 객체를 반환합니다. 이 객체에는 엘리먼트를 이동할 수 있는 메소드, 크기나 스타일을 변경하거나 특정 행동을 추가할 수 있는 메소드들이 포함됩니다. 각각의 메소드는 객체를 반환하기 때문에 각 메소드 호출 결과를 다음 호출에 사용할 수 있습니다.

연속 호출은 매우 표현적인 인터페이스를 제공할 수 있게 합니다. 연속 호출은 한번에 많은 작업을 할 수 있는 인터페이스를 만드는데 도움이 됩니다.

## 14 | 커링(Curry)<sup>6</sup>

함수는 값(value)이며, 이 함수값을 흥미로운 방법으로 다룰 수 있습니다. 커링은 함수와 인수를 결합하여 새로운 함수를 만들 수 있게 합니다.

```

var add1 = add.curry(1);
document.writeln(add1(6));      // 7

```

---

6. 역사 주/ 커링에 익숙하지 않은 분들은 커링이나 Haskell 언어와 커링을 같이 검색하면 원하는 내용을 찾을 수 있습니다. 그리고 영문이기는 하지만 <http://en.wikipedia.org/wiki/Currying>에 아주 잘 정리돼 있습니다.

add1은 add의 curry 메소드에 1을 넘겨서 생성한 함수입니다. add1 함수는 자신의 인수에 1을 더합니다. 자바스크립트는 curry 메소드가 없지만 다음과 같이 Function.prototype에 이를 추가할 수 있습니다.

```
Function.method('curry', function ( ) {
    var args = arguments, that = this;
    return function ( ) {
        return that.apply(null, args.concat(arguments));
    };
}); // 뭔가 잘못된 점이...
```

curry 메소드는 커링할 원래 함수와 인수를 유지하는 클로저를 만드는 방식으로 동작합니다. 이 curry 메소드는 새로운 함수를 만들어 반환하는데 이렇게 반환되는 함수는 curry 메소드를 호출할 때 받은 인수와 자신을 호출할 때 받게 되는 인수를 결합하여 curry를 실행한 원래 함수를 호출합니다. curry 메소드는 arguments 배열 두 개를 연결하기 위해 배열의 concat 메소드를 사용합니다.

불행하게도 앞서 살펴본 것처럼 arguments 배열은 배열이 아닙니다. 그래서 arguments는 concat이라는 메소드가 없습니다. 이 문제를 해결하기 위해 arguments 배열 두 개에 배열의 slice 메소드를 적용할 것입니다. 이렇게 하면 concat 메소드를 포함하는 진정한 배열이 반환됩니다.

```
Function.method('curry', function ( ) {
    var slice = Array.prototype.slice,
        args = slice.apply(arguments),
        that = this;
    return function ( ) {
        return that.apply(null, args.concat(slice.apply(arguments)));
    };
});
```

## 15 | 메모이제이션(memoization)<sup>7</sup>

함수는 불필요한 작업을 피하기 위해서 이전에 연산한 결과를 저장하고 있는 객체를 사용할 수 있습니다. 이러한 최적화 기법을 메모이제이션(memoization)이라고 합니다. 자바스크립트의 객체와 배열은 메모이제이션에 매우 유용합니다.

이제 피보나치 수열을 재귀 함수로 계산하는 경우를 생각해 보겠습니다. 피보나치 수열에서 한 항의 값은 앞선 두 항의 값을 더한 값입니다. 이 수열에서 첫 번째 두 항은 0과 1입니다.

```
var fibonacci = function (n) {
    return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);
};

for (var i = 0; i <= 10; i += 1) {
    document.writeln('// ' + i + ': ' + fibonacci(i));
}

// 0: 0
// 1: 1
// 2: 1
// 3: 2
// 4: 3
// 5: 5
// 6: 8
// 7: 13
// 8: 21
// 9: 34
// 10: 55
```

---

7. 역사 주/ 메모이제이션과 관련해서는 <http://ko.wikipedia.org/wiki/Memoization>를 참조하세요.

이 예제는 잘 동작합니다. 하지만 처리해야 하는 작업이 꽤 많습니다. 자그마치 fibonacci 함수를 453번이나 호출해야 합니다. 여기에서 11번은 직접 호출한 것이고 나머지 442번은 이미 계산한 값들을 다시 계산하기 위해 호출한 횟수입니다. fibonacci 함수에 메모이제이션을 적용하면 작업량을 현저하게 줄일 수 있습니다.

이전에 작업한 결과는 클로저를 통해 숨겨지는 memo라는 배열에 저장할 것입니다. 함수가 호출되면 제일 먼저 결과가 저장돼 있는지를 살핍니다. 그래서 만약 결과가 있는 경우 연산을 수행하지 않고 바로 결과를 반환합니다.

```
var fibonacci = function () {
    var memo = [0, 1];
    var fib = function (n) {
        var result = memo[n];
        if (typeof result !== 'number') {
            result = fib(n - 1) + fib(n - 2);
            memo[n] = result;
        }
        return result;
    };
    return fib;
}();
```

이 함수는 앞서의 함수와 같은 결과를 보이지만 호출은 단지 29번만 발생합니다. 11번은 직접 호출한 것이고 나머지 18번은 앞선 메모이제이션 결과를 얻기 위해 호출한 것입니다.

이러한 메모이제이션 작업은 메모이제이션 함수를 만들 수 있게 도와주는 함수를 만들어서 일반화할 수 있습니다. 다음의 예제 코드에 나오는 momoizer 함수는 결과를 저장할 배열(매개변수 memo)과 메모이제이션을 할 함수(매개변수 fundamental)를 인수로 받습니다. 그런 다음 memo에 저장되는 데이터를 관리하고 필요한 경우 fundamental 함수를 호출하는 shell 함수를 반환합니다. shell 함수와 함수가 받게 되는 인수는 fundamental 함수에 전달합니다.

```

var memoizer = function (memo, fundamental) {
  var shell = function (n) {
    var result = memo[n];
    if (typeof result !== 'number') {
      result = fundamental(shell, n);
      memo[n] = result;
    }
    return result;
  };
  return shell;
};

```

이제 다음과 같이 매개변수 memo에 해당하는 배열과 매개변수 fundamental에 해당하는 메모이제이션할 함수를 memoizer에 전달하여 fibonacci 함수를 정의할 수 있습니다.

```

var fibonacci = memoizer([0, 1], function (shell, n) {
  return shell(n - 1) + shell(n - 2);
});

```

} 8

이런 식으로 다른 함수를 만들어내는 함수를 고안하면 작업 양을 현저하게 줄일 수 있습니다. 예를 들어 다음과 같이 기본적인 계승(factorial) 공식만 제공하면 메모이제이션을 사용하는 factorial이라는 계승 함수를 손쉽게 만들 수 있습니다.

```

var factorial = memoizer([1, 1], function (shell, n) {
  return n * shell(n - 1);
});

```

---

8. 역사 주/ 바로 앞에 제시한 코드의 memoizer에서 shell과 이 예에서의 shell을 구분할 필요가 있습니다. memoizer에서의 shell은 메모이제이션을 행하는 실제 함수이고, 여기서는 함수를 하나 받아서 사용할 것인데 그 매개변수 이름을 shell로 한 것입니다. 물론 여기서 받게 되는 함수는 앞서 제시한 코드에 있는 shell이기 때문에 저자는 이러한 뜻을 나타내기 위해서 shell이라는 같은 이름을 사용한 것입니다.



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

05

## 상속

... 하나가 여러 대상(object)으로 나뉘는군요. 저 사시경(斜視鏡)을 정면으로 보니 그저 혼란한 것이...

— 윌리엄 셰익스피어, 리처드 2세



... 상속은 대부분의 프로그래밍 언어에서 중요한 주제입니다.

자바 같은 클래스 기반의 언어에서 상속(또는 확장)은 두 가지 유용한 점을 제공합니다. 첫째로 상속은 코드 재사용의 한 형태입니다. 만약 새로 만들 클래스가 기존에 있는 클래스와 매우 유사하다면, 상속을 통해 단지 다른 점만을 구현하면 됩니다. 코드를 재사용하는 패턴은 소프트웨어 개발 비용을 현저하게 줄일 수 있는 잠재력이 있기 때문에 매우 중요합니다. 클래스 상속의 또 다른 이점은 상속에 데이터 타입 체계의 명세가 포함된다는 것입니다. 이러한 속성은 프로그래머들이 명시적으로 캐스팅 작업을 해야 할 필요를 없애줍니다. 만약 상속 시에 캐스팅을 해야 한다면 프로그래머들의 작업이 많아지는 것도 많아지는 것이지만 데이터 타입 체계가 안전하게 전달되는 이점을 잊게 됩니다.

데이터 타입 확인이 엄격하지 않은 자바스크립트는 캐스팅을 절대 하지 않습니다. 객체의 계보는 별 상관이 없습니다. 객체에서 중요한 점은 어떤 일을 하느냐지 어디서 유래했는지가 아닙니다.

자바스크립트는 더 풍부한 코드 재사용 패턴을 제공합니다. 자바스크립트에서도

물론 클래스 패턴처럼 상속을 할 수 있지만 그 보다 더 표현적인 다른 패턴들도 지원합니다. 자바스크립트에서 가능한 상속 패턴들은 매우 다양합니다. 이번 장에서는 그 중에서 가장 직관적인 패턴 몇 가지를 살펴볼 것입니다. 물론 훨씬 더 복잡한 방법들이 있지만 언제나 최선의 방법은 단순함을 유지하는 것입니다.

클래스 기반의 언어에서 객체는 클래스의 인스턴스이며 클래스는 다른 클래스로 상속될 수 있습니다. 자바스크립트는 프로토타입 기반 언어인데 이 말은 즉 객체가 다른 객체로 바로 상속된다는 뜻입니다.

## 01 | 의사 클래스 방식(Pseudoclassical)<sup>1</sup>

자바스크립트는 자신의 프로토타입 본질과 모순되는 점들이 있습니다. 자바스크립트의 프로토타입 메커니즘은 클래스와 비슷하게 보이는 일부 복잡한 구문들 때문에 명확히 두드러지지 않습니다. 프로토타입적 본성에 맞게 객체에서 다른 객체로 직접 상속하는 방법을 갖는 대신에 생성자 함수를 통해 객체를 생성하는 것과 같은 불필요한 간접적인 단계가 있습니다.

함수 객체가 만들어질 때, 함수를 생성하는 Function 생성자는 다음과 같은 코드를 실행합니다.

```
this.prototype = {constructor: this};
```

새로운 함수 객체는, 새로운 함수 객체를 값으로 갖는 constructor라는 속성이 있는 객체를 prototype 속성에 할당 받습니다. prototype 객체는 상속할 것들이 저장되는 장소입니다. 자바스크립트는 어떤 함수가 생성자로 사용되기 위해 만들어졌는지 알 수 있는 방법을 제공하지 않기 때문에 모든 함수는 prototype 객체를

1. 역자 주/ 이 책에서 pseudoclass는 의사 클래스라고 번역했습니다. 이 단어에 대해 모조 클래스라고 번역하는 경우도 있고 이렇게 번역하는 것이 의미적으로 더 와닿을 수도 있지만 프로그래밍 언어측면에서 의사 코드, 의사 난수, 의사 컴퓨터 등 의사(擬似)가 많이 사용되고 있기 때문에 의사 클래스라고 번역했습니다. 추가로 구글 검색 결과에서도 모조 클래스보다 의사 클래스가 현저하게 많이 나오고 있습니다.

갖습니다. prototype 객체 내에 기본적으로 할당되는 constructor 속성은 유용하지 않습니다. 중요한 것은 prototype 객체 자체입니다.

new 연산자를 사용하여 생성자 호출 패턴으로 함수가 호출되면 함수가 실행되는 방법이 변경됩니다. 만약 new 연산자가 메소드였다면, 아마 다음과 같이 구현됐을 것입니다.

```
Function.method('new', function () {  
    // 생성자의 프로토타입을 상속받는 새로운 객체 생성.  
    var that = Object.create(this.prototype)  
    // this를 새로운 객체에 바인딩하면서 생성자 호출.  
    var other = this.apply(that, arguments);  
    // 반환값이 객체가 아니면, 새로운 객체로 대체.  
    return (typeof other === 'object' && other) || that;  
});
```

2

다음과 같이 생성자를 정의하고 prototype에 메소드를 추가할 수 있습니다.

```
var Mammal = function (name) {  
    this.name = name;  
};  
  
Mammal.prototype.get_name = function () {  
    return this.name;  
};  
  
Mammal.prototype.says = function () {  
    return this.saying || '';  
};
```

2. 역사 주/ 이 코드를 잘 이해할 필요가 있습니다. 결국 new로 (생성자) 함수를 호출하면 이 함수가 명시적으로 객체를 반환하지 않는 한, 이 함수의 prototype 속성 객체가 프로토타입이 되는 (새로 만든) 객체를 반환합니다.

그리고 나서 다음과 같이 인스턴스를 생성합니다.

```
var myMammal = new Mammal('Herb the Mammal');
var name = myMammal.get_name(); // 'Herb the Mammal'
```

이제 생성자 함수를 정의하고 이 함수의 prototype을 Mammal 인스턴스로 대체하는 방식으로 또 다른 의사 클래스(pseudoclass)를 만들 수 있습니다.

```
var Cat = function (name) {
    this.name = name;
    this.saying = 'meow';
};

// Cat.prototype을 Mammal의 새 인스턴스로 대체.

Cat.prototype = new Mammal();

// 새로운 prototype에 purr와 get_name 메소드 추가

Cat.prototype.purr = function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
        if (s) {
            s += '-';
        }
        s += 'r';
    }
    return s;
};

Cat.prototype.get_name = function () {
    return this.says() + ' ' + this.name +
        ' ' + this.says();
};

var myCat = new Cat('Henrietta');
var says = myCat.says(); // 'meow'
var purr = myCat.purr(5); // 'r-r-r-r-r'
```

```
var name = myCat.get_name( );
//               'meow Henrietta meow'
```

의사 클래스 패턴은 객체지향처럼 보이게 고안 됐지만 이는 마치 어디 외계에서 온 패턴 같습니다. 이렇게 이상한 코드의 일부분을 앞에서 정의했던 method 메소드와 다음과 같이 inherits 메소드를 정의해서 숨길 수 있습니다.

```
Function.method('inherits', function (Parent) {
    this.prototype = new Parent();
    return this;
});
```

inherits와 method 메소드는 this를 반환하는데 이러한 속성 때문에 연속 호출 스타일을 사용할 수 있습니다. 이제 다음과 같이 Cat을 한 문장으로 만들 수 있습니다.

```
var Cat = function (name) {
    this.name = name;
    this.saying = 'meow';
}.
inherits(Mammal),
method('purr', function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
        if (s) {
            s += '-';
        }
        s += 'r';
    }
    return s;
}).
method('get_name', function () {
    return this.says() + ' ' + this.name +
        ' ' + this.says();
});
```

객체의 prototype을 사용하는 것을 method와 inherits라는 메소드를 만들어 숨김으로써 외계에서 온 것 같은 부분이 조금 경감됐습니다. 하지만 실제로 향상된 부분이 있을까요? 이제 클래스 같이 동작하는 생성자 함수를 갖게 됐습니다. 하지만 면면을 살펴보면 전혀 그렇지 않습니다. private은 전혀 없고 모든 속성은 public입니다. 그리고 부모 메소드로의 접근도 전혀 할 수 없습니다.

설상가상으로 생성자 함수를 사용하는데는 심각한 위험이 있습니다. 만약 생성자 함수를 호출할 때 new 연산자를 포함하는 것을 잊게 되면 this는 새로운 객체와 바인딩되지 않습니다. 불행히도 this는 전역객체와 연결되고 이렇게 됨으로써 새로운 객체에 필요한 기능을 추가하게 되는 것이 아니라 전역변수에 영향을 미치게 됩니다. 가히 심각한 문제라고 밖에 할 수 없습니다. new를 누락해도 어떠한 컴파일 경고나 실행시간 경고가 발생하지 않습니다.

이러한 점은 언어에 있어서 심각한 설계 오류입니다. 이러한 문제점을 경감시키기 위한 한가지 방법은 단어 첫 글자를 대문자로 표기하는 표기법(일명 파스칼 표기법)을 모든 생성자 함수의 이름에 사용하고 그 외 다른 것들은 이 표기법을 사용하지 않는 것입니다. 이러한 방법을 사용함으로써 그나마 new를 빼먹은 것을 보다 쉽게 식별할 수 있습니다. 물론 더 나은 대안은 new를 사용하는 방식을 피하는 것입니다.

의사 클래스(pseudoclass)를 사용하는 방법은 자바스크립트에 익숙하지 않은 프로그래머들에게 편안함을 제공합니다. 하지만 이 방법은 자바스크립트라는 언어가 가진 진정한 속성을 가리기도 합니다. 클래스에서 영감 받은 표기법은 프로그래머들에게 불필요하게 복잡하고 단계가 많은 구조를 만들도록 유도할 수 있습니다. 클래스 계층의 복잡함 대부분은 정적 타입 확인이라는 제약사항으로 인해 발생합니다. 자바스크립트는 이러한 제약사항으로부터 완전히 자유롭습니다. 클래스 기반의 언어에서는 클래스 상속이 코드를 재사용할 수 있는 유일한 방법이지만 자바스크립트는 더 좋은 방법들이 있습니다.

## 02 | 객체를 기술하는 객체(Object Specifiers)

때때로 생성자가 매우 많은 매개변수를 갖는 경우가 있습니다. 이런 경우 일일이 인수의 순서를 외우기가 힘들기 때문에 성가실 수 있습니다. 그래서 생성자가 많은 인수를 받는 대신에 객체를 기술하는 하나의 객체를 받도록 정의하면 보다 사용하기 편리한 형태가 됩니다. 객체를 기술하는 객체는 만들어질 객체의 명세를 포함합니다. 그래서 다음과 같이 작성하는 대신에,

```
var myObject = maker(f, l, m, c, s);
```

다음과 같이 작성할 수 있습니다.

```
var myObject = maker({
  first: f,
  last: l,
  state: s,
  city: c
});
```

이제 인수는 꼭 순서를 맞출 필요가 없으며, 생성자가 기본값들을 똑똑하게 설정하고 있다면 인수를 생략할 수도 있습니다. 그리고 이렇게 함으로써 코드의 가독성이 높아집니다.

이러한 방법은 JSON을 사용하여 작업하는 경우 부가적인 이점을 제공합니다 (JSON은 부록 E를 참조하세요). JSON 텍스트는 단지 데이터만을 기술할 수 있습니다. 그런데 JSON으로 기술되는 데이터가 객체인 경우 메소드들과 함께 구성됐을 때, 더 편리한 경우들이 있습니다. 이를 위해 객체를 기술하는 객체를 받아들이는 생성자를 사용하면 쉽게 처리할 수 있습니다. 즉 간단하게 생성자에 JSON 객체를 넘기고, 이 생성자가 필요한 기능을 갖춘 객체를 구성하여 반환하는 형식으로 처리할 수 있습니다.

## 03 | 프로토타입 방식

순수하게 프로토타입에 기반한 패턴에서는 클래스가 필요 없습니다. 대신에 객체에만 초점을 맞추면 됩니다. 프로토타입에 의한 상속은 개념적으로 클래스에 의한 상속보다 더 간단합니다. 즉 새로운 객체는 기존 객체의 속성들을 상속받을 수 있습니다. 이러한 개념이 익숙하지 않을지 모르지만 실제로 이해하기 쉬운 개념입니다. 일단 먼저 유용한 객체를 만드는 것으로 시작합니다. 그러면 이후부터는 이와 유사한 객체들을 보다 많이 만들 수 있습니다. 애플리케이션을 중첩된 추상 클래스들로 분해하는 클래스화 과정은 전혀 필요가 없습니다.

그러면 이제 유용한 객체를 생성하는 객체 리터럴로 시작해 보겠습니다.

```
var myMammal = {
    name : 'Herb the Mammal',
    get_name : function () {
        return this.name;
    },
    says : function () {
        return this.saying || '';
    }
};
```

일단 위와 같은 객체를 생성하고 나면 3장에서 살펴봤던 Object.create 메소드를 사용하여 이 객체의 더 많은 인스턴스를 만들 수 있습니다. 그리고 나서 이렇게 새로 만든 인스턴스를 필요에 맞게 맞춤화할 수 있습니다. 즉 인스턴스에 원하는 대로 메소드나 속성들을 추가할 수 있습니다.<sup>3</sup>

```
var myCat = Object.create(myMammal);
myCat.name = 'Henrietta';
myCat.saying = 'meow';
```

3. 역사 주/ 프로토타입 방식은 Object.create가 핵심이라고 할 수 있습니다. 저자가 권하는 대로 이 메소드를 사용하여 프로토타입 방식으로만 상속을 하는 것도 좋지만 의사 클래스 방식을 이해하고 이 방식을 내부적으로 응용한 Object.create 메소드를 온전히 이해하면 언어를 이해하는데 더욱 도움이 됩니다.

```

myCat.purr = function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
        if (s) {
            s += '-';
        }
        s += 'r';
    }
    return s;
};

myCat.get_name = function () {
    return this.says + ' ' + this.name + ' ' + this.says();
};

```

이러한 방법은 클래스에 의한 상속과는 분명히 구별되는 상속 방법입니다. 새로운 객체를 맞춤화함으로써 기반이 되는 객체와 차이점을 만들 수 있습니다.

이러한 방법은 때때로 기존의 데이터 구조를 상속받는 데이터 구조에 유용합니다. 다음은 그 예입니다. 중괄호로 유효범위(scope)를 지정하는 자바스크립트나 TEC 같은 언어를 파싱한다고 가정해 보겠습니다. 특정 유효범위 내에 정의된 항목은 바깥 유효범위에서 볼 수 없어야 합니다. 어떤 의미에서는 내부의 유효범위가 외부의 유효범위를 상속받는다고 볼 수 있습니다. 자바스크립트의 객체는 이러한 관계를 나타내는데 매우 적합합니다. block이라는 함수는 왼쪽 중괄호를 만나게 되면 호출됩니다. parse 함수는 유효범위 내에서 심볼들을 검색해서 새로운 심볼이 정의된 경우 이를 scope에 추가합니다.

```

var block = function () {

    // 현재 scope를 기억. 현재 scope의 모든 것을 포함하는
    // 새로운 scope 생성.

    var oldScope = scope;
    scope = Object.create(scope);

    // 왼쪽 중괄호를 지나 앞으로 전진.

```

```

advance('{' );
// 새로운 scope를 사용하여 파싱.

parse(scope);

// 오른쪽 중괄호를 지나 전진. 새로운 scope를 포기하고
// 이전 scope 복원

advance('}' );
scope = oldScope;
};

```

## 04 | 함수를 사용한 방식

지금까지 살펴본 프로토타입에 의한 상속 패턴의 한가지 단점은 private 속성을 가질 수 없다는 것입니다. 객체의 모든 속성은 public입니다. private 변수도 private 메소드도 모두 생성할 수 없습니다. 어떤 경우에는 이것이 문제가 안 되지만, 경우에 따라서는 아주 중요한 문제일 수 있습니다. 특히 골치 아픈 경우는 인식 없는 일부 프로그래머들이 이상한 방법으로 private을 흉내내는 패턴을 사용하는 경우입니다. 즉 private이라는 의미로 사용하기 원하는 속성들의 이름을 이상하게 표기하여 다른 코드에서 이 속성들이 사용되지 않게 하는 경우입니다. 다행히도 앞서 살펴본 모듈 패턴에서 좀더 나은 대안을 찾을 수 있습니다.

먼저 객체를 생성하는 함수를 만드는 것으로 시작할 것입니다. 이 함수는 new 연산자를 사용하지 않을 것이기 때문에 이름을 소문자로 시작할 것입니다. 이 함수는 다음의 4 단계로 작업을 진행합니다.

1. 새로운 객체를 생성합니다. 객체를 만드는데는 다양한 방법이 있습니다. 객체 리터럴로 만들 수도 있고, new 연산자를 사용하면서 생성자 함수를 호출할 수도 있고, 기존의 객체에서 새로운 인스턴스를 만들어주는 Object.create 메소드를 사용할 수도 있으며, 객체를 반환하는 함수를 호출할 수도 있습니다.

2. 필요한 private 변수와 메소드를 정의합니다. 이것들은 단지 함수 안의 일반적인 변수입니다.
3. that에 새로운 객체를 할당하고 메소드를 추가합니다. 이때 추가되는 메소드들은 함수의 매개변수와 2번째 단계에서 정의한 변수들을 접근할 수 있는 권한을 갖습니다.
4. 새로운 객체 that을 반환합니다.

다음은 이러한 함수를 위한 의사 코드(pseudocode) 템플릿입니다(강조를 위해 볼드체를 사용했습니다).

```
var constructor = function (spec, my) {
  var that, 필요한 private 변수들;
  my = my || {};

  공유할 변수와 함수를 my에 추가

  that = 새로운 객체

  앞서 정의한 변수들에 접근할 권한이 있는 메소드들을 that에 추가

  return that;
};
```

spec 객체는 constructor가 인스턴스를 만드는데 필요한 모든 정보가 있습니다. spec의 내용들은 private 변수에 복사되거나 다른 함수에 의해서 처리될 수 있습니다. 또한 메소드에서 필요한 정보를 spec에서 얻을 수도 있습니다(간단한 방법은 spec을 하나의 값으로 대체하는 것입니다. 객체를 생성하는데 전체 명세를 나타내는 spec 객체가 필요 없는 경우에는 이러한 방법이 유용합니다).

my 객체는 상속 연결상에서 생성자와 공유하게 되는 비밀들을 담는 컨테이너입니다. my 객체를 사용하는 것은 선택사항입니다. my 객체가 전달되지 않으면 내부에서 이 객체가 만들어집니다.

다음은 private 변수와 메소드를 정의하는 것입니다. 방법은 간단합니다. 단지 constructor 내부에서 변수와 함수를 정의하면 이것들이 private 변수와 함수가 됩니다. 내부 함수는 spec, my, that과 함께 정의된 모든 (private) 변수를 다 접근할 수 있습니다.

다음은 공유할 private 요소들을 my 객체에 추가하는 것입니다. 방법은 다음과 같이 할당하는 것입니다.

```
my.member = value;
```

이제 새로운 객체를 만들고 that에 할당합니다. 새로운 객체는 다양한 방법으로 만들 수 있습니다. 객체 리터럴로 만들 수도 있고, new 연산자를 사용하는 의사 클래스 생성자로 만들 수도 있으며, prototype 객체에 Object.create 메소드를 사용할 수도 있습니다. 또한 별도의 함수형 생성자에 spec, my 객체를 넘겨 만들 수도 있습니다(물론 spec과 my 객체는 constructor에 넘어온 것 그대로 넘길 수 있습니다). my 객체는 별도의 생성자와 my에 넣은 것들을 공유하게 합니다. 여기에서 사용되는 별도의 생성자도 my 객체에 자신만의 비밀을 담아 constructor에 전달할 수 있습니다.<sup>4</sup>

다음은 객체의 인터페이스를 담당하게 될 메소드들을 that에 추가하는 것입니다 (이 메소드들은 private 변수들을 접근할 수 있습니다). 새로운 함수를 that의 구성 요소로 추가하거나, 더 안전하게 다음과 같이 일단 새로운 함수를 private 메소드로 정의한 다음에 이 함수를 that에 할당할 수 있습니다.

```
var methodical = function () {  
    ...  
};  
that.methodical = methodical;
```

---

4. 역사 주/ my에 대한 이해가 명확하지 않다면 다음에 이어지는 05. 클래스 구성을 위한 부속품 절을 보기 바랍니다.

이렇게 두 단계를 거쳐 methodical을 정의하면 좋은 점은 methocidal을 호출하기 원하는 다른 메소드가 that.methodical()로 호출하는 대신 바로 methodical()로 호출할 수 있다는 것입니다. 또한 that.methodical이 대체되어 변경이 된다 하더라도 private인 methodical은 이러한 변경에 영향을 받지 않기 때문에, methodical을 호출하는 메소드는 계속해서 같은 작업을 수행할 수 있게 됩니다.

마지막으로 that을 반환합니다.

이제 앞서 살펴본 mammal 예제에 이 패턴을 적용해 보겠습니다. 여기서는 my가 필요 없기 때문에 my는 없애고 spec 객체만 사용할 것입니다.

다음과 같이 함수를 사용한 패턴을 적용하면 name과 saying은 완전한 private 속성이 됩니다. 이 속성들은 get\_name과 says 메소드만이 접근할 수 있습니다.

```
var mammal = function (spec) {
    var that = {};

    that.get_name = function () {
        return spec.name;
    };

    that.says = function () {
        return spec.saying || '';
    };

    return that;
};

var myMammal = mammal({name: 'Herb'});
```

의사 클래스 패턴에서는 Cat 생성자 함수가 Mammal 생성자가 하는 작업과 같은 작업을 중복해서 해야만 했습니다. 하지만 함수형 패턴에서는 Cat 생성자가 Mammal 생성자를 호출하고 Mammal 생성자가 객체 생성을 위해 필요한 대부분의 작업을 하기 때문에 이러한 작업이 필요 없습니다. Cat은 다만 자신에게 추가되는 부분만 신경 쓰면 됩니다.

```

var cat = function (spec) {
    spec.saying = spec.saying || 'meow';
    var that = mammal(spec);
    that.purr = function (n) {
        var i, s = '';
        for (i = 0; i < n; i += 1) {
            if (s) {
                s += '-';
            }
            s += 'r';
        }
        return s;
    };
    that.get_name = function () {
        return that.says() + ' ' + spec.name +
            ' ' + that.says();
    }
    return that;
};

var myCat = cat({name: 'Henrietta'});

```

함수형 패턴은 또한 super 메소드를 다룰 수 있는 방법을 제공합니다. 이제 메소드 이름을 받아서 해당 메소드를 실행하는 함수를 반환하는 `superior`라는 메소드를 만들어 보겠습니다. `superior` 메소드가 반환하는 함수는 속성이 변경되더라도 원래 함수를 호출합니다.

```

Object.method('superior', function (name) {
    var that = this,
        method = that[name];
    return function () {
        return method.apply(that, arguments);
    };
})();

```

이제 cat과 같으면서 추가적으로 super 메소드를 호출하는 메소드인 get\_name을 가진 coolcat을 통해 시험해 보겠습니다. 이를 위해서는 준비가 필요한데, super\_get\_name이라는 변수를 선언하고 여기에 superior 메소드를 호출한 결과를 할당하는 것입니다.

```
var coolcat = function (spec) {
    var that = cat(spec),
        super_get_name = that.superior('get_name');
    that.get_name = function (n) {
        return 'like ' + super_get_name() + ' baby';
    };
    return that;
};

var myCoolCat = coolcat({name: 'Bix'});
var name = myCoolCat.get_name();
//           'like meow Bix meow baby'
```

함수형 패턴은 유연성이 매우 좋습니다. 이 패턴은 의사 클래스 패턴보다 작업량이 적고 캡슐화와 정보은닉 그리고 super 메소드에 접근할 수 있는 방법까지 제공합니다.

객체의 상태 모두가 private이면 객체는 방탄이 됩니다. 객체의 속성은 대체되거나 삭제될 수 있지만 객체의 무결성은 전혀 영향을 받지 않습니다. 함수형 스타일로 객체를 만들고 객체의 모든 메소드가 this나 (this를 할당받는) that을 사용하지 않는다면 이 객체는 영구적으로 변치 않습니다. 이러한 객체는 단순히 다양한 기능을 하는 함수들을 모아 놓은 집합 역할을 합니다.

이러한 객체는 절대 타협하지 않습니다. 이러한 객체는 주어진 메소드 외에는 내부 상태를 접근할 수 없기 때문에 악의적인 공격자들로부터 안전합니다.

## 05 | 클래스 구성을 위한 부속품

제품을 만들 때 부속품들을 가져다 조립을 하듯이 객체를 구성할 때도 같은 방법으로 할 수 있습니다. 간단하게 이벤트 처리 기능을 객체에 추가하는 함수의 예를 통해 이를 살펴보겠습니다. 이 함수는 on, fire 메소드와 이벤트 목록을 관리하는 private 속성의 registry를 객체에 추가합니다.

```
var eventuality = function (that) {
    var registry = {};

    that.fire = function (event) {

        // 객체에서 이벤트에 상응하는 처리기를 실행시킴.
        // 매개변수 event는 이벤트 이름을 포함하는 문자열이거나
        // 이벤트 이름을 갖고 있는 type 속성을 가진 객체일 수 있음.
        // on 메소드에 의해 등록되는 이벤트 이름과 같은 처리 함수가 호출됨.

        var array,
            func,
            handler,
            i,
            type = typeof event === 'string' ?
                event : event.type;

        // 해당 이벤트에 상응하는 처리 함수 목록 배열이 있으면
        // 루프를 돌면서 이 배열에 등록돼 있는 모든 처리 함수를 실행시킴

        if (registry.hasOwnProperty(type)) {
            array = registry[type];
            for (i = 0; i < array.length; i += 1) {
                handler = array[i];

                // 처리 함수 배열에 속하는 항목 하나는
                // 처리 함수인 method와 매개변수인 parameters라는 배열로 구성됨
                // (parameters는 옵션). method가 함수 자체가 아니라 이름이면
                // this에서 해당 함수를 찾음.

                func = handler.method;
```

```

        if (typeof func === 'string') {
            func = this[func];
        }

        // 처리 함수 호출. parameters가 있으면 이를 넘김.
        // 만약 없으면 event 객체를 넘김.

        func.apply(this,
                    handler.parameters || [event]);
    }
}

return this;
};

that.on = function (type, method, parameters) {

    // 이벤트 등록. handler 항목을 만들고 해당 이벤트 타입의 배열에 추가.
    // 만약 기존에 배열이 없다면 해당 이벤트 타입에 대해 새로운 배열 생성.

    var handler = {
        method: method,
        parameters: parameters
    };
    if (registry.hasOwnProperty(type)) {
        registry[type].push(handler);
    } else {
        registry[type] = [handler];
    }
    return this;
};

return that;
};

```

이제 원하는 객체에 특정 이벤트 처리를 위해 eventuality를 호출할 수 있습니다. 또한 앞서 살펴본 constructor 함수에서 that을 반환하기 전에 다음과 같이 eventuality를 호출할 수도 있습니다.

```
eventuality(that);
```

이러한 방법으로 constructor는 객체에 필요한 기능을 마치 부품을 가져다 조립하듯 추가할 수 있습니다(즉, 이 예처럼 객체에 이벤트 관련 기능이 필요한 경우 eventuality라는 부속을 사용하여 해당 기능을 추가할 수 있습니다). 자바스크립트의 엄격하지 않은 데이터 타입 체크는 이런 부분에서 큰 이점을 줍니다. 왜냐하면 클래스의 상속 계통에서 일일이 데이터 타입 체계를 신경쓸 필요가 없기 때문입니다. 대신에 각각이 가진 내용과 기능들에만 초점을 맞추면 됩니다.

만약 eventuality에서 객체의 private 부분들을 접근하기 원한다면 my를 넘기면 됩니다.<sup>5</sup>

---

5. 역사 주/ 예를 든 eventuality의 기능은 다음의 간단한 예제를 실행해보면 쉽게 이해할 수 있습니다.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title> eventuality 예제 </title>
</head>

<body>
사각형을 클릭하세요<br>
<div id="divTest" style="width:100px; height:100px; border:solid 1px black; cursor:pointer;" onclick="this.fire(event)"></div>
<script type="text/javascript">
<!--
    var eventuality = function (that) {
        함수 내용은 위의 내용 참조
    };

    var oDiv = eventuality(document.getElementById('divTest'));
    oDiv.on('click', function(){alert('test')});
<!-->
</script>
</body>
</html>
```



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

06

## 배열

너는 내가 몰아내겠다. 양 가죽(sheep's array)을 쓴 늑대야.

— 윌리엄 셰익스피어, 헨리 6세 제1부



배열은 해당 항목의 오프셋을 계산할 수 있는 정수를 통해 각 항목들을 접근할 수 있는 연속적인 메모리 할당입니다. 보통 배열은 매우 빠른 데이터 구조입니다. 하지만 불행하게도 자바스크립트에는 이런 류의 배열은 없습니다.

대신에 자바스크립트는 배열 같은 특성을 지닌 객체를 제공합니다. 자바스크립트는 배열 첨자를 문자열로 변환하여 속성을 만듭니다. 이는 실제 배열보다 심각하게 느리지만, 사용하는데는 더 편리할 수 있습니다. 속성들을 읽거나 캐싱하는 작업은 정수형 이름을 가진 속성에 좀 특별한 트릭이 있다는 것을 제외하고는 일반 객체와 똑같습니다. 배열에는 자신만의 리터럴 형식이 있습니다. 또한 배열에는 8장에 기술돼 있는 것처럼 매우 유용한 내장 메소드들이 있습니다.

## 01 | 배열 리터럴

배열 리터럴은 새로운 배열을 만드는데 매우 편리한 표기법으로 값이 없거나 하나 이상의 값을 쉼표로 구분하여 대괄호로 묶은 것입니다. 배열 리터럴은 표현식이 위치할 수 있는 곳이라면 어디에라도 위치할 수 있습니다. 배열의 첫 번째 값은 속성 '0'으로 읽을 수 있습니다. 두 번째는 '1', 세 번째는 '2' 이런 식으로 배열 요소들을 읽을 수 있습니다.

```
var empty = [];
var numbers = [
  'zero', 'one', 'two', 'three', 'four',
  'five', 'six', 'seven', 'eight', 'nine'
];

empty[1]           // undefined
numbers[1]          // 'one'

empty.length        // 0
numbers.length      // 10
```

다음은 유사한 결과를 보이는 객체 리터럴입니다.

```
var numbers_object = {
  '0': 'zero', '1': 'one', '2': 'two',
  '3': 'three', '4': 'four', '5': 'five',
  '6': 'six', '7': 'seven', '8': 'eight',
  '9': 'nine'
};
```

numbers와 number\_object 모두 10개의 속성을 가졌고 각 속성은 모두 같은 이름과 같은 값이 있습니다. 하지만 두 객체에 근본적인 차이점이 있는데, numbers는 Array.prototype을 상속했고, number\_object는 Object.prototype을 상속했

다는 점입니다. 그래서 numbers만이 많은 수의 유용한 메소드를 상속하게 됩니다. 또한 numbers는 numbers\_object가 가지지 못한 length라는 신비한 속성이 있습니다.

대부분의 언어에서 배열의 구성요소들은 모두 같은 데이터 타입이어야 합니다. 하지만 자바스크립트에서는 배열 하나에 어떤 데이터 타입의 조합이라도 다 포함될 수 있습니다.

```
var misc = [
    'string', 98.6, true, false, null, undefined,
    ['nested', 'array'], {object: true}, NaN,
    Infinity
];
misc.length // 10
```

## 02 | length 속성

모든 배열은 length 속성이 있습니다. 여타 다른 언어들과는 달리 자바스크립트에서 배열의 길이는 상계(upper bound) 기반이 아닙니다. 만약 현재 length보다 더 큰 첨자로 항목을 추가하면 length는 새로운 항목을 추가할 수 있게 늘어납니다. 어떠한 배열 경계 오류도 발생하지 않습니다.

length 속성은 배열의 가장 큰 정수 속성 이름보다 하나 더 큰 값입니다. 그렇기 때문에 length의 값이 배열에 있는 속성의 수와 반드시 일치하는 것은 아닙니다.

```
var myArray = [];
myArray.length // 0

myArray[1000000] = true;
myArray.length // 1000001
// myArray는 단지 하나의 속성만 가짐.
```

첨자를 둘러싸는 [] 연산자는 안쪽의 표현식이 `toString` 메소드를 가진 경우 이를 사용하여 표현식을 문자열로 변환합니다. 이렇게 변환된 문자열이 속성 이름으로 사용됩니다. 만약 문자열이 배열의 현재 `length` 값보다 크거나 같은 양수이면서 4,294,967,295보다 작은 경우 배열의 `length` 속성의 값을 새로운 첨자에 1이 더해진 값으로 할당됩니다.

`length`의 값은 명시적으로 설정할 수 있습니다. `length` 값을 크게 설정해도 배열을 위해 더 많은 공간이 할당되지는 않습니다. 하지만 `length` 값을 현재 보다 크게 설정했을 경우 설정한 값보다 크거나 같은 첨자에 해당하는 속성은 모두 삭제됩니다.

```
numbers.length = 3;  
// numbers is ['zero', 'one', 'two']
```

새로운 항목을 배열의 현재 `length` 값으로 추가하면 배열의 끝에다 추가할 수 있습니다.

```
numbers[numbers.length] = 'shi';  
// numbers is ['zero', 'one', 'two', 'shi']
```

배열의 마지막에 새로운 항목을 추가하는 것은 때때로 `push` 메소드를 사용하는 것 이 더욱 편리합니다.

```
numbers.push('go');  
// numbers is ['zero', 'one', 'two', 'shi', 'go']
```

## 03 | 삭제

자바스크립트의 배열은 실제 객체이기 때문에 배열의 요소를 삭제하는데 delete 연산자를 사용할 수 있습니다.

```
delete numbers[2];
// numbers는 ['zero', 'one', undefined, 'shi', 'go']
```

불행히도 이런 식으로 배열의 요소를 삭제하면 그 위치에 구멍이 생기게 됩니다. 즉 삭제한 요소의 오른쪽에 있는 것들은 계속해서 자신의 원래 이름을 유지합니다. 하지만 배열의 한 요소를 삭제했을 때 보통 원하는 것은 삭제한 요소 오른쪽 요소들의 이름이 수치적으로 하나씩 줄어들게 변하는 것입니다.

다행히도 자바스크립트 배열에는 splice라는 메소드가 있습니다. 이 메소드는 배열을 수술할 수 있는 능력이 있는데, 즉 배열 요소의 일부를 삭제하고 이를 나머지 요소들로 대체할 수 있습니다. 이 메소드의 첫 번째 인수는 배열의 시작점이고 두 번째 인수는 삭제할 요소의 수입니다. 이 외의 추가적인 인수들은 배열의 삭제한 지점에 추가되는 요소들입니다.

```
numbers.splice(2, 1);
// numbers는 ['zero', 'one', 'shi', 'go']
```

값이 'shi'인 속성의 이름은 3에서 2로 변경됩니다. 삭제된 속성 뒤에 있는 모든 속성은 삭제된 후에 새로운 이름으로 다시 삽입됩니다. 이러한 과정을 거치기 때문에 배열이 아주 큰 경우에는 빠르게 동작하지 않을 수 있습니다.

## 04 | 열거

계속 언급되지만 자바스크립트의 배열은 실제 객체이기 때문에 for in 문으로 배열의 모든 속성을 열거할 수 있습니다. 하지만 for in 문이 배열을 열거하는데 그다지 적합한 편은 아닙니다. 왜냐하면 대부분의 배열을 열거하는 작업에서는 배열의 첨자 순으로 열거되는 것을 당연하게 생각하는데 반해 for in 문은 속성들의 순서를 보장하지 않습니다. 또한 for in 문을 사용하면 프로토타입 체인(prototype chain)에 있는 예상치 못한 속성도 열거될 수 있습니다.

다행히도 일반적인 for 문을 사용하여 이러한 문제를 피할 수 있습니다. 다음의 예처럼 배열의 length 속성을 반복 횟수의 조건으로 하여 for 문을 사용하면 됩니다.

```
var i;
for (i = 0; i < myArray.length; i += 1) {
    document.writeln(myArray[i]);
}
```

## 05 | 객체와 배열의 혼동

자바스크립트 프로그램에서 흔한 오류 중 하나는 배열이 필요할 때 객체를 사용한다거나 객체가 필요할 때 배열을 사용하는 경우입니다. 규칙은 간단합니다. 속성 이름이 작은 크기의 연속된 정수이면 배열을 사용하고 그렇지 않으면 객체를 사용하는 것입니다.

자바스크립트 자체도 배열과 객체의 차이점을 혼동하고 있다고 볼 수 있습니다. 왜냐하면 typeof 연산자로 배열의 타입을 확인해보면 'array'가 아니라 'object'라고 알려주기 때문입니다.

자바스크립트에는 배열과 객체를 구분하는 마땅한 메커니즘이 없습니다. 이러한 결점을 보완하기 위해 다음과 같은 is\_array 함수를 만들어 사용할 수 있습니다.

```
var is_array = function (value) {
    return value &&
        typeof value === 'object' &&
        value.constructor === Array;
};
```

이 함수는 한가지 문제가 있는데 다른 창(window)이나 프레임에서 생성한 배열은 구분하지 못한다는 것입니다. 이러한 문제점을 고치려면 다음과 같이 좀더 작업을 해야 합니다.

```
var is_array = function (value) {
    return value &&
        typeof value === 'object' &&
        typeof value.length === 'number' &&
        typeof value.splice === 'function' &&
        !(value.propertyIsEnumerable('length'));
};
```

먼저 value가 참인지를 확인합니다. 이렇게 함으로써 null이나 여타 다른 거짓값을 배제할 수 있습니다. 두 번째로 typeof value가 'object'인지 확인합니다. value가 객체나 배열 또는 (조금 이상하지만) null인 경우에 참이 됩니다. 세 번째로 value에 숫자값을 가진 length 속성이 있는지를 확인합니다. 이 부분은 배열인 경우에 항상 참이며 객체인 경우에는 보통 거짓입니다. 네 번째로 value가 splice 메소드를 가졌는지를 확인합니다. 이 부분은 배열인 경우에만 참이 됩니다. 마지막으로 length 속성이 열거 가능한지를 확인합니다. 즉 for in 문으로 열거 가능한지를 확인하는 것입니다. 배열인 경우 length는 열거할 수 없습니다. 이러한 방법이 필자가 찾아낸 가장 신뢰할 만한 방법입니다. 이 방법의 문제점이라면 절차가 좀 복잡하다는 것입니다.

이러한 확인 방법을 사용하면 넘어온 값이 단일 값인지 배열인지를 구분하여 그에 맞는 작업을 하는 함수를 만들 수 있습니다.

## 06 | 배열의 메소드

자바스크립트는 배열에 동작하는 메소드들을 제공합니다. 이 메소드들은 Array.prototype에 저장돼 있는 함수들입니다. 3장에서 Object.prototype에 원하는 것들을 추가하는 것을 살펴본 것처럼, Array.prototype에도 원하는 메소드를 추가할 수 있습니다.

예를 들어 배열을 대상으로 연산을 할 수 있는 메소드를 추가하고 싶다고 가정해 보겠습니다.

```
Array.method('reduce', function (f, value) {
    var i;
    for (i = 0; i < this.length; i += 1) {
        value = f(this[i], value);
    }
    return value;
});
```

Array.prototype에 함수를 추가하면 모든 배열이 추가한 메소드를 상속받게 됩니다. 이제 함수와 시작값을 취하는 reduce라는 메소드를 정의했습니다. 배열의 각 요소들은 reduce에 넘겨진 함수에 value와 같이 넘겨지고 계산된 값이 다시 value에 저장됩니다. 모든 작업을 마쳤을 때 value를 반환합니다. 만약 두 수를 더하는 함수를 넘겼다면 reduce 호출의 결과는 배열 요소들 전체의 합이 되고, 두 수를 곱하는 함수를 넘겼다면 전체의 곱이 됩니다.

```
// 숫자들이 요소인 배열 생성.
var data = [4, 8, 15, 16, 23, 42];

// 간단한 함수 두 개를 정의.
// 하나는 두 수를 더하는 함수이고 다른 하나는
// 두 수를 곱하는 함수.

var add = function (a, b) {
```

```

        return a + b;
    };

var mult = function (a, b) {
    return a * b;
};

// add 함수를 넘기면서 data의 reduce 메소드 호출.

var sum = data.reduce(add, 0);      // 합은 108

// multiply 함수를 넘기면서 reduce 메소드를 다시 호출.

var product = data.reduce(mult, 1);
// 곱은 7418880

```

배열은 실제 객체이기 때문에 다음과 같이 개별 배열에 직접적으로 메소드를 추가할 수 있습니다.

```

// data 배열에 total 메소드 추가.

data.total = function () {
    return this.reduce(add, 0);
};

total = data.total();      // total은 108

```

문자열 'total'은 정수가 아니기 때문에, total 속성을 추가한다고 해도 length의 값은 변하지 않습니다. 배열은 속성들의 이름이 정수일 때 가장 유용하지만 배열은 객체이고 객체는 어떠한 문자열도 속성 이름으로 허용합니다.

3장에 나오는 Object.create 메소드는 배열이 아니라 객체를 반환하기 때문에 배열에 사용하는 것은 별로 유용하지 않습니다. Object.create로 배열을 받아 만들어진 객체는 배열의 값과 메소드를 상속받기는 하지만 배열의 특수 속성인 length는 갖지 못합니다.

## 07 | 배열의 크기와 차원

자바스크립트의 배열은 보통 초기화되지 않습니다. 만약 새로운 배열을 []로 만들게 되면 배열은 비어있게 됩니다. 그리고 존재하지 않는 요소를 접근하게 되면 undefined 값을 얻게 됩니다. 이러한 사실을 알고 있거나 또는 배열 요소를 참조하기 전에 모든 배열 요소의 값을 설정한다면 배열을 사용하는데 아무런 문제가 없습니다. 하지만 만약에 배열의 모든 요소가 0과 같이 알 수 있는 값으로 시작한다고 가정하는 알고리즘을 구현하는 경우라면 이에 문제가 없게 배열을 준비시켜야 합니다. 이를 위해 자바스크립트는 Array.dim 같은 메소드를 제공했어야 합니다. 하지만 현실은 그렇지 못하며 다음과 같은 메소드를 정의함으로써 이러한 결점을 고칠 수 있습니다.

```
Array.dim = function (dimension, initial) {
    var a = [], i;
    for (i = 0; i < dimension; i += 1) {
        a[i] = initial;
    }
    return a;
};

// 10개의 0을 갖는 배열 생성.
var myArray = Array.dim(10, 0);
```

자바스크립트에는 다차원 배열이 없지만 대부분 C 유형의 언어처럼 다음과 같이 배열의 배열을 사용할 수 있습니다.

```
var matrix = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
];
matrix[2][1] // 7
```

2차원 배열이나 배열들의 배열을 만들기 위해서는 다음과 같이 직접 배열을 만들어야 합니다.

```
for (i = 0; i < n; i += 1) {  
    my_array[i] = [];  
}  
  
// 주의: Array.dim(n, [])는 여기에서 동작하지 않음.  
// 각각의 요소들이 같은 배열의 참조를 갖게 됨.
```

이 코드는 각 배열들을 초기화하지 못합니다. `my_array`의 요소들인 각 배열을 초기화하려면 일일이 명시적으로 설정해야 합니다. 이런 경우를 위해 자바스크립트가 행렬을 위한 메소드를 지원하면 좋겠지만 역시 그렇지 못합니다. 다시 한번 이러한 결점을 다음과 같은 코드로 보완할 수 있습니다.

```
Array.matrix = function (m, n, initial) {  
    var a, i, j, mat = [];  
    for (i = 0; i < m; i += 1) {  
        a = [];  
        for (j = 0; j < n; j += 1) {  
            a[j] = initial;  
        }  
        mat[i] = a;  
    }  
    return mat;  
};  
  
// 0으로 채워진 4 * 4 행렬 생성.  
  
var myMatrix = Array.matrix(4, 4, 0);  
  
document.writeln(myMatrix[3][3]);      // 0  
  
// 행과 열이 같은 수의 행렬을 만드는 메소드
```

```
Array.identity = function (n) {
    var i, mat = Array.matrix(n, n, 0);
    for (i = 0; i < n; i += 1) {
        mat[i][i] = 1;
    }
    return mat;
};

myMatrix = Array.identity(4);

document.writeln(myMatrix[3][3]);      // 1
```



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

07

## 정규 표현식

그러나 그 반대의 경우는 천복을 가져오며, 평화의 모범(pattern)인 것입니다. 국왕 헨리 폐하에게 어울리는(match) 배우자는...

– 윌리엄 셰익스피어, 헨리 6세 제I부



자바스크립트의 많은 부분은 다른 언어들에서 차용한 것입니다. 구문은 자바, 함수는 Scheme, 프로토타입에 의한 상속은 Self에서 비롯된 것입니다. 그리고 자바스크립트의 정규 표현식은 Perl에서 가져온 것입니다.

정규 표현식은 간단한 언어 구문의 특화된 형태입니다. 정규 표현식은 문자열에서 특정 내용을 찾거나 대체 또는 발췌하는데 사용합니다. 정규 표현식을 사용하는 메소드는 regexp.exec, regexp.test, string.match, string.replace, string.search, string.split 등이 있습니다. 이러한 메소드들에 대해서는 8장에서 설명합니다. 자바스크립트에서 정규 표현식은 보통 같은 문자열에 대해 반복해서 연산을 수행할 때 주목할 만한 성능상의 이점이 있습니다.

정규 표현식은 정규 언어에 대한 수학적 연구에서 비롯된 것입니다. Ken Thompson은 Stephen Kleene의 이론을 type-3 언어에 적용하여, 텍스트 에디터 같은 툴이나 프로그래밍 언어에서 특정 패턴을 찾을 수 있게 실질적인 패턴 매칭 기능을 추가했습니다.

자바스크립트에서 정규 표현식 구문은 약간의 재해석과 Perl의 확장 구문을 채택한 것을 제외하고는 벨 연구소<sup>1</sup>에서 비롯된 원래의 체계를 거의 그대로 따르고 있습니다. 정규 표현식에서는 특정 위치의 문자가 연산자로 해석되고 또 조금은 다른 위치에 있는 문자를 리터럴로 간주하기 때문에 이를 작성하는 것은 매우 복잡할 수 있습니다. 그런데 작성의 어려움보다 더 안 좋은 것은 이러한 어려움이 정규 표현식을 읽기 어렵게 만들고 수정하기 위험하게 만든다는 것입니다. 정규 표현식을 바르게 읽기 위해서는 복잡한 정규 표현식 체계 전체를 온전히 이해해야 합니다. 필자는 이러한 어려움을 조금이나마 쉽게 하기 위해서 규칙들을 조금 단순화시켰습니다. 이 장에서 설명하는 방식으로 하면, 정규 표현식이 간결한 편은 아닐지 몰라도, 바르게 사용하는 데는 생각보다 쉬울 수 있습니다. 또한 이렇게 하면 유지보수나 디버깅의 어려움을 조금이라도 경감할 수 있습니다.

오늘날의 정규 표현식은 엄격하게 정규적이지는 않지만 매우 유용합니다. 정규 표현식은 매우 간결해지는 경향이 있습니다. 그래서 심지어는 암호화된 것처럼 보이기도 있습니다. 간단한 형태에서는 사용이 쉽지만, 곧 하나 둘 추가되다 보면 급격히 복잡해집니다. 자바스크립트의 정규 표현식은 주석이나 공백을 허용하지 않기 때문에 다소 읽기가 어렵습니다. 즉 정규 표현식의 모든 부분은 빈틈 없이 하나로 연결돼 있기 때문에 해석하기가 쉽지 않습니다. 이러한 부분은 정규 표현식이 탐색이나 검증을 위해 보안 애플리케이션에서 사용될 때 특히 염려되는 부분입니다. 정규 표현식을 제대로 읽고 해석할 수 없다면 어떻게 원하는대로 동작한다고 확신할 수 있을까요? 하지만 이러한 명백한 단점에도 불구하고 정규 표현식은 널리 사용되고 있습니다.

---

1. 역사 주/ Ken Tompson이 일하던 곳. 벨 연구소는 컴퓨터 역사에 있어서 많은 역할을 합니다.

## 01 | 예제

다음은 URL에 일치하는 정규 표현식 예제입니다. 이 책의 폭은 무한대로 넓지 않기 때문에 예제를 두 줄로 나눴습니다. 하지만 편의상 둘로 나눈 것뿐이며, 실제 자바스크립트 프로그램에서 정규 표현식은 한 줄이어야만 합니다. 공백문자(Whitespace)는 사용할 수 없습니다.

```
var parse_url = /^(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)(?::(\d+))?(?:\/([^\?#]*))?(?:\?([^\#]*))?(?:\#(.*))?$/;
var url = "http://www.ora.com:80/goodparts?q#fragment";
```

parse\_url의 exec 메소드를 호출해 보겠습니다. 이 메소드는 넘겨받은 URL 문자열에서 일치하는 부분을 찾게 되는 경우, 이 부분들을 추출하여 배열에 담아 반환합니다.

```
var url = "http://www.ora.com:80/goodparts?q#fragment";
var result = parse_url.exec(url);
var names = ['url', 'scheme', 'slash', 'host', 'port',
    'path', 'query', 'hash'];
var blanks = '        ';
var i;
for (i = 0; i < names.length; i += 1) {
    document.writeln(names[i] + ':' +
        blanks.substring(names[i].length), result[i]);
}
```

실행 결과는 다음과 같습니다.

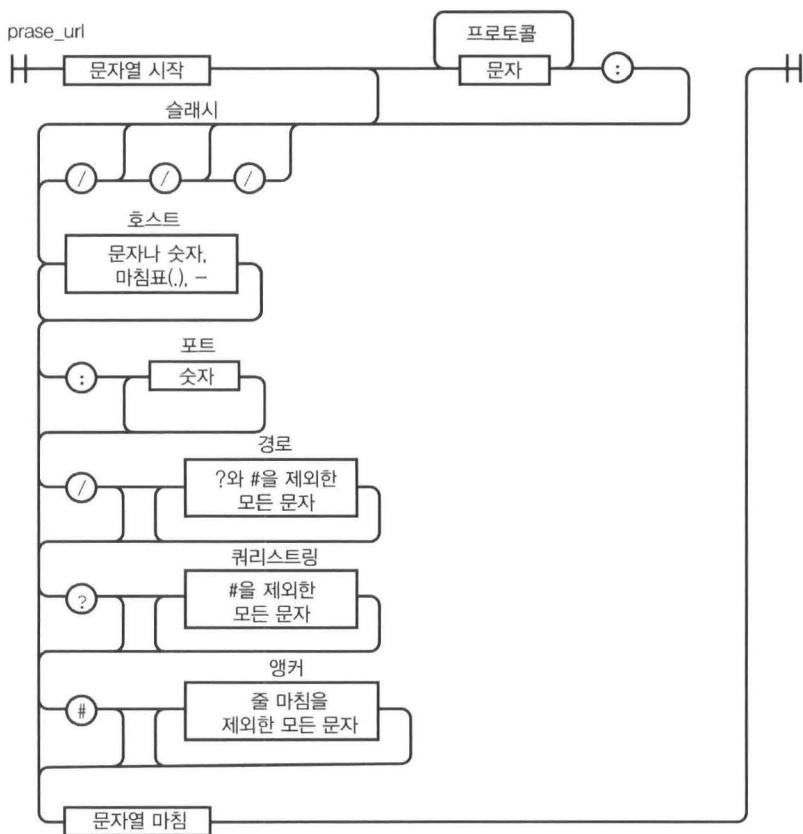
```
url:      http://www.ora.com:80/goodparts?q#fragment
scheme:  http
```

```

slash:  //
host:   www.ora.com
port:   80
path:   goodparts
query:  q
hash:   fragment

```

2장에서는 자바스크립트 언어를 기술하기 위해서 철도 다이어그램을 사용했습니다. 이 다이어그램은 정규 표현식으로 정의된 내용을 기술하는데도 사용할 수 있습니다. 철도 다이어그램을 사용하여 표현하면 정규 표현식이 하는 일을 보다 쉽게 이해할 수 있습니다. 다음은 `prase_url`에 대한 철도 다이어그램입니다.



정규 표현식은 함수처럼 작은 부분으로 나눌 수 없기 때문에 `parse_url`을 나타내는 트랙은 여러 개가 아니라 길게 이어진 하나입니다.

이제 `parse_url`의 각 부분들이 어떻게 동작하는지 분해해 보겠습니다.

^

^ 문자는 문자열의 시작을 나타냅니다. 이 문자는 exec 메소드가 URL 같지 않은 시작을 보이는 문자열은 바로 건너뛰게 하는 역할을 합니다.

(?: ([A-Za-z]+) : ) ?

이 부분은 프로토콜 이름(http, ftp 등등)에 일치하는 부분입니다. 단, 프로토콜 이름 다음에 :이 올 때만입니다. (?: ...)는 캡처하지 않는 그룹을 나타냅니다. 끝에 붙은 ?는 이 그룹이 옵션이라는 뜻입니다. 즉 이 말은 없거나 한 번 반복된다 는 뜻입니다. (...)는 캡처하는 그룹을 나타냅니다. 캡처 그룹은 일치하는 텍스트를 복사하여 이를 결과 배열에 넣습니다. 각각의 캡처 그룹에는 번호가 주어집니다. 첫 번째 캡처 그룹은 1입니다. 그래서 첫 번째 캡처 그룹에 일치하는 텍스트의 복사본도 `result[1]`에 나타납니다. [...]는 문자 클래스를 나타냅니다. 문자 클래스 A-Za-z는 알파벳 대문자 26자와 소문자 26자를 포함합니다. -는 A부터 Z 까지 범위를 나타냅니다. 접미사 +는 문자 클래스가 한 번 이상 일치한다는 것을 나타냅니다. 뒤이어 나오는 :는 문자 그대로 그 위치에 : 문자가 일치해야 한다는 뜻입니다.

(\/{0,3})

다음 부분은 2번째 캡처 그룹입니다. \/는 /(슬래시)와 일치해야 한다는 것을 의미합니다. /(슬래시)는 정규 표현식 리터럴의 끝으로 잘못 해석되지 않게 \((백슬래시)로 이스케이프됩니다. 뒤에 붙는 {0,3}는 /(슬래시)가 0에서 3번 일치 중에 하나라는 것을 나타냅니다.

```
( [0-9 . \-A-Za-z] )+
```

다음은 3번째 캡처 그룹입니다. 이 부분은 호스트 이름과 일치하는 부분인데 하나 이상의 숫자나 문자 그리고 .(마침표)나 -로 구성됩니다. -는 범위를 나타내는 하이픈과 혼동을 막기 위해서 이스케이프됩니다.

```
(? :: (\d+)) ?
```

다음 부분은 옵션으로 올 수 있는 포트 번호입니다. 포트 번호는 : 다음에 하나 이상의 숫자의 연속입니다. \d는 숫자 문자를 나타냅니다. 하나의 이상의 숫자는 4번째 캡처 그룹이 됩니다.

```
(? : \ / ([^?#]*)) ?
```

또 하나의 옵션 그룹이 있는데 이 그룹은 /로 시작합니다. 문자 클래스 [^?#]는 ^로 시작하는데 이것은 ?와 #를 제외한 모든 문자를 나타냅니다. \*는 0번 이상 일치한다는 것을 의미합니다.

여기서 한가지 짚고 넘어가야 할 것이 있습니다. ?와 #를 제외한 모든 문자 클래스에는 줄 마침 문자, 제어문자뿐만 아니라 여기에서 일치해서는 안 되는 많은 수의 다른 문자들이 포함됩니다. 일치하기 원하는 대로 정규식을 작성했지만 여전히 원치 않는 문자가 끼어들 위험성이 남아 있습니다. 어설픈 정규 표현식은 보안 익스플로잇(exploit)의 소스가 될 수 있습니다. 엄격한 정규 표현식을 작성하는 것보다 어설픈 정규식을 작성하기가 훨씬 쉽습니다.

```
(?:\:?( [^#]* ) )?
```

다음은 ?로 시작하는 옵션 그룹입니다. 여기에는 #이 아닌 문자가 0번 이상 나오는 6번째 캡처 그룹이 포함됩니다.

```
(?:#(.*) )?
```

이제 #로 시작하는 마지막 옵션 그룹입니다. .(마침표)는 라인 종료 문자를 제외한 어떤 문자하고도 일치된다는 것을 의미합니다.

§

\$는 문자열의 끝을 의미합니다. \$를 마지막에 붙임으로써 URL의 마지막 다음에 어떠한 여분의 문자도 없다는 것을 나타내게 됩니다.

여기까지가 정규 표현식 parse\_url의 모든 부분입니다.<sup>2</sup>

parse\_url보다 더 복잡한 정규 표현식도 만들 수 있지만 필자는 권하고 싶지 않습니다. 정규 표현식은 길이가 짧고 간단할 때 최상이라고 할 수 있습니다. 이런 경우에만 정규 표현식이 제대로 동작하는지 확신할 수 있고 필요한 경우 문제없이 수정할 수 있습니다.

자바스크립트 언어를 지원하는 환경들 간에는 상당히 높은 수준의 호환성이 있습니다. 이러한 상황에서도 가장 이식성이 떨어지는 부분이 정규 표현식 구현 부분입니다. 매우 복잡하거나 뒤엉킨 정규 표현식은 이식성 문제가 발생할 확률이 더 높습니다. 또한 중첩된 정규 표현식은 일부 환경에서 치명적인 성능 저하 문제를 야기하기도 합니다. 결국 단순함이 최상의 전략입니다.

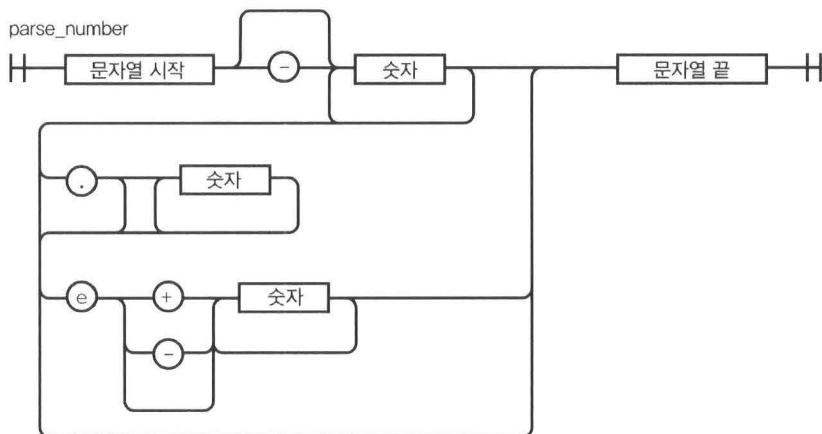
2. 저자 주 / 이 부분을 다음과 같이 모아 놓으면 시각적으로 매우 훈란스럽습니다.

```
/^(?:([A-Za-z]+)?(₩/(0,3))([0-9₩-A-Za-z]+)(?:₩d+)?(?:₩/[("^?#"]*))?(?:₩?([^#]*))?(?:#(.*)?)$/
```

이제 또 하나의 예를 살펴보겠습니다. 살펴볼 정규 표현식은 숫자에 일치하는 것입니다. 숫자는 정수 부분(필요한 경우 앞에 -기호 포함)과 경우에 따라 소수 부분과 지수 부분을 갖습니다.

```
var parse_number = /^-?\d+(?:(\.\d*)?(?:e[+\-]?\d+)?$/i;  
  
var test = function (num) {  
    document.writeln(parse_number.test(num));  
};  
  
test('1'); // true  
test('number'); // false  
test('98.6'); // true  
test('132.21.86.100'); // false  
test('123.45E-67'); // true  
test('123.45D-67'); // false
```

parse\_number는 문자열이 기준에 맞는지 틀리는지를 잘 분별합니다. 하지만 맞지 않는 경우 왜 그런지 아니면 어디서 문제가 있는지에 대한 정보는 제공하지 않습니다.



이제 parse\_number를 하나씩 살펴보겠습니다.

/ ^      \$ / i

또 다시 ^와 \$이 사용되고 있습니다. 이것은 결국 텍스트 내의 모든 문자가 정규 표현식에 대응해서 일치해야 한다는 뜻입니다. 만약 이 두 문자를 제외하면, 숫자가 포함된 문자열을 알려주는 정규 표현식이 됩니다. 그리고 ^만을 사용하면 숫자로 시작하는 문자열에 대응하게 되고 \$만을 사용하면 숫자로 끝나는 문자열에 대응하게 됩니다.

i 플래그는 문자가 일치하는지를 검사할 때 대소문자를 구분하지 않게 합니다. 이번 패턴에 들어 있는 유일한 문자는 e입니다. e 부분을 [Ee]나 (?:(E|e))로 나타낼 수도 있지만 i 플래그를 사용함으로써 그럴 필요가 없습니다.

- ?

- 기호 뒤에 붙는 ?는 음수 기호가 옵션이라는 것을 나타냅니다.

\d+

\d는 [0-9]와 같은 의미입니다. 즉 숫자에 대응합니다. 뒤에 붙는 +는 하나 이상의 숫자에 일치한다는 것을 의미합니다.

(?: \. \d\*) ?

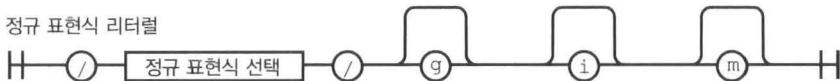
(?: . . . )?는 옵션인(뒤에 붙은 ?) 비캡처 그룹을 나타냅니다. 캡처 그룹은 성능상의 약점이 있기 때문에 보통의 경우 약간 복잡함이 더해지더라도 캡처 그룹보다는 비캡처 그룹을 사용하는 것이 좋습니다. 이 그룹은 소수점과 그 뒤에 오는 0개 이상의 숫자에 대응합니다.

```
(?:e[+\-]?\d+)?
```

이 부분은 또 하나의 옵션인 비캡처 그룹입니다. 이 부분은 e(또는 E)와 선택적인 부호(+/-) 그리고 하나 이상의 숫자(0-9)에 대응합니다.

## 02 | 정규 표현식 객체 생성

정규 표현식 객체인 RegExp 객체는 두 가지 방법으로 만들 수 있습니다. 선호되는 방법은 앞선 예제에서 살펴본 것과 같이 정규 표현식 리터럴을 사용하는 것입니다.<sup>3</sup>



정규 표현식 리터럴은 /로 끝납니다. /는 나누기와 주석에도 사용되기 때문에 주의가 필요합니다.

RegExp에는 3가지 플래그가 있습니다. 3가지 플래그는 [표 7-1]에 나오는 것처럼 g, i, m이라는 세가지 문자로 표시됩니다. 플래그는 정규 표현식 리터럴 뒤에 바로 붙습니다.

```
// 자바스크립트의 문자열에 일치하는 정규 표현식 객체 생성
```

```
var my_regex = "/(?:\.\|[^\\\"])*"/g;
```

3. 역자 주/ 이후 RegExp와 정규 표현식 객체라는 단어를 혼용해서 사용하겠습니다. 두 가지를 같은 의미로 인지하면 됩니다.

(표 7-1) 정규 표현식 플래그

플래그	설명
g	Global(여러 번 일치함. 정확한 의미는 메소드에 따라 다름)
i	Insenstive (대소문자를 구분하지 않음)
m	Multiline(^과 \$이 라인 끝 문자에 일치할 수 있음)

정규 표현식을 만드는 또 하나의 방법은 RegExp 생성자를 사용하는 것입니다. 이 생성자는 문자열을 받아서 RegExp 객체로 컴파일합니다. 문자열을 구성할 때는 약간의 주의가 필요한데 역슬래시가 정규 표현식 리터럴과는 조금 다른 의미를 지니기 때문입니다. 보통 두 개의 역슬래시가 필요하면 따옴표도 이스케이프시켜야 합니다.

// 자바스크립트의 문자열에 일치하는 정규 표현식 객체 생성

두 번째 매개변수는 플래그를 지정하는 문자열입니다. RegExp 생성자는 프로그래밍 시에는 알 수 없고 실행시간에만 알 수 있는 자료를 가지고 정규 표현식을 생성해야 할 때 유용합니다.

RegExp 객체는 [표 7-2]에 나오는 속성들이 있습니다.

[표 7-2] RegExp 객체의 속성

속성	설명
global	g 플래그가 사용된 경우 true
ignoreCase	i 플래그가 사용된 경우 true
lastIndex	다음 exec 실행을 위한 시작점을 나타냄. 초기값은 0
multiline	m 플래그가 사용된 경우 true
source	정규 표현식의 소스 텍스트

정규 표현식 리터럴로 만들어진 RegExp 객체는 다음과 같이 하나의 인스턴스를 공유합니다.

```
function make_a_matcher( ) {
    return /a/gi;
}

var x = make_a_matcher( );
var y = make_a_matcher( );

// x와 y는 같은 객체입니다.

x.lastIndex = 10;

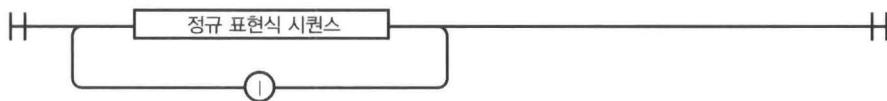
document.writeln(y.lastIndex); // 10
```

## 03 | 구성요소

이제 정규 표현식을 구성하는 요소들을 좀더 자세히 살펴보겠습니다.

### 선택

정규 표현식 선택

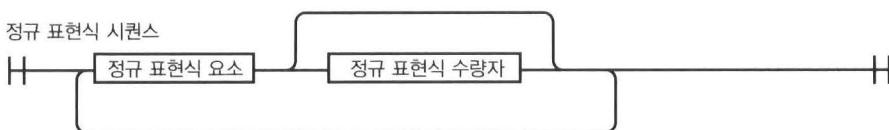


정규 표현식의 선택에는 하나 이상의 정규 표현식 시퀀스가 포함됩니다. 각 시퀀스는 | 문자로 구분됩니다. 이렇게 구분된 각 시퀀스가 하나라도 맞으면 일치하는 것입니다. 그러므로 선택 구문은 각 시퀀스 순서대로 하나씩 일치하는지 대조해보고 일치하는 것을 만나면 뒤에 시퀀스가 더 남았더라도 그것을 일치하는 것으로 봅니다.

다. 그래서 다음의 예제는 into에서 in이 일치하게 됩니다. int도 일치는 하지만 in에서 일치하는 것을 찾았기 때문에 in이 일치하는 것이 됩니다.

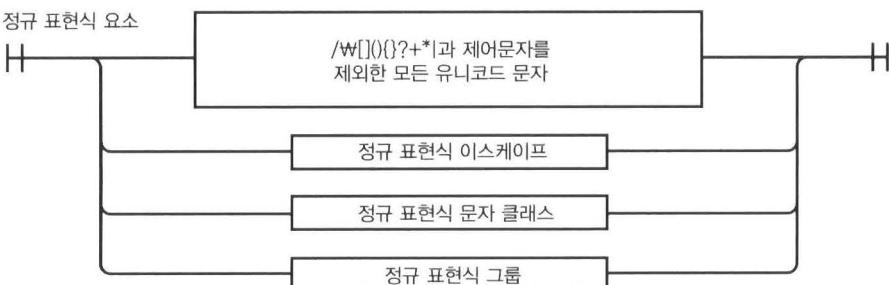
```
"into".match(/in|int/)
```

## 시퀀스



정규 표현식 시퀀스는 하나 이상의 정규 표현식 요소를 포함합니다. 각각의 요소 뒤에는 정규 표현식 요소가 몇 번 반복해서 나오는지를 나타내는 수량자(\*, + 등)를 선택적으로 붙일 수 있습니다. 수량자(quantifier)가 없다면 그 요소는 정확히 한 번 일치하는 것입니다.

## 정규 표현식 요소



정규 표현식 요소는 문자, 팔호로 묶인 그룹, 문자 클래스, 이스케이프 시퀀스 등이 될 수 있습니다. 제어문자나 다음과 같은 특별한 문자를 제외한 모든 문자는 리터럴로 다뤄집니다.

\ / [ ] ( ) { } ? + \* | . ^ \$

이 문자들이 리터럴로 다뤄지게 하려면 앞에 \를 붙여 이스케이프시켜야 합니다. 의심이 가는 경우 원하는 특수 문자 앞에 \를 붙여 리터럴로 만들 수 있습니다. 일반 숫자나 문자 앞에는 \를 붙여봤자 아무 효용이 없습니다.

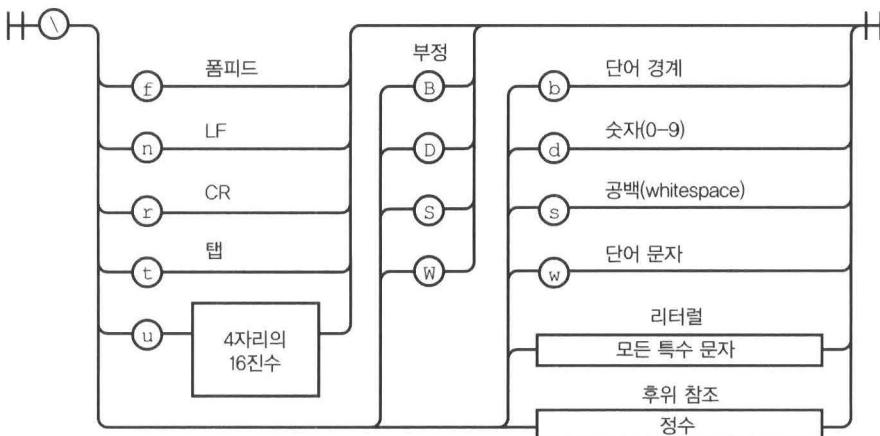
이스케이프되지 않은 .(마침표)는 줄 끝 문자를 제외한 모든 문자에 일치합니다.

이스케이프되지 않은 ^는 lastIndex 속성이 0일 때 텍스트의 시작에 대응합니다. 또한 m 플래그가 지정된 경우 라인 끝 문자에도 일치합니다.

이스케이프되지 않은 \$는 텍스트의 끝과 일치합니다. 또한 m 플래그가 지정된 경우 라인 끝 문자에도 일치합니다.

## 이스케이프

정규 표현식 이스케이프



역슬래시(\)는 정규 표현식 요소와 문자열에서 모두 이스케이프됐다는 것을 나타냅니다. 하지만 정규 표현식 요소에서는 조금 다른 점이 있습니다.

문자열과 마찬가지로 정규 표현식 요소에서도 \f는 폼피드, \n는 LF(라인 피드), \r는 CR(캐리지 리턴), \t는 탭을 의미하고, \u는 4자리의 16진수로 유니코드 문자를 지정할 때 사용합니다. 하지만 \b는 역스페이스 문자가 아닙니다.

\d는 [0-9]와 같으며 아라비아 숫자에 일치합니다. \D는 그 반대로 [^0-9]를 의미합니다.

\s는 [\f\n\r\t\u000B\u0020\u00A0\u2028\u2029]와 같습니다(즉 공백 문자입니다). 이는 유니코드 공백 문자 집합의 일부분입니다. \S는 반대로 [^\f\n\r\t\u000B\u0020\u00A0\u2028\u2029]를 의미합니다.

\w는 [0-9A-Z\_a-z]와 같고 그 반대인 \W는 [^0-9A-Z\_a-z]를 의미합니다. 이 이스케이프 문자는 단어를 나타나는 문자를 표현하기 위해서 정의된 거 같습니다. 하지만 불행하게도 이 클래스는 실제 언어를 다루는데는 그다지 쓸모 있지 않습니다. 만약 이런 기능을 하는 클래스가 필요하다면 자신이 직접 만들어 써야 합니다.

간단한 문자 클래스는 [A-Za-z\u00C0-\u1FFF\u2800-\uFFFD]과 같이 정의할 수 있습니다. 이 클래스는 모든 유니코드 문자를 포함합니다. 하지만 여기에는 문자가 아닌 수많은 기호 또한 포함됩니다. 유니코드는 방대하고 복잡합니다. BMP(Basic Multilingual Plane)에 해당하는 문자 클래스를 추출할 수는 있지만 이것은 상당히 방대하고 비효율적입니다. 자바스크립트의 정규식은 국제화(internationalization)와 관련해서는 상당히 빈약하게 지원하고 있습니다.

\b는 단어 경계를 나타내어 텍스트를 문자 기반으로 일치시키기 쉽게 하기 위해서 고안된 것입니다. 하지만 애석하게도 단어 경계를 위해 \w에 해당하는 규칙이 사용되기 때문에 다중 언어 애플리케이션에서는 무용지물입니다.

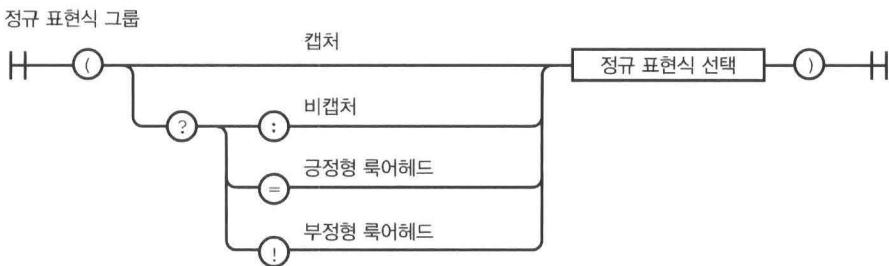
\1은 첫 번째 그룹에 캡처된 텍스트에 대한 참조입니다. 그래서 \1은 일치하는 것을 위해 다시 사용될 수 있습니다. 예를 들어 중복된 단어를 찾고자 하는 경우 다음과 같이 정규 표현식을 작성할 수 있습니다.

```
var doubled_words =  
/([A-Za-z\u00C0-\u1FFF\u2800-\uFFFD'\-]+)\s+\1/gi;
```

doubled\_words는 단어(하나 이상의 문자를 포함하는 문자열) 다음에 공백 문자가 나오고 다시 같은 단어가 나오는 경우를 찾습니다.

\2는 두 번째 그룹에 대한 참조, \3는 세 번째 이런 식으로 이어집니다.

## 그룹



다음과 같은 네 가지 그룹이 있습니다.

### 캡처

캡처 그룹은 팔호로 묶인 정규 표현식 선택입니다. 그룹에 일치하는 문자들은 캡처됩니다. 모든 캡처 그룹에는 번호가 주어지는데 첫 번째 캡처 그룹이 1, 그 다음 그룹이 2 이런 식으로 주어집니다.

### 비캡처

비캡처 그룹은 (? : 접두어가 있습니다. 비캡처 그룹은 단순히 일치하는지를 확인할 뿐이지 일치하는 텍스트를 캡처하지는 않습니다. 그래서 성능이 약간 더 빨라지는 이점이 있습니다. 비캡처 그룹은 캡처 그룹의 그룹 숫자에 영향을 미치지 않습니다.

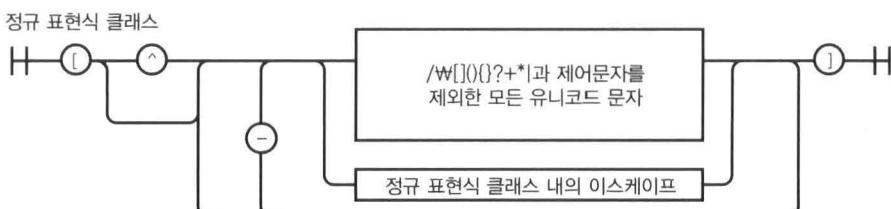
## 긍정형 룩어헤드(positive lookahead)

긍정형 룩어헤드 그룹은  $(?=$  접두어를 갖습니다. 이 그룹은 비캡처 그룹과 유사한데 다만 일치되는 부분을 찾은 후에 텍스트를 그룹이 시작하는 지점으로 다시 돌립니다. 즉 실질적으로 찾는 것이 아니라고 볼 수 있습니다. 긍정형 룩어헤드는 좋은 점이 아닙니다.

## 부정형 룩어헤드(negative lookahead)

부정형 룩어헤드 그룹은  $(?!$  접두어를 갖습니다. 이 그룹은 긍정형 룩어헤드와 비슷한데 다만, 찾지 못했을 때가 일치하는 경우입니다. 부정형 룩어헤드도 좋은 점이 아닙니다.<sup>4</sup>

## 클래스



클래스는 특정 문자 집합 하나를 지정하는데 편리한 방법입니다. 예를 들어, 알파벳 모음에 일치하게 하려면,  $(?:a|e|i|o|u)$ 라고 작성하면 되지만 좀더 편리하게 [aeiou]라는 클래스를 사용할 수 있습니다.

클래스는 두 가지의 편리함을 제공합니다. 첫 번째는 문자의 범위를 지정할 수 있다는 것입니다. 그래서 다음과 같은 32개의 ASCII 특수 문자를,

! " # \$ % & ' ( ) \* +, - . / :  
; < = > ? @ [ \ ] ^ \_ ` { | }

4. 역자 주 / 필자는 룩어헤드에 대해서 좋은 점이 아니라고 하면서 자세한 설명은 하고 있지 않습니다. 그러므로 이에 대한 자세한 설명이 필요하신 분은 검색을 하거나 「정규 표현식 완전 해부와 실습(개정판)」(한빛미디어, 2003)을 참조하기 바랍니다.

다음과 같이 나타낼 수도 있지만,

```
(?:!|"|#|\$|%&|'|\\(|\)|\*|\+|,|-|\.|\;/:|;|<|=|>|@|\[|\\\\]|\\^|_|`|\\{|\||\}|)
```

좀 더 작성하기 쉽게 다음과 같이 나타낼 수 있습니다.

```
[ !-`\/:-\@\[-`{-]
```

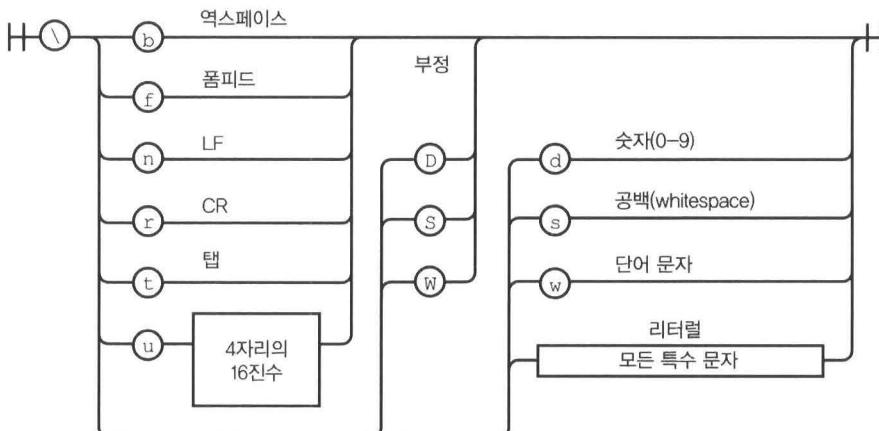
즉 !에서 / 문자까지, :에서 @문자까지, [에서 ` 문자까지, { 이후 문자까지를 포함하게 작성할 수 있습니다. 여전히 좀 복잡해 보이기는 합니다.

두 번째 편리함은 부정형 클래스입니다. [ 다음에 ^를 붙이면 클래스는 지정된 문자들을 배제합니다.

그래서 [^!-`\/:-\@\[-`{-]는 ASCII 특수 문자에 해당하지 않는 문자 하나를 의미합니다.

## 클래스 내의 이스케이프

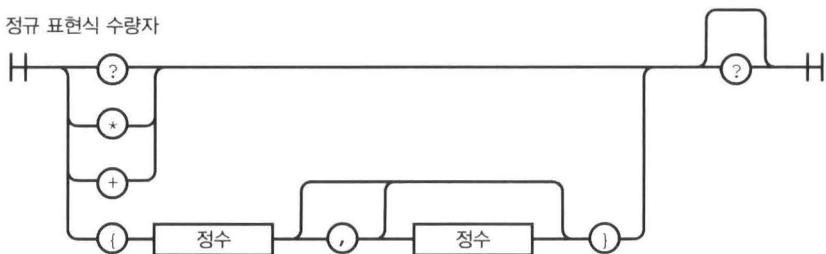
정규 표현식 클래스 내의 이스케이프



문자 클래스에서 이스케이프하는 방법은 정규 표현식 요소에서 하던 방법과는 조금 다른 점이 있는데, [\\b]가 단어 경계가 아니고, 역스페이스 문자입니다. 다음은 문자 클래스에서 이스케이프되어야 하는 특수문자입니다.

- / [ \ ] ^

## 수량자(Quantifier)



정규 표현식 요소에는 요소의 뒤에 붙어 요소가 몇 번 일치해야 하는지를 결정하는 수량자(quantifier)가 있습니다. 중괄호로 묶인 숫자는 표현식 요소가 일치하는 횟수를 의미합니다. 그래서 /www/는 /w{3}/과 같습니다. {3, 6}은 3번에서 6번 일치한다는 의미입니다. {3,}는 3번 이상을 의미합니다.

?는 {0,1}과 같습니다. \*는 {0,}과 그리고 +는 {1,}과 같은 의미입니다.

수량자를 하나만 사용하면 일치하는 부분을 최대한 많이 찾으려고 합니다. 반면에 수량자 뒤에 ?가 붙으면 일치하는 부분을 찾는 일이 나태해지는 경향이 있어서 가능한 적게 일치하는 부분을 찾으려고 합니다. 일반적으로 ?를 붙이지 않는 방법을 사용하는 것이 좋은 방법입니다.<sup>5</sup>

5. 역자 주 / ?를 붙이는 것과 관련하여 좀더 설명을 하겠습니다. 문자열 "aaa"에 대해서 정규 표현식 /a\*/와 /a\*?/의 차이점은 /a\*/는 최대한 많이 반복되는 부분을 찾기 때문에 aaa를 찾지만 /a\*?/는 최소한의 반복만을 찾기 때문에 처음 한 번 반복되는 a만을 찾습니다. 그리고 어떤 경우에는 원하지 않는 방향으로 동작하기도 합니다. 그러므로 저자가 얘기한 것처럼 특별한 경우가 아니라면 수량자 뒤에 ?를 붙이는 것은 사용하지 않는 것이 좋습니다.





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

08

## 메소드

미치긴 미쳤는데 말에 조리(method)가 있거든.

— 윌리엄 셰익스피어, 햄릿



자바스크립트는 표준 데이터 타입에 사용할 수 있는 작은 규모의 표준 메소드 집합이 있습니다.

---

### 배열

#### array.concat(item...)

concat 메소드는 자신의 복사본에 인수로 넘어온 값들을 추가한 새로운 배열을 반환합니다. 인수로 넘어온 값이 배열이면 각각의 요소를 개별적으로 새로운 배열에 추가합니다. 이 장의 뒤에 나오는 array.push(item...)도 참조하기 바랍니다.

```
var a = ['a', 'b', 'c'];
var b = ['x', 'y', 'z'];
var c = a.concat(b, true);
// c는 ['a', 'b', 'c', 'x', 'y', 'z', true]
```

## array.join (구분자)

join 메소드는 배열로 문자열을 만듭니다. 문자열을 만드는 과정은 일단 배열의 모든 요소를 각각 문자열로 만든 후에 이들을 각 항목 사이에 구분자를 붙여서 하나로 합치는 것입니다. 기본 구분자는 ‘,’입니다. 구분 없이 연결하고 싶은 경우에는 구분자로 빈 문자열을 넘기면 됩니다.

만약 많은 수의 문자열을 하나로 합쳐야 한다면 각각의 문자열을 배열에 담은 후 join 메소드를 사용하는 것이 + 연산자로 연결하는 것보다 빠릅니다.<sup>1</sup>

```
var a = ['a', 'b', 'c'];
a.push('d');
var c = a.join(''); // c는 'abcd'
```

## array.pop()

pop과 push 메소드는 배열을 스택처럼 동작하게 합니다. pop 메소드는 배열에서 마지막 요소를 제거하고 제거한 요소를 반환해줍니다. 만약 빈 배열일 경우 undefined를 반환합니다.

```
var a = ['a', 'b', 'c'];
var c = a.pop(); // a는 ['a', 'b'], c는 'c'
```

pop 메소드는 다음과 같이 구현할 수 있습니다.

```
Array.method('pop', function () {
    return this.splice(this.length - 1, 1)[0];
});
```

---

1. 역사 주/ 브라우저를 예를 들면 이런 문제는 IE에서 특히 두드러집니다. FireFox나 Opera, Safari 등에서는 두 방법의 속도 차이가 거의 없거나 경우에 따라 반대의 결과가 나오는 경우도 있습니다. 책의 예제 소스 모음에 stringbuffer.html을 추가했습니다. 한빛미디어 사이트에서 내려받은 후 브라우저 별로 테스트하면 그 차이를 확인할 수 있습니다.

## **array.push(item...)**

push 메소드는 인수로 넘어온 항목을 배열의 끝에 추가합니다. concat 메소드와 다르게 이 메소드는 배열 자체를 수정하여 넘어온 인수 전체를 배열에 추가합니다. 반환값은 배열의 새로운 length 값입니다.

```
var a = ['a', 'b', 'c'];
var b = ['x', 'y', 'z'];
var c = a.push(b, true);
// a는 ['a', 'b', 'c', ['x', 'y', 'z']], true
// c는 5
```

push 메소드는 다음과 같이 구현할 수 있습니다.

```
Array.method('push', function () {
    this.splice.apply(
        this,
        [this.length, 0].
            concat(Array.prototype.slice.apply(arguments)));
    return this.length;
});
```

## **array.reverse( )**

reverse 메소드는 배열 요소의 순서를 반대로 변경합니다. 반환값은 배열입니다.

```
var a = ['a', 'b', 'c'];
var b = a.reverse();
// a, b 모두 ['c', 'b', 'a']
```

## array.shift( )

shift 메소드는 배열에서 첫 번째 요소를 제거하고 제거한 요소를 반환합니다. 빈 배열일 경우 undefined를 반환합니다. shift는 보통 pop보다 많이 느립니다.

```
var a = ['a', 'b', 'c'];
var c = a.shift(); // a는 ['b', 'c'], c는 'a'
```

shift는 다음과 같이 구현할 수 있습니다.

```
Array.method('shift', function () {
    return this.splice(0, 1)[0];
});
```

## array.slice(start, end)

slice 메소드는 배열의 특정 부분에 대한 복사본을 만듭니다. 복사되는 첫 번째 요소는 매개변수 start에 해당하는 첨자를 갖는 배열 요소입니다. 그리고 매개변수 end에 해당하는 첨자를 가진 요소 전까지 복사됩니다. end 매개변수는 옵션이며 지정하지 않을 경우 기본값은 배열의 length 속성값입니다. 매개변수가 음수인 경우 이들을 양수로 만들기 위해 length 값이 더해집니다. 만약 시작값이 length 값보다 크거나 같을 경우 빈 배열을 반환합니다. slice와 splice를 혼동해서는 안 됩니다. 그리고 문자열의 메소드인 .slice와도 구분해야 합니다. 문자열의 slice 메소드는 이 장의 뒤에서 살펴볼 것입니다.

```
var a = ['a', 'b', 'c'];
var b = a.slice(0, 1); // b는 ['a']
var c = a.slice(1); // c는 ['b', 'c']
var d = a.slice(1, 2); // d는 ['b']
```

## array.sort (비교 함수)

sort 메소드는 배열의 내용을 적절하게 정렬합니다. 이 메소드는 숫자들의 배열을 제대로 정렬하지 못하는 문제가 있는데, 다음의 예를 보기 바랍니다.

```
var n = [4, 8, 15, 16, 23, 42];
n.sort();
// n은 [15, 16, 23, 4, 42, 8]
```

자바스크립트의 기본 비교 함수는 정렬될 배열 요소들을 문자열로 간주합니다. 이 메소드는 비교를 하기 전에 배열 요소의 타입을 확인할 정도로 똑똑하지 못합니다. 그래서, 그냥 숫자를 문자열로 변경한 후에 비교하기 때문에 원하지 않는 이상한 결과를 보이는 것입니다.

다행인 것은 비교 함수를 바꿔서 사용할 수 있다는 것입니다. 직접 지정하는 비교 함수는 매개변수 두 개를 취하여, 두 매개변수가 같은 경우에는 0을, 첫 번째 것이 먼저 와야 하는 경우에는 음수를, 두 번째 것이 먼저 와야 하는 경우에는 양수를 반환해야 합니다(연륜이 있는 프로그래머라면 아마 FORTRAN II의 수치적 IF문이 생각날 것입니다).

```
n.sort(function (a, b) {
    return a - b;
});
// n은 [4, 8, 15, 16, 23, 42]
```

이 비교 함수는 숫자값만을 비교할 수 있지 문자열은 비교하지 못합니다. 하나가 아닌 여러 종류의 기본 데이터 타입을 가진 배열을 정렬하고 싶다면 다음과 같이 좀더 많은 작업이 필요합니다.

```

var m = ['aa', 'bb', 'a', 4, 8, 15, 16, 23, 42];
m.sort(function (a, b) {
    if (a === b) {
        return 0;
    }
    if (typeof a === typeof b) {
        return a < b ? -1 : 1;
    }
    return typeof a < typeof b ? -1 : 1;
});
// m은 [4, 8, 15, 16, 23, 42, 'a', 'aa', 'bb']

```

만약 대소문자를 구분하지 않는다면 비교 전에 피연산자를 소문자로 변환해야 합니다. 추가적으로 뒤에 나오는 문자열 메소드 중 `.localeCompare` 메소드를 살펴보기 바랍니다.

좀더 똑똑한 비교 함수를 사용하면 객체 배열도 정렬할 수 있습니다. 일반적인 경우에 보다 쉽게 적용할 수 있도록 비교 함수를 만들어주는 함수를 작성해 보겠습니다.

```

// 객체의 속성 이름을 문자열로 받고 이것과 같은 이름의 속성을 포함하는
// 객체들의 배열을 그 속성의 값으로 정렬하는
// 함수를 반환하는 함수

var by = function (name) {
    return function (o, p) {
        var a, b;
        if (typeof o === 'object' && typeof p === 'object' && o && p) {
            a = o[name];
            b = p[name];
            if (a === b) {
                return 0;
            }
            if (typeof a === typeof b) {
                return a < b ? -1 : 1;
            }
        }
    }
}

```

```

        return typeof a < typeof b ? -1 : 1;
    } else {
        throw {
            name: 'Error',
            message: 'Expected an object when sorting by ' + name
        };
    }
};

var s = [
    {first: 'Joe', last: 'Besser'},
    {first: 'Moe', last: 'Howard'},
    {first: 'Joe', last: 'DeRita'},
    {first: 'Shemp', last: 'Howard'},
    {first: 'Larry', last: 'Fine'},
    {first: 'Curly', last: 'Howard'}
];
s.sort(by('first'));
// s는 [
//     {first: 'Curly', last: 'Howard'},
//     {first: 'Joe', last: 'DeRita'},
//     {first: 'Joe', last: 'Besser'},
//     {first: 'Larry', last: 'Fine'},
//     {first: 'Moe', last: 'Howard'},
//     {first: 'Shemp', last: 'Howard'}
// ]

```

sort 메소드는 안정적이지 않습니다.

```
s.sort(by('first')).sort(by('last'));
```

그래서 위와 같은 실행문은 정확한 실행 흐름에 의한 결과를 보장하지 않습니다. 만약 다중 정렬을 원한다면 추가 작업이 더 필요합니다. 즉 by 함수에다가 첫 번째 정렬 요소로 정렬할 때 두 값이 같은 경우에, 적용할 비교 함수를 두 번째 매개변수로 받게 수정하는 것입니다. 이렇게 함으로써 다중 정렬을 할 수 있습니다.

```

// 객체의 속성 이름을 문자열로 받고 이것과 같은 이름의 속성을 포함하는
// 객체들의 배열을 그 속성의 값으로 정렬하는
// 함수를 반환하는 함수. 옵션으로 minor라는 비교 함수를
// 받아 비교하는 두 값이 같은 경우 이 함수를 적용.
// 즉, o[name]과 p[name]이 같으면 minor 함수 적용.

var by = function (name, minor) {
    return function (o, p) {
        var a, b;
        if (o && p && typeof o == 'object' && typeof p == 'object') {
            a = o[name];
            b = p[name];
            if (a === b) {
                return typeof minor == 'function' ? minor(o, p) : 0;
            }
            if (typeof a === typeof b) {
                return a < b ? -1 : 1;
            }
            return typeof a < typeof b ? -1 : 1;
        } else {
            throw {
                name: 'Error',
                message: 'Expected an object when sorting by ' + name
            };
        }
    };
};

s.sort(by('last', by('first')));      // s는 [
//      {first: 'Joe', last: 'Besser'},
//      {first: 'Joe', last: 'DeRita'},
//      {first: 'Larry', last: 'Fine'},
//      {first: 'Curly', last: 'Howard'},
//      {first: 'Moe', last: 'Howard'},
//      {first: 'Shemp', last: 'Howard'}
// ]

```

## array.splice(start, deleteCount, item...)

splice 메소드는 기존의 배열 요소들을 제거하고 그 부분을 새로운 항목으로 대체 합니다. start 매개변수는 배열에서 위치를 나타내는 수입니다. deleteCount 매개변수는 시작점부터 삭제할 요소의 수를 나타냅니다. 이 외에 추가적인 매개변수가 있는 경우 이 매개변수들(item...)은 삭제한 위치에 추가됩니다. 이 메소드는 삭제된 요소들을 가진 배열을 반환합니다.

splice의 가장 일반적인 사용은 배열에서 요소들을 삭제할 때입니다. splice와 slice를 혼동해서는 안 됩니다.

```
var a = ['a', 'b', 'c'];
var r = a.splice(1, 1, 'ache', 'bug');
// a는 ['a', 'ache', 'bug', 'c']
// r은 ['b']
```

splice는 다음과 같이 구현할 수 있습니다.

```
Array.method('splice', function (start, deleteCount) {
    var max = Math.max,
        min = Math.min,
        delta,
        element,
        insertCount = max(arguments.length - 2, 0),
        k = 0,
        len = this.length,
        new_len,
        result = [],
        shift_count;

    start = start || 0;
    if (start < 0) {
        start += len;
    }
    delta = deleteCount - insertCount;
    if (delta > 0) {
        for (var i = start; i < start + delta; i++) {
            result.push(this[i]);
        }
        for (var i = start + delta; i < len; i++) {
            this[i] = undefined;
        }
    } else if (delta < 0) {
        for (var i = start; i < len; i++) {
            result.push(this[i]);
        }
        for (var i = len; i < start + insertCount; i++) {
            this[i] = undefined;
        }
    }
    if (insertCount > 0) {
        for (var i = 0; i < insertCount; i++) {
            this[start + i] = arguments[i + 2];
        }
    }
    return result;
})
```

```
start = max(min(start, len), 0);
deleteCount = max(min(typeof deleteCount === 'number' ?
    deleteCount : len, len - start), 0);
delta = insertCount - deleteCount;
new_len = len + delta;
while (k < deleteCount) {
    element = this[start + k];
    if (element !== undefined) {
        result[k] = element;
    }
    k += 1;
}
shift_count = len - start - deleteCount;
if (delta < 0) {
    k = start + insertCount;
    while (shift_count) {
        this[k] = this[k - delta];
        k += 1;
        shift_count -= 1;
    }
    this.length = new_len;
} else if (delta > 0) {
    k = 1;
    while (shift_count) {
        this[new_len - k] = this[len - k];
        k += 1;
        shift_count -= 1;
    }
}
for (k = 0; k < insertCount; k += 1) {
    this[start + k] = arguments[k + 2];
}
return result;
});
```

## array.unshift(item...)

unshift 메소드는 항목들을 배열의 뒤가 아니라 앞에다가 추가한다는 점만 빼고 push 메소드와 같습니다. 이 메소드는 배열의 새로운 length 값을 반환합니다.<sup>2</sup>

```
var a = ['a', 'b', 'c'];
var r = a.unshift('?', '@');
// a는 ['?', '@', 'a', 'b', 'c']
// r은 5
```

unshift는 다음과 같이 구현할 수 있습니다.

```
Array.method('unshift', function ( ) {
    this.splice.apply(this,
        [0, 0].concat(Array.prototype.slice.apply(arguments)));
    return this.length;
});
```

---

## 함수

### function.apply(thisArg, argArray)

apply 메소드는 this에 연결될 객체(thisArg)와 (옵션인) 인수 배열(argArray)을 넘기면서 함수를 호출합니다. apply 메소드는 apply 호출 패턴에서 사용합니다(4장 참고).

```
Function.method('bind', function (that) {
    // 특정 함수를 that 객체의 메소드처럼 호출하는 함수 반환
```

---

2. 역사 주 / 참고로 IE에서는 이 메소드를 제대로 구현하지 못했는지 반환값으로 length 값을 반환하는 것이 아니라 undefined를 반환합니다. 이후 IE8 정식 버전부터 수정될 여지는 있지만 어쨌든 IE에서 unshift를 사용할 때는 이 부분에 주의해야 합니다.

```

var method = this,
    slice = Array.prototype.slice,
    args = slice.apply(arguments, [1]);
return function ( ) {
    return method.apply(that,
        args.concat(slice.apply(arguments, [0])));
};
}) ;

var x = function ( ) {
    return this.value;
}.bind({value: 666});
alert(x( )); // 666

```

## 숫자(Number)

### **number.toExponential(fractionDigits)**

toExponential 메소드는 숫자를 지수 형태의 문자열로 변환합니다. 옵션인 fractionDigits 매개변수는 소수점 아래 몇 째 자리까지 표시할 것인지를 지정합니다. 이 값은 0에서 20사이의 값이어야 합니다.

```

document.writeln(Math.PI.toExponential(0));
document.writeln(Math.PI.toExponential(2));
document.writeln(Math.PI.toExponential(7));
document.writeln(Math.PI.toExponential(16));
document.writeln(Math.PI.toExponential(  ));

// 결과

3e+0
3.14e+0
3.1415927e+0
3.1415926535897930e+0
3.141592653589793e+0

```

## **number.toFixed(fractionDigits)**

toFixed 메소드는 숫자를 고정 소수점 형태로 변환합니다. 옵션인 fractionDigits 매개변수는 소수점 아래 몇 째 자리까지 표시할 것인지를 지정합니다. 이 값은 0에서 20사이의 값이어야 하며 기본값은 0입니다.

```
document.writeln(Math.PI.toFixed(0));
document.writeln(Math.PI.toFixed(2));
document.writeln(Math.PI.toFixed(7));
document.writeln(Math.PI.toFixed(16));
document.writeln(Math.PI.toFixed( ));

// 결과
3
3.14
3.1415927
3.1415926535897930
3
```

## **number.toPrecision(precision)**

toPrecision 메소드는 숫자를 10진수 형태의 문자열로 변환합니다. 옵션인 precision 매개변수는 반환되는 문자열에 포함된 숫자의 개수입니다. 이 값은 1에서 21사이입니다.

```
document.writeln(Math.PI.toPrecision(2));
document.writeln(Math.PI.toPrecision(7));
document.writeln(Math.PI.toPrecision(16));
document.writeln(Math.PI.toPrecision( ));

// 결과
3.1
3.141593
3.141592653589793
3.141592653589793
```

## **number.toString(radix)**

toString 메소드는 숫자를 문자열로 변환합니다. 옵션인 radix 매개변수는 기수(또는 진법)를 지정합니다. 이 값은 2에서 36사이의 값이어야 합니다. radix의 기본값은 10입니다. 보통 정수형에 radix 매개변수가 가장 많이 사용되지만, radix는 어떤 숫자에도 적용할 수 있습니다.

대부분의 경우 숫자.toString()은 보다 간단하게 String(숫자)로 작성할 수 있습니다.

```
document.writeln(Math.PI.toString(2));
document.writeln(Math.PI.toString(8));
document.writeln(Math.PI.toString(16));
document.writeln(Math.PI.toString(  ));

// 결과

11.001001000011111011010100010001000010110100011
3.1103755242102643
3.243f6a8885a3
3.141592653589793
```

---

## 객체

### **object.hasOwnProperty(name)**

hasOwnProperty 메소드는 객체가 매개변수 name과 같은 이름의 속성이 있으면 true를 반환하고 그렇지 않으면 false를 반환합니다. 해당 이름이 프로토타입 체인(prototype chain) 상에 있는지는 확인하지 않습니다. 이 메소드는 name의 값이 hasOwnProperty일 경우에는 무용지물입니다.

```
var a = {member: true};  
var b = Object.create(a); // 3장에서 설명한 메소드  
var t = a.hasOwnProperty('member'); // t는 true  
var u = b.hasOwnProperty('member'); // u는 false  
var v = b.member; // v는 true
```

---

## 정규 표현식 객체(RegExp)

### regexp.exec(string)

exec 메소드는 정규 표현식을 사용하는 메소드들 중에서 가장 강력한(가장 느리기도 한) 메소드입니다. 이 메소드는 regexp를 string에 적용해서 일치하는 경우 배열을 반환합니다. 배열의 첫 번째 요소(첨자 0)는 regexp에 일치하는 문자열을 포함하고, 두 번째 요소(첨자 1)부터는 그룹 1에 캡처된 텍스트, 세 번째 요소는 두 번째 그룹 식으로 이어집니다. 만약 일치하지 않으면 null을 반환합니다.

정규 표현식 객체(regexp)가 g 플래그를 가진 경우 조금 더 복잡해집니다. 검색의 시작은 문자열의 0번째 위치부터 시작하는 것이 아니라 regexp.lastIndex 값의 위치부터 시작합니다(초기값은 0). 일치하는 것을 찾게 되면 regexp.lastIndex 값은 일치하는 부분 다음에 나오는 첫 글자의 위치로 설정됩니다. 일치하는 것을 찾지 못하면 regexp.lastIndex 값은 0으로 재설정됩니다.

이러한 원리를 이용하여 반복적으로 exec를 호출하여 문자열에 포함된 패턴과 일치하는 부분을 모두 찾을 수 있습니다. 그런데 이와 관련하여 주의해야 할 사항이 있습니다. 루프를 다 돌기 전에 일찍 빠져나가는 경우 다시 루프를 돌아 전부 찾으려고 할 때는 먼저 regexp.lastIndex 값을 0으로 설정해야 합니다. 그리고 ^ 부분은 regexp.lastIndex 값이 0일 때만 일치됩니다.<sup>3</sup>

---

3. 역자 주/ 이 장의 뒷 부분에 나오는 String.match 메소드에 대한 설명을 보고 g 플래그를 설정했을 때 어떻게 다른지 비교해서 알아둘 필요가 있습니다.

```

// 간단한 html을 태그와 텍스트로 분리.
// (entityify는 string.replace 부분 참조)

// 각각의 태그나 텍스트를 찾았을 때 반환되는 배열의 구조
// [0] 태그나 텍스트 전체
// [1] /가 있는 경우 /
// [2] 태그 이름
// [3] 태그 속성이 있다면 태그 속성

var text = '<html><body bgcolor=linen><p>' +
    'This is <b>bold</b>!</p></body></html>';
var tags = /[^<>]+|<(/?) ([A-Za-z]+) ([^<>]*)>/g;
var a, i;

while ((a = tags.exec(text))) {
    for (i = 0; i < a.length; i += 1) {
        document.writeln('// [' + i + '] ' + a[i]).entityify( ));
    }
    document.writeln( );
}

// 결과

// [0] <html>
// [1]
// [2] html
// [3]

// [0] <body bgcolor=linen>
// [1]
// [2] body
// [3] bgcolor=linen

// [0] <p>
// [1]
// [2] p
// [3]

// [0] This is

```

```
// [1] undefined
// [2] undefined
// [3] undefined

// [0] <b>
// [1]
// [2] b
// [3]

// [0] bold
// [1] undefined
// [2] undefined
// [3] undefined

// [0] </b>
// [1] /
// [2] b
// [3]

// [0] !
// [1] undefined
// [2] undefined
// [3] undefined

// [0] </p>
// [1] /
// [2] p
// [3]
```

### regexp.test(string)

test 메소드는 정규 표현식을 사용하는 메소드 가운데 가장 간단하고 가장 빠릅니다. regexp가 문자열에 일치하면 true를 반환하고 그렇지 않으면 false를 반환합니다. 이 메소드와 g 플래그는 같이 사용해서는 안 됩니다.

```
var b = /&.+/.test('frank & beans');
// b는 true
```

test는 다음과 같이 구현할 수 있습니다.

```
RegExp.method('test', function (string) {  
    return this.exec(string) !== null;  
});
```

---

## 문자열(String)

### string.charAt(pos)

charAt 메소드는 문자열에서 pos 위치에 있는 문자를 반환합니다. pos의 값이 0보다 작거나 문자열의 .length 값보다 크거나 같으면 이 메소드는 빈 문자열을 반환합니다. 자바스크립트는 문자 데이터 타입이 없습니다. 그러므로 이 메소드의 결과는 문자 하나지만 문자열입니다.

```
var name = 'Curly';  
var initial = name.charAt(0); // 머리 글자는 'C'
```

charAt은 다음과 같이 구현할 수 있습니다.

```
String.method('charAt', function (pos) {  
    return this.slice(pos, pos+1);  
});
```

### string.charCodeAt(pos)

charCodeAt 메소드는 charAt과 같은데 다만 문자열 대신 해당 위치 문자의 코드를 반환하는 것이 다릅니다. pos의 값이 0보다 작거나 문자열의 .length 값보다 크거나 같으면 이 메소드는 NaN을 반환합니다.

```
var name = 'Curly';
var initial = name.charCodeAt(0);      // 머리 글자 코드는 67
```

### **string.concat(string...)**

concat 메소드는 자신과 인수로 넘어온 문자열들을 연결하여 새로운 문자열을 만듭니다. 이 메소드는 더 편리한 + 연산자가 있기 때문에 잘 사용하지 않습니다.

```
var s = 'C'.concat('a', 't');      // s는 'Cat'
```

### **string.indexOf(searchString, position)**

indexOf 메소드는 문자열에서 searchString을 검색합니다. 만약 찾게 되는 경우 첫 번째로 일치하는 문자의 위치를 반환하며 그렇지 않은 경우 -1을 반환합니다. 옵션인 position 매개변수는 검색 시작 위치를 지정합니다.

```
var text = 'Mississippi';
var p = text.indexOf('ss');        // p는 2
p = text.indexOf('ss', 3);        // p는 5
p = text.indexOf('ss', 6);        // p는 -1
```

### **string.lastIndexOf(searchString, position )**

lastIndexOf 메소드는 검색을 문자열의 앞이 아니라 끝에서부터 시작한다는 것을 제외하고는 indexOf와 같습니다.

```
var text = 'Mississippi';
var p = text.lastIndexOf('ss');     // p는 5
p = text.lastIndexOf('ss', 3);     // p는 2
p = text.lastIndexOf('ss', 6);     // p는 5
```

## string.localeCompare(that)

localCompare 메소드는 문자열을 매개변수 that 문자열과 비교합니다. 문자열을 어떻게 비교하는지는 해당 환경의 로케일 정보에 따라 달라집니다. 문자열이 that 문자열보다 적은 경우에는 음수가, 같으면 0이, 크면 양수가 반환됩니다. 이는 array.sort 메소드의 비교 함수와 비슷한 방법입니다.

```
var m = ['AAA', 'A', 'aa', 'a', 'Aa', 'aaa'];
m.sort(function (a, b) {
    return a.localeCompare(b);
});
// m은 (특정 로케일에서) ['a', 'A', 'aa', 'Aa', 'aaa', 'AAA']
```

## string.match(regexp)

match 메소드는 정규 표현식과 일치하는 부분을 문자열에서 찾습니다. 찾는 방법은 g 플래그에 따라 달라지는데 g 플래그가 설정되지 않은 경우에 string.match(regexp) 호출의 반환값은 regexp.exec(string) 호출의 반환값과 같습니다. 그러나 g 플래그가 설정된 경우에는 일치하는 모든 부분을 배열로 반환합니다. 단, 캡처 그룹과 일치하는 부분은 반환하지 않습니다.

```
var text = '<html><body bgcolor=linen><p>' +
    'This is <b>bold</b>!</p></body></html>';
var tags = /^[^<>]+|<(\?)([A-Za-z]+)([^<>]*)>/g;
var a, i;

a = text.match(tags);
for (i = 0; i < a.length; i += 1) {
    document.writeln('// [' + i + '] ' + a[i]).entityify());
}

// 결과

// [0] <html>
```

```
// [1] <body bgcolor=linen>
// [2] <p>
// [3] This is
// [4] <b>
// [5] bold
// [6] </b>
// [7] !
// [8] </p>
// [9] </body>
// [10] </html>
```

### **string.replace(searchValue, replaceValue)**

replace 메소드는 문자열을 대상으로 검색 및 대체 작업을 실행하여 새로운 문자열을 생성합니다. searchValue는 문자열이거나 정규 표현식 객체입니다. 만약 searchValue가 문자열이면 첫 번째로 찾은 부분만이 교체됩니다. 그래서 다음 예제의 결과는 실망스럽게도 “mother-in\_law”입니다.

```
var result = "mother_in_law".replace('_', '-');
```

searchValue가 정규 표현식이고 g 플래그가 설정된 경우라면 일치하는 모든 부분이 교체됩니다. 하지만 정규 표현식이라 하더라도 g 플래그가 설정되지 않은 경우에는 처음 한번만 교체됩니다.

replaceValue는 문자열이나 함수입니다. replaceValue가 문자열일 때, 문자 \$는 특별한 의미를 지닙니다.

```
// 팔호 안의 3개의 숫자 캡처
```

```
var oldareacode = /\((\d{3})\)\)/g;
var p = '(555) 666-1212'.replace(oldareacode, '$1-');
// p는 '555-666-1212'
```

달러 시퀀스	교체되는 내용
\$\$	\$
\$&	일치하는 텍스트
\$숫자	해당 숫자의 캡처 그룹 텍스트
\$`	일치하는 부분 앞에 있는 텍스트
\$'	일치하는 부분 뒤에 있는 텍스트

replaceValue가 함수면 일치할 때마다 이 함수가 호출되고 이 함수가 반환하는 문자열이 대체되는 텍스트로 사용됩니다. 이 함수에 전달되는 첫 번째 매개변수는 일치되는 텍스트고, 두 번째 매개변수부터는 캡처 그룹에 캡처된 텍스트들입니다.

```
String.method('entityify', function () {
    var character = {
        '<' : '&lt;',
        '>' : '&gt;',
        '&' : '&amp;',
        '"' : '&quot;'
    };

    // replace 메소드를 호출한 결과를 반환하는 string.entityify
    // 메소드 반환. replaceValue 함수는 문자를 character 객체에서 찾아
    // 그 결과를 반환. 보통 이런 방법으로 객체를 사용하는 것이
    // switch 문을 사용하는 것보다 성능이 좋음.

    return function () {
        return this.replace(/(<>&"")/g, function (c) {
            return character[c];
        });
    };
})( );
alert("<&>".entityify()); // &lt;&amp;&gt;
```

## **string.search(regexp)**

search 메소드는 문자열 대신에 정규 표현식 객체를 인수로 받는다는 것을 제외하고는 indexOf 메소드와 같습니다. 이 메소드는 일치하는 내용을 찾을 경우 첫 번째 일치하는 부분의 첫 번째 문자 위치를 반환합니다. 그리고 검색에 실패했을 때는 -1을 반환합니다. g 플래그는 무시되며 검색 시작 위치를 지정하는 매개변수는 없습니다.

```
var text = 'and in it he says "Any damn fool could';
var pos = text.search(/['']/);      // pos는 18
```

## **string.slice(start, end)**

slice 메소드는 문자열의 일부분을 복사하여 새로운 문자열을 만듭니다. start 매개 변수가 음수이면, 이 값에 문자열의 .length 값을 더합니다. end 매개변수는 옵션이며 기본값은 문자열의 .length 값을입니다. 만약 end 매개변수가 음수이면 이 값에 문자열의 .length 값이 더해집니다. end 매개변수의 기본값은 마지막 문자의 위치 + 1입니다. p의 위치부터 n개의 문자를 얻으려면 문자열.slice(p, p + n)과 같이 사용하면 됩니다. 문자열의 substring 메소드와 배열의 .slice 메소드도 참조하기 바랍니다.

```
var text = 'and in it he says "Any damn fool could';
var a = text.slice(18);
// a는 '"Any damn fool could'
var b = text.slice(0, 3);
// b는 'and'
var c = text.slice(-5);
// c는 'could'
var d = text.slice(19, 32);
// d는 'Any damn fool'
```

## **string.split(separator, limit)**

split 메소드는 구분자(separator 매개변수)로 문자열을 쪼개어 각각의 항목이 담긴 배열을 생성합니다. 옵션인 limit는 나눌 개수를 지정합니다. separator 매개변수는 문자열이나 정규 표현식 모두 가능합니다.

구분자로 빈 문자열을 사용하면 문자 하나 하나씩 나뉘어진 배열이 만들어집니다.

```
var digits = '0123456789';
var a = digits.split('', 5);
// a는 ['0', '1', '2', '3', '456789']
```

구분자가 일반 문자열이면 문자열에서 구분자가 있는 곳을 모두 검색하게 되며, 구분자 사이에 있는 텍스트들이 배열에 복사됩니다. 구분자로 정규 표현식 객체가 넘어왔을 때 g 플래그는 무시됩니다.

```
var ip = '192.168.1.0';
var b = ip.split('.');
// b는 ['192', '168', '1', '0']

var c = '|a|b|c|'.split('|');
// c는 ['', 'a', 'b', 'c', ''']

var text = 'last, first ,middle';
var d = text.split(/\s*,\s*/);
// d는 [
//   'last',
//   'first',
//   'middle'
// ]
```

구분자로 정규 표현식을 사용할 때는 주의해야 할 점이 있습니다. 그 중 하나는 캡처 그룹에 캡처되는 텍스트도 결과에 포함된다는 것입니다.

```
var e = text.split(/\s*(,)\s*/);
// e는 [
//   'last',
//   ',',
//   'first',
//   ',',
//   'middle'
// ]
```

일부 환경에서는 구분자로 정규 표현식을 사용하면 결과 배열에 빈 문자열이 포함됩니다.

```
var f = '|a|b|c|'.split(/\|/);
// 어떤 시스템에서 f는 ['a', 'b', 'c']
// 다른 어떤 시스템에서 f는 ['', 'a', 'b', 'c', '']
```

### **string.substring(start, end)**

substring 메소드는 음수로 된 매개변수를 변환하지 않는 점을 빼면 slice 메소드와 같습니다. substring 메소드를 쓰는 것보다는 slice를 사용하는 것이 좋습니다.

### **string.toLocaleLowerCase( )**

toLocaleLowerCase 메소드는 로케일 규칙을 사용하여 문자열을 소문자로 변환한 새로운 문자열을 반환합니다. 이 메소드로 이득을 보는 주된 언어 중 하나는 터키어인데, 왜냐하면 İ를 i 대신 İ로 변환하기 때문입니다.

### **string.toLocaleUpperCase( )**

toLocaleUpperCase 메소드는 로케일 규칙을 사용하여 문자열을 대문자로 변환한 새로운 문자열을 반환합니다. 이 메소드로 이득을 보는 주된 언어 중 하나는 터키어인데, 왜냐하면 i를 I 대신 İ로 변환하기 때문입니다.

### **string.toLowerCase( )**

toLowerCase는 문자열을 소문자로 변환하여 반환합니다.

### **string.toUpperCase( )**

toUpperCase 메소드는 문자열을 대문자로 변환하여 반환합니다.

### **String.fromCharCode(char...)**

fromCharCode 메소드는 인수로 넘어온 숫자값들을 문자열로 변환하여 반환합니다.

```
var a = String.fromCharCode(67, 97, 116);
// a는 'Cat'
```



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

09

## 스타일

어이없게 어리석은 작위(style)를 줄줄 늘어놓는군!

— 윌리엄 셰익스피어, 헨리 6세 제I부



컴퓨터 프로그램은 인간이 만든 가장 복잡한 것 중에 하나입니다.

프로그램은 실질적으로 오류가 없어야 하는 수많은 함수, 문장, 표현식 등으로 구성됩니다. 이러한 프로그램은 구현할 때 의도한 대로 동작하지 않는 경우가 많으며, 그래서 보통 소프트웨어는 실질적인 사용기간 동안 계속해서 수정해야 합니다. 문제가 없는 프로그램을 다른 정상적인 프로그램으로 변경하는 것은 대단히 도전적인 일입니다.

좋은 프로그램은 미래에 수정할 여지가 있다는 것을 감안한(하지만 그에 따라 크게 부담이 되지 않는) 구조를 갖습니다. 또한 좋은 프로그램은 명확한 프리젠테이션을 갖습니다. 프로그램을 잘 표현하면, 이를 명확히 이해할 가능성이 높아지고 그로 인해 변경이나 오류 수정을 성공적으로 할 수 있습니다.

이러한 개념은 모든 프로그래밍 언어에 해당하는 내용이며 특별히 자바스크립트에는 더욱 그렇습니다. 자바스크립트의 느슨한 데이터 타입 체크와 과도한 오류 허용은 컴파일 시에 프로그램에 대한 품질을 보장하지 못합니다. 그러므로 이를 보완하기 위해서는 엄격한 규칙을 정하고 코딩해야 합니다.

자바스크립트에는 좋은 프로그램을 만드는데 위험한 많은 수의 약점과 문제점이 있습니다. 자바스크립트에 있는 최악의 속성들은 반드시 피해야 합니다. 하지만 이것뿐만 아니라 때때로 유용하지만 간간히 위험할 수 있는 속성들 또한 피해야 합니다. 이러한 속성들은 유인성 불법 유해물(attractive nuisances)이며 이러한 점들을 포함으로써 다양한 유형의 잠재적인 오류를 피할 수 있습니다.

소프트웨어가 오랜 기간 동안 가치를 유지할 수 있는 직접적인 부분은 코드의 품질입니다. 프로그램은 라이프타임 전체에 걸쳐서 수많은 손과 눈을 거칩니다. 프로그램이 자신의 구조와 특성을 명확하게 나타낼 수 있으면, 머지 않은 장래에 수정이 가해질 때 프로그램이 망가지는 일은 거의 없습니다.

자바스크립트 코드는 때때로 일반에게 직접 전달이 됩니다. 그러므로 배포해도 문제 없는 수준의 품질과 깔끔함을 항상 유지해야 합니다. 깔끔하고 일관된 프로그램 작성 스타일은 프로그램을 보다 읽기 쉽게 만듭니다.

프로그래머들은 좋은 스타일을 선정하기 위해서 끊임 없이 토의할 수 있습니다. 대부분의 프로그래머는 자신이 현재 사용하고 있는 스타일에 단단히 고착해있습니다. 이런 스타일은 보통 자신이 다녔던 학교나 첫 직장에서 유행하던 스타일입니다. 일부 프로그래머는 프로그래밍 스타일에 전혀 관심이 없음에도 좋은 경력이 있습니다. 이것을 스타일이 별로 중요하지 않다는 증거로 볼 수 있을까요?

프로그래밍 스타일은 작문할 때 스타일이 중요한 것과 같은 이유로 중요하다는 것이 밝혀졌습니다. 좋은 프로그래밍 스타일은 가독성을 높여줍니다.

컴퓨터 프로그램은 때때로 쓰기 전용 매체처럼 여겨지기 때문에 프로그램이 동작하는 한 어떻게 작성했는지는 중요하게 여겨지지 않습니다. 하지만 프로그램이 동작할 가능성은 가독성이 높을 때 놀랄 정도로 높아지는 것으로 밝혀졌으며, 그에 따라 의도한 대로 동작할 가능성도 높아진다는 것이 밝혀졌습니다. 또한 소프트웨어가 실질적으로 사용되는 기간 내내 다양하게 수정되는 것은 소프트웨어의 본성입니다. 프로그램을 잘 읽고 이해할 수 있다면, 이를 문제없이 수정하고 향상시킬 수 있다고 기대할 수 있습니다.

필자는 이 책 전체에 걸쳐 일관된 스타일을 사용했습니다. 필자의 의도는 예제 코드를 가능한 쉽게 읽을 수 있도록 하는 것이었습니다. 예제 프로그램의 의미를 이해하는데 필요한 단서를 더 많이 제공하기 위해서 일관되게 공백 whitespace)을 사용했습니다.

블록이나 객체 리터럴의 내용은 4칸 들여쓰기를 했습니다. 그리고 if와 ( 사이에는 빈 칸을 두어 if가 함수를 호출하는 것처럼 보이지 않게 했습니다. 반드시 호출하는 경우에만 앞선 심볼과 (를 붙여서 작성했습니다. .(마침표)와 [를 제외한 중위 연산자 앞뒤로는 빈칸을 붙였으며, 모든 쉼표와 콜론 다음에는 빈칸을 붙였습니다.

필자는 가능하면 한 줄에 문장 하나를 위치시켰습니다. 한 줄에 여러 문장이 있는 경우 잘못 읽히는 경우가 있습니다. 만약 문장이 한 줄에 다 들어가지 않으면, 쉼표나 이항 연산자 다음에서 다음 줄로 나눴습니다. 이렇게 하는 것이 세미콜론 자동 삽입에 의해서 숨겨지는 오류를 줄여줍니다(세미콜론 자동 삽입에 대한 비극은 부록 A에서 살펴봅니다). 나뉘어진 나머지 문장은 4칸을 더 들여 쓰거나 모호한 경우 8칸을 들여 썼습니다(예를 들어 줄을 나눈 부분이 if 문의 조건 부분인 경우).

필자는 if나 while 같은 구조적 문장을 사용할 때 블록을 사용합니다. 왜냐하면 다음의 예처럼 블록을 사용하지 않으면 오류가 발생하기 쉽습니다.

```
if (a)
    b();
```

위의 코드가 아래와 같이 됐을 때,

```
if (a)
    b();
    c();
```

이 코드는 다음과 같이 인식되어 오류를 찾기 힘들 수 있습니다.

```
if (a) {  
    b();  
    c();  
}
```

하지만 실제 위의 코드가 의미하는 바는 다음과 같습니다.

```
if (a) {  
    b();  
}  
c();
```

이런 뜻인 것처럼 보이는데 실제로는 다른 뜻을 의미하는 코드는 버그의 원인이 되기 쉽습니다. 중괄호 한 쌍은 찾기 어려운 버그를 막는 저렴한 예방책 중에 하나입니다.

필자는 항상 K&R 스타일을 사용합니다. 즉 {를 줄의 처음에 놓는 것이 아니라 마지막에 놓습니다. 이렇게 함으로써 자바스크립트의 return 문과 관련된 치명적인 설계 실수로 인한 문제를 피할 수 있습니다.

필자는 예제에 얼마의 주석을 포함시켰습니다. 후에 프로그램을 읽게 되는 사람(또는 자신)에게 어떤 생각으로 이렇게 작성했는지 알게 해주는 주석을 남기는 것을 좋아합니다. 때때로 주석을 미래의 자신에게 중요한 메시지를 보내는 타임머신이라고 생각합니다.

또한 주석을 항상 최신 내용에 맞게 업데이트 하려고 노력합니다. 잘못된 주석은 오히려 프로그램을 더 읽기 어렵고 이해하기 힘들게 만듭니다. 이런 상황은 정말 안 좋습니다.

필자는 다음의 예처럼 불필요한 주석으로 여러분의 시간을 빼앗지 않으려고 노력했습니다.

```
i = 0; // i를 0으로 설정
```

자바스크립트에서 필자는 한 줄 주석을 선호합니다. 블록 주석은 공식적인 문서화나 일시적으로 코드를 주석화시킬 때(comment out)만 사용합니다.

필자가 선호하는 또 하나는 주석 없이도 프로그램 자체가 모든 것을 말하는 구조를 갖는 것입니다. 항상 성공적인 것은 아니지만 그렇게 하려고 노력하고 있으며 그렇지 못 한 경우에는 주석을 사용합니다.

자바스크립트는 C 구문을 가지고 있지만, 블록은 유효범위를 갖지 않습니다. 그래서 변수를 처음 사용하기 바로 전에 선언하라는 권고는 자바스크립트에서는 나쁜 충고입니다. 자바스크립트는 블록 유효범위가 아니라 함수 유효범위가 있습니다. 그래서 필자는 모든 변수를 함수의 첫 부분에서 선언합니다. 자바스크립트는 변수가 사용된 후에 선언되는 것을 허용하기는 하지만, 필자는 이렇게 실수처럼 보이는 프로그래밍을 원하지 않습니다. 실제 실수라면 눈에 바로 뛸 수 있어야 합니다. 이와 유사하게 필자는 할당 표현식을 if 문의 조건 부분에 결코 사용하지 않습니다.

```
if (a == b) { ... }
```

왜냐하면 위의 예는 마치 아래와 같은 코드가 원래 맞는 데 실수를 한 것처럼 보이기 때문입니다.

```
if (a === b) { ... }
```

필자는 이렇게 실제 실수가 아닌데 실수처럼 보이는 코드를 가능한 피하려고 합니다.

필자는 switch 문의 case에서 하나의 case 절을 실행한 후 벗어나지(break 문) 않고 다음 case 절로 내려가게(fall through) 하지 않습니다. 전에 다음 case 절까지 실행시키는 것이 때때로 좋은 이유에 관해 강력한 연설을 한 후에, 필자가 작성한 코드에서 이로 인해 발생한 의도하지 않은 버그를 찾게 됐습니다. 이러한 경험을 통해 다음 case 절까지 실행하는 것이 나쁘다는 것을 배우게 된 것은 어쩌면 행운이었습니다. 왜냐하면 이후로 언어의 기능들을 살펴볼 때, 때때로 유용하지만 어떤 경우에는 위험한 기능들에 특별한 주의를 기울이기 때문입니다. 이러한 것들은 제대로 사용됐는지 알기가 어렵기 때문에 가장 안 좋은 부분이라고 할 수 있습니다. 이런 부분들이 버그가 숨어있는 부분입니다.

품질이라는 문제는 자바스크립트의 설계나 구현, 표준화 단계에서 중요하게 고려된 내용이 아니었습니다. 이런 점이 언어의 약점들을 피해서 사용하려는 사람들에게 커다란 짐을 안겨줬습니다.

자바스크립트는 규모가 큰 프로그램에 필요한 것들을 제공하지만 반면에 대규모 프로그램에 반하는 것들도 제공합니다. 예를 들어, 전역변수를 쉽게 사용할 수 있는 편리함을 제공하기는 하지만 프로그램의 복잡함이 커지면 커질수록 전역변수는 문제 발생 확률을 높입니다.

필자는 애플리케이션이나 라이브러리를 포함하는 하나의 전역변수를 사용합니다. 모든 객체는 자신의 이름 공간(namespace)을 갖기 때문에 이렇게 하는 것이 객체를 사용하는 코드를 구성하는 쉬운 방법입니다. 클로저의 사용은 더 나은 정보 은닉과 모듈화를 강력하게 합니다.



## 아름다운 속성에 대한 단상

나는 대답을 기다리며 내 손을 그대의 발에다, 내 눈을 그대의 상(像)에다, 내 심장을 그대의 전신 각부에다 감히 더럽히기를 꺼리지 않겠소. 그대를 위하여 근면 극기하는 ...

—윌리엄 셰익스피어, 사랑의 헛수고



필자는 작년에 컴퓨터 프로그램에서 말 그대로 아름다움을 주제로 한 모음집인 『Beautiful Code』(한빛미디어, 2007)의 한 장(chapter)을 위한 기고가로 초대받았는데, 그 장을 자바스크립트로 작성하고 싶었습니다. 자바스크립트라는 언어가 기반하고 있는 추상적이고 강력하면서도 유용한 것들을 보여주길 원했습니다. 그리고 자바스크립트가 판에 박힌 역할을 하는 곳을 연상시키는 브라우저나 그 외 다른 환경들을 피하고 싶었습니다. 자바스크립트에 무게를 실어주는 뭔가 홀륭한 것을 보여주고 싶었습니다.

필자는 즉시 JSLint(부록 C 참조)에서 사용한 Vaughn Pratt의 하향식(top down) 연산자 우선 순위 파서를 염두에 두었습니다. 파싱은 컴퓨터 작업 중에서 중요한 주제 중 하나입니다. 언어 자체를 위한 컴파일러를 작성할 수 있는 능력의 여부가 여전히 한 언어의 완전성을 테스트하는 잣대입니다.

필자는 자바스크립트로 작성한 자바스크립트 파서의 모든 코드를 책에 실고 싶었습니다. 하지만 필자의 장(chapter)은 30, 40개의 장 중에서 단지 하나일 뿐이었습니다. 그래서 사용해야 할 페이지 수에 제한을 생각하지 않을 수 없었으며, 설상

가상으로 그 책을 읽게 되는 대부분의 독자는 자바스크립트에 대한 경험이 없어서 자바스크립트 언어 자체와 특징들도 설명을 해야만 하는 상황이었습니다.

그래서 언어의 부분집합을 만들기로 결정했습니다. 이렇게 함으로써, 언어 전체를 파싱할 필요도 없어졌고, 언어 전체에 대해 기술할 필요도 없어졌습니다. 필자는 자바스크립트의 부분집합을 단순화된(Simplified) 자바스크립트로 불렀습니다. 파일을 작성하는데 필요한 기능들만 부분집합에 포함시켰기 때문에 부분집합을 구성하는 것은 쉬웠습니다. 이렇게 해서 『Beautiful Code』의 한 장을 집필할 수 있었습니다.

단순화된 자바스크립트에는 다음과 같은 좋은 것들만이 포함돼 있습니다.

#### 일급 객체(first-class object)인 함수<sup>1</sup>

단순화된 자바스크립트에서 함수는 어휘적 유효범위를 갖는 람다(lambda)입니다.

#### 프로토타입으로 상속을 하는 동적 객체

객체는 클래스가 필요 없습니다. 어떤 객체든지 단순히 할당하는 식으로 새로운 구성요소를 추가할 수 있습니다. 객체는 다른 객체로부터 상속 받을 수 있습니다.

#### 객체 리터럴과 배열 리터럴

새로운 객체와 배열을 생성하는 매우 편리한 표기법입니다. 자바스크립트의 리터럴은 JSON이라는 데이터 교환 형식에 영감을 주었습니다.

이 부분집합은 좋은 점(Good Parts) 중에서도 가장 좋은 부분들만을 포함하고 있습니다. 매우 작은 언어이기는 하지만 매우 표현적이며 강력합니다. 자바스크립트에는 실제로 별로 드릴게 없는 많은 수의 부가 기능이 있습니다. 그리고 이어지는 부록들에서 살펴보겠지만 오히려 손해가 되는 기능도 많이 있습니다. 하지만 이 부분집합에는 그러한 나쁜 것들이 모두 제외되어 하나도 포함돼 있지 않습니다.

---

1. 역자 주/ 일급 객체에 대한 설명은 1장의 역자 주를 참고하세요.

단순화된 자바스크립트는 아주 엄격한 의미에서 자바스크립트의 부분집합은 아닙니다. 왜냐하면 필자가 몇몇 속성을 추가했기 때문입니다. 여기에는 pi가 간단한 상수로 추가돼 있는데, 이렇게 한 것은 파서의 기능을 보여주기 위해서였습니다. 또한 단순화된 자바스크립트에는 더 나은 예약어 정책을 보이려고 했으며 심지어 예약어가 필요 없다는 점도 나타냈습니다. 그래서 함수에서는 어떤 단어든지 변수(또는 매개변수)나 속성(이를테면 if 등)의 이름으로 쓰일 수 있지만 동시에 둘 다에 쓰일 수는 없습니다. 이렇게 정의하면 예약어라는 개념이 없기 때문에 사용해서는 안 되는 속성들을 모두 알고 있을 필요가 없어져서 언어를 배우기가 훨씬 쉽습니다. 또한 새로운 기능을 추가하기 위해 더 많은 단어를 예약할 필요가 없어지므로 언어를 확장하는 것도 보다 쉬워집니다.

필자는 또한 블록 유효범위를 추가했습니다. 블록 유효범위가 꼭 필요한 기능은 아니지만 없을 경우에는 이 개념에 익숙한 프로그래머들에게 혼란을 줍니다. 필자가 블록 유효범위 개념을 추가한 주된 이유는 이 파서가 자바스크립트 외에 블록 유효범위 개념을 잘 갖춘 다른 언어를 파싱하는데에도 사용할 수 있겠다는 생각 때문이었습니다. 필자가 작성한 파서의 코드는 블록 유효범위를 사용할 수 있든지 없든지 간에 그것에 영향을 받지 않는 스타일로 작성되었습니다. 필자는 여러분이 이런 식으로 프로그래밍하기를 권합니다.

필자가 이 책을 구상하기 시작했을 때 원했던 것은 부분집합 개념을 더 발전시키고, 기존의 언어에 변화를 주지 않으면서 중요성이 떨어지는 기능들을 배제시키고 사용하는 방법을 보이는 것이었습니다.

각 기능의 비용이 제대로 계산되지 않은 체 기능 중심으로만 설계된 제품을 많이 볼 수 있습니다. 제품을 이해하고 사용하기 어렵게 만드는 기능은 소비자에게 오히려 부정적인 요소입니다. 사람들이 좋아하는 제품은 딱 필요한 기능을 하는 제품입니다. 제대로 된 기능을 하는 제품을 설계하는 것이 기능만 많은 제품을 설계하는 것보다 훨씬 더 어렵습니다.

기능을 추가하기 위해서는 명세를 정하고, 설계하고 개발하는 비용이 필요합니다. 거기에다가 테스트 비용까지도 포함됩니다. 기능이 많으면 많을수록 문제를 일으키거나 또는 다른 부분과 이상하게 상호작용할 기능이 많아질 가능성이 높아집니다. 소프트웨어 시스템에서 저장 비용은 이제 거의 무시해도 될 문제가 됐지만 모바일 애플리케이션에서는 여전히 중요한 문제입니다. 또한 무어의 법칙이 배터리에는 적용되지 않기 때문에 실행 비용의 상승도 무시 못할 문제입니다.

기능의 추가는 문서화 비용도 관련돼 있습니다. 모든 기능은 매뉴얼에 페이지를 가중시키고 이를 익히기 위한 비용도 증가시킵니다. 소수의 사용자에게만 가치를 제공하는 기능들은 그 외 대다수의 사용자에게 불필요한 비용 추가라는 짐을 지우게 됩니다. 그러므로 제품이나 프로그래밍 언어를 설계할 때는 제대로 된 핵심 기능들, 즉 좋은 점만을 포함해야 합니다. 이런 기능들만으로도 필요한 것을 대부분 할 수 있습니다.

우리는 사용하는 제품들에서 좋은 점(good parts)만을 찾아낼 수 있습니다. 단순 함이야말로 가치가 있으며 만약 단순하게 제공되지 않는다면 우리가 직접 단순화 할 수 있습니다. 필자의 전자레인지는 정말 많은 기능들이 있습니다. 하지만 필자가 사용하는 기능은 단순히 시계와 조리 기능뿐입니다. 그런데 시계의 경우는 시간을 맞추는 일조차도 도전적입니다. 우리는 좋은 점만을 찾고 이것들만 사용함으로써 기능 위주 설계로 인한 복잡함에 대처할 수 있습니다.

제품이든지 프로그래밍 언어이든지 간에 좋은 점(good parts)으로만 설계된다면 좋을 것입니다.



A.P.P.E.N.D.I.X.

A

## 나쁘지만 사용해야 하는 부분들(Awful parts)

언행이 모두 귀감(awful)이 되는 ...

– 윌리엄 셰익스피어, 페리클리스



이번 장에서는 자바스크립트에서 쉽게 피하기 힘든 문제가 있는 부분들을 살펴볼 것입니다. 이러한 부분들을 잘 인지하고 적절히 다룰 준비를 해야 합니다.

### 01 | 전역변수

자바스크립트의 나쁜 점들 중에서 가장 나쁜 점은 언어가 전역변수에 기반하고 있다는 것입니다. 전역변수는 모든 유효범위(scope)에서 보이는 변수입니다. 매우 작은 프로그램에서는 전역변수가 편리할 수 있지만, 프로그램이 점점 커짐에 따라 전역변수는 다루기가 까다로워집니다. 전역변수는 언제든지 프로그램의 모든 부분에서 변경될 수 있기 때문에 프로그램의 동작을 심각할 정도로 뒤틀릴 수 있습니다. 그러므로 전역변수를 사용하면 프로그램의 신뢰성을 저하시킵니다.

전역변수는 같은 프로그램 내에 있는 하위 프로그램이 독립적으로 실행되는 것을 더 어렵게 만듭니다. 하위 프로그램들이 같은 이름의 전역변수들을 공유하면, 각각

서로를 방해하고 오류가 발생하기 쉽습니다. 게다가 오류가 발생했을 경우 어디가 잘못됐는지 찾기가 어려워집니다.

많은 언어에 전역변수가 있습니다. 예를 들어, 자바의 public static 속성은 전역변수입니다. 전역변수가 자바스크립트에서 유독 문제가 되는 것은 전역변수가 필요한 경우에만 사용할 수 없기 때문입니다. 자바스크립트는 링커가 없습니다. 그래서 모든 컴파일 단위는 하나의 공용 전역객체에 로딩됩니다.

전역변수를 정의하는 방법은 세가지가 있습니다. 첫 번째는 어떠한 함수에도 속하지 않는 위치에 다음과 같이 var 문을 사용하는 것입니다.

```
var foo = value;
```

두 번째 방법은 전역객체에 직접적으로 속성을 추가하는 것입니다. 전역객체는 모든 전역변수의 컨테이너입니다. 웹 브라우저에서 전역객체 이름은 window입니다.

```
window.foo = value;
```

세 번째 방법은 변수를 선언 없이 사용하는 것입니다. 다음과 같은 방법이 뮤시적인 전역변수 선언 방법입니다.<sup>1</sup>

```
foo = value;
```

이런 방법은 초보자들이 변수를 선언하지 않아도 사용할 수 있게 하기 위해서 고안된 것입니다. 하지만 불행히도 이렇게 선언 없이 변수를 사용하는 것은 많은 실수를 유발합니다. 자바스크립트의 이러한 정책은 때때로 매우 찾기 힘든 버그를 만들기도 합니다.

---

1. 역사 주/ 즉 var 없이 변수를 바로 사용하는 것입니다. 부언하면 함수 내에서건 블록 내에서건 var로 선언한 적이 없는 변수를 사용하면 그 변수는 전역변수가 됩니다.

## 02 | 유효범위(Scope)

자바스크립트의 구문은 C에서 가져온 것입니다. C 같은 구문을 가진 다른 언어들은 블록(중괄호로 묶인 문장들의 집합)이 유효범위를 가집니다. 블록 내에서 선언된 변수들은 블록 바깥에서 보이지 않습니다. 자바스크립트는 블록 구문을 사용하기는 하지만, 블록 유효범위를 제공하지 않습니다. 즉 블록 내에서 선언한 변수는 블록을 포함하는 함수 내부 모든 곳에서 볼 수 있습니다. 이러한 속성은 다른 언어에 익숙한 프로그래머들에게는 놀랄만한 일입니다.

대부분의 언어에서는 일반적으로 변수가 처음 사용되는 지점에서 선언하는 것이 가장 좋습니다. 하지만 자바스크립트에서는 블록 유효범위를 갖지 않기 때문에 이렇게 하는 것이 좋지 않습니다. 대신에 함수에서 사용하는 모든 변수를 함수의 첫 부분에서 선언하는 것이 좋습니다.

## 03 | 세미콜론 삽입

자바스크립트에는 자동으로 세미콜론을 삽입하여 잘못된 프로그램을 교정하려는 메커니즘이 있습니다. 하지만 이러한 메커니즘에 의존해서는 안 됩니다. 그렇게 하면 더 중대한 오류가 발생할 수 있습니다.

때때로 세미콜론이 원하지 않는 곳에 자동 삽입됩니다. 다음의 예를 통해 세미콜론이 return 문장에 자동 삽입됐을 때의 결과를 살펴보겠습니다. return 문으로 값을 반환하려면, 값을 나타내는 표현식의 시작이 반드시 return과 같은 줄에 있어야 합니다.

```
return
{
    status: true
};
```

이 예는 status라는 속성을 가진 객체를 반환하려는 것 같아 보입니다. 하지만 안 타깝게도 세미콜론의 자동 삽입으로 인해 undefined를 반환합니다. 프로그램을 잘못 해석하게 하는 세미콜론 자동 삽입에 대한 아무런 경고도 없습니다. 이 문제를 피하기 위해서는 다음과 같이 {를 다음 줄의 시작이 아니라 return이 있는 줄의 마지막에 두어야 합니다.

```
return {  
    status: true  
};
```

## 04 | 예약어

다음에 나오는 단어들은 모두 예약어입니다.

```
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 단어들의 대부분은 실제 자바스크립트에서 사용하지 않습니다.

이 예약어들은 변수나 매개변수의 이름으로 사용할 수 없습니다. 예약어가 객체 리터럴에서 속성 이름으로 사용될 때는 따옴표로 묶어줘야 합니다. 이렇게 사용된 이름은 마침표(.) 표기법으로는 사용할 수 없고, 대신에 대괄호 표기법으로 사용해야 합니다.

```
var method;           // ok
var class;           // 잘못된 사용
object = {box: value}; // ok
object = {case: value}; // 잘못된 사용
object = {'case': value}; // ok
object.box = value; // ok
object.case = value; // 잘못된 사용
object['case'] = value; // ok
```

## 05 | 유니코드

자바스크립트는 유니코드가 최대 65,536자를 갖는 시절에 설계됐습니다. 하지만 유니코드는 현재 백만 자 이상을 수용할 수 있습니다.

자바스크립트에서 문자는 16비트입니다. 이것은 원래 유니코드 65,536자(현재 BMP(Basic Multilingual Plane)로 불려짐)를 다루는데는 충분합니다. 여기에 포함되지 않는 유니코드 문자들은 문자의 쌍으로 표현할 수 있습니다. 즉 \u 표기 두 개를 연속으로 붙여서 사용합니다.<sup>2</sup> 유니코드 상으로는 이렇게 16비트 글자 한 쌍(pair)이 한 글자입니다. 하지만 자바스크립트는 이러한 한 쌍을 두 개의 글자로 생각합니다.

## 06 | typeof

typeof 연산자는 피연산자의 타입을 알 수 있게 해주는 문자열을 반환합니다. 그래서 다음의 예는 'number'를 반환합니다.

```
typeof 98.6
```

---

2. 역사 주/ 예를 들어 높은음자리표를 나타내는 U+1D11E는 \uD834\uDD1E과 같이 나타냅니다. 이러한 표기를 surrogate pair라고 하며 자바에서도 사용하는 방법입니다. 더 자세한 내용은 [http://en.wikipedia.org/wiki/Surrogate\\_pair](http://en.wikipedia.org/wiki/Surrogate_pair)를 참조하세요.

그러나 안타깝게도 다음의 예는 null이 아니라 object를 반환합니다.

```
typeof null
```

그러므로 null을 확인하는 더 좋은 방법은 다음과 같이 하는 것입니다.

```
my_value === null
```

여기에서 더 큰 문제는 객체인지 아닌지를 판별할 때입니다. typeof 연산자는 null과 객체를 구분하지 못합니다. 하지만 null은 거짓이고 모든 객체는 참이라는 속성을 이용하면 다음과 같이 구분할 수 있습니다.

```
if (my_value && typeof my_value === 'object') {  
    // my_value는 객체나 배열  
}
```

뒤에 나오는 10. NaN 절과 11. 가짜 배열(Phony Arrays) 절도 참조하기 바랍니다.

정규 표현식 객체에 대한 타입 값은 자바스크립트 실행환경에 따라 다르게 나타납니다. 다음의 예를 어떤 실행환경에서는 'object'라고 반환하는데 반해 어떤 환경에서는 'function'이라고 반환합니다. 'regexp'를 반환하는 것이 더 유용해 보이기는 하지만 표준에서는 이를 허용하고 있지 않습니다.

```
typeof /a/
```

## 07 | parseInt

parseInt는 문자열을 정수로 바꿔주는 함수입니다. 이 함수는 문자열의 첫 문자부터 변환을 하다가 숫자가 아닌 문자를 만나게 되면 변환을 멈춥니다. 그래서 parseInt("16")과 parseInt("16 tons")은 결과가 같습니다. 변환되지 않은 여분의 텍스트가 무엇인지 알려주면 좋겠지만 이 함수는 그렇게 하지 않습니다.

문자열의 첫 번째 문자가 0이면 문자열은 10진수 대신에 8진수로 간주됩니다. 8진수에서 8과 9는 숫자가 아니기 때문에 parseInt("08")과 parseInt("09")의 결과값은 0입니다. 이러한 오류는 날짜나 시간을 파싱할 때 문제를 일으킬 수 있습니다. 그나마 다행인 것은 parseInt에 기수(진법)를 지정할 수 있다는 것입니다. 그래서 parseInt("08", 10)의 값은 8입니다. parseInt를 사용할 때는 항상 기수 매개변수를 지정할 것을 권합니다.

## 08 | +

+ 연산자는 덧셈을 하거나 문자열을 연결합니다. 어떤 작업을 할지는 피연산자의 데이터 타입에 따라 결정됩니다. 둘 중에 하나의 피연산자가 빈 문자열이면, 다른 피연산자를 문자열로 변환하여 결과값을 돌려줍니다. 두 피연산자가 모두 숫자이면 덧셈을 실행합니다. 이 외의 경우에는 두 피연산자를 모두 문자열로 변환하여 두 문자열을 연결합니다. 이러한 복잡한 행동이 버그의 일반적인 원인이 됩니다. + 연산자를 더하기로 사용한다면 두 피연산자가 모두 숫자인지 확인하는 것이 좋습니다.

## 09 | 부동 소수점

이진 부동 소수점(binary floating-point) 숫자는 소수 부분을 제대로 처리하지 못합니다. 그래서  $0.1 + 0.2$ 는  $0.3$ 과 같아야 하지만 실제로 그렇지 않습니다. 이러한 문제는 자바스크립트에서 가장 빈번하게 보고되는 버그이며 이진 부동 소수점 연산에 관한 IEEE 표준(IEEE 754)을 채택한 고의적인 결과라고 볼 수 있습니다. 이 표준은 많은 애플리케이션에 자주 채택되지만, 중학교만 가도 배울 수 있는 당연한 규칙을 위반하고 있습니다. 다행히도 부동 소수점에서 정수 부분에 대한 연산은 정확합니다. 그래서 소수점 부분의 연산 오류는 값을 조정함으로써 피할 수 있습니다.

예를 들어 센트까지 포함하는 달러 값은 여기에 100을 곱한 후에 덧셈을 합니다. 그리고 나서 이 결과를 다시 100으로 나누어서 원하는 결과를 정확히 얻을 수 있습니다. 사람들은 보통 돈을 계산할 때 그 결과가 정확할 것이라고 당연하게 생각합니다. 그러므로 돈과 관련된 계산이라면 이 문제와 관련해서 더욱 주의를 기울여야 합니다.

## 10 | NaN

NaN은 IEEE 754에 정의돼 있는 특별한 값입니다. NaN은 숫자가 아님(not a number)을 의미함에도 다음의 예는 참입니다.

```
typeof NaN === 'number'      // 참
```

이 값은 문자열이 숫자 형태가 아닌 데, 이 문자열을 숫자로 변경하려 하는 경우에 발생합니다. 예를 들면 다음과 같습니다.

```
+ '0'          // 0
+ 'oops'       // NaN
```

NaN이 산술 연산의 피연산자이면 결과는 NaN이 됩니다. 그러므로 일련의 산술 공식을 실행한 결과가 NaN이라면 적어도 피연산자 하나가 NaN이거나 어느 부분에 선가 NaN이 발생했다는 뜻입니다.

앞서 살펴본 것처럼 typeof 연산자는 숫자와 NaN을 구분하지 못합니다. 그리고 놀랍게도 다음의 예가 보여주는 것처럼 NaN은 자신과 같지 않습니다.

```
NaN === NaN      // 거짓  
NaN !== NaN     // 참
```

자바스크립트는 숫자와 NaN을 구분하도록 isNaN이라는 함수를 제공합니다.

```
isNaN(NaN)      // 참  
isNaN(0)        // 거짓  
isNaN('oops')   // 참  
isNaN('0')      // 거짓
```

isFinite 함수는 NaN과 Infinity를 거부하기 때문에 어떤 값이 사용할 수 있는 숫자인지를 확인하는데는 isFinite 함수가 최상의 방법입니다. 한가지 아쉬운 점은 isFinite는 인수로 넘어온 값을 숫자로 변환하려고 합니다. 그래서 값이 실제 숫자가 아닌 경우에는 적절하지 않습니다. 그러므로 다음과 같이 isNumber 함수를 정의해서 사용하는 것도 좋은 방법입니다.

```
var isNumber = function isNumber(value) {  
    return typeof value === 'number' && isFinite(value);  
}
```

## 11 | 가짜 배열(Phony Arrays)

자바스크립트는 진정한 배열이 없습니다. 하지만 그렇다고 해도 전부 나쁘지는 않습니다. 자바스크립트의 배열은 실제 사용하기 쉽습니다. 굳이 크기를 지정하지 않아도 되고, 결코 침자 경계를 초과했다는 오류를 발생하지 않습니다. 하지만 성능에 있어서는 진짜 배열보다 꽤 나쁩니다.

`typeof` 연산자는 배열과 객체를 구분하지 않습니다. 그러므로 값이 배열인지 확인하기 위해서는 `constructor` 속성을 확인해봐야 합니다.

```
if (my_value && typeof my_value === 'object' &&
    my_value.constructor === Array) {
    // my_value는 배열
}
```

이 확인 방법은 배열이 다른 프레임이나 창(window)에서 만들어진 경우 제대로 동작하지 않습니다. 그래서 다음과 같이 수정해야 그러한 경우에도 잘 확인할 수 있습니다.

```
if (my_value && typeof my_value === 'object' &&
    typeof my_value.length === 'number' &&
    !(my_value.propertyIsEnumerable('length'))) {
    // my_value는 배열
}
```

`arguments` 배열은 배열이 아닙니다. `arguments`는 `length` 속성을 가진 객체입니다. 위의 확인 방법을 사용하여 `arguments` 배열을 테스트하면 배열로 구분하는데 이는 `arguments`가 배열 메소드들을 가지고 있지 않더라도 때때로 적절할 수 있습니다. 이 확인 방법의 한 가지 문제점은 `propertyIsEnumerable` 메소드가 재정의되는 경우 제대로 동작하지 않는다는 것입니다.

## 12 | 거짓인 값들(Falsy Values)

자바스크립트는 [표 A-1]에 나오는 것처럼 놀라울 정도로 많은 거짓 값이 있습니다.

[표 A-1] 자바스크립트의 많은 거짓 값

값	데이터 타입
0	숫자
NaN(숫자가 아님)	숫자
'' (빈 문자열)	문자열
false	불리언
null	객체
undefined	undefined

이 값들은 모두 거짓이지만, 서로 맞바꿀 수 없습니다. 예를 들어 어떤 객체가 특정 속성이 있는지 확인하는 다음과 같은 코드는 잘못된 방법입니다.

```
value = myObject[name];
if (value == null) {
    alert(name + ' not found.');
}
```

객체에 없는 속성을 참조하면 값은 undefined입니다. 그런데 코드에서는 이를 null과 비교하고 있습니다. 비교를 위해 보다 신뢰 있는 === 연산자 대신에 == 연산자(부록 B 참조)를 사용하고 있는데 이 연산자는 필요한 경우 타입을 변환하여 비교하기도 합니다. 이러한 두 가지 문제로 이 코드는 제대로 동작할 수도 있고 안 할 수도 있습니다. 즉 myObject[name]이 실제 존재하지 않는 경우에는 문제 없이 동작하지만, myObject[name]이 존재하는데 그 값이 null인 경우에는 제대로 동작하지 않습니다. 그러므로 위의 값들이 모두 거짓이라 해도 서로 바꿔가며 써서는 안 되고 경우에 맞는 값을 사용해야 합니다.

`undefined`와 `NaN`은 상수가 아닙니다. 이 둘은 전역변수이며 원하는 경우 값을 변경할 수도 있습니다. 실제로 이것은 불가능해야 맞는데 가능한 게 현실입니다. 그렇다 하더라도 절대 `undefined`와 `NaN`의 값을 변경해서는 안 됩니다.

## 13 | `hasOwnProperty`

3장에서 `hasOwnProperty` 메소드는 `for in` 문의 문제를 해결하기 위한 대안으로 필터 역할을 했습니다. 불행히도 `hasOwnProperty`는 연산자가 아니라 메소드여서, 이 메소드를 다른 함수나 심지어는 함수가 아닌 값으로 대체할 수 있습니다.

```
var name;
another_stooge.hasOwnProperty = null;           // 문제가 되는 지점
for (name in another_stooge) {
    if (another_stooge.hasOwnProperty(name)) { // 문제 발생
        document.writeln(name + ': ' + another_stooge[name]);
    }
}
```

## 14 | 객체

자바스크립트의 객체는 프로토타입 체인 상의 구성요소들을 불러올 수 있기 때문에 결코 온전히 빈 상태를 가질 수 없습니다. 그런데 때때로 이 점이 문제가 될 수 있습니다. 예를 들어 텍스트에서 각각의 단어가 몇 번 나오는지 숫자를 세는 프로그램을 작성한다고 가정해 보겠습니다. 먼저 `toLowerCase` 메소드를 사용하여 텍스트를 소문자로 통일시키고 난 후, `split` 함수를 정규 표현식과 함께 사용하여 각각의 단어를 배열에 담을 수 있습니다. 그리고 나서, 배열을 돌면서 각각이 몇 번 나오는지 숫자를 셀 수 있습니다.

```

var i;
var word;
var text =
    "This oracle of comfort has so pleased me, " +
    "That when I am in heaven I shall desire " +
    "To see what this child does, " +
    "and praise my Constructor.";

var words = text.toLowerCase().split(/\s+/);
var count = {};
for (i = 0; i < words.length; i += 1) {
    word = words[i];
    if (count[word]) {
        count[word] += 1;
    } else {
        count[word] = 1;
    }
}

```

예제의 결과를 살펴보면 count['this']는 2고 count.heaven은 1입니다. 하지만 count,constructor는 이상하게 보이는 문자열을 포함하고 있습니다. 이런 이유는 count 객체가 constructor라는 객체를 속성으로 갖는 Object.prototype을 상속 받았기 때문입니다. + 연산자처럼 += 연산자는 피연산자가 숫자가 아닌 경우 더하지 않고 연결을 합니다. 그러므로 count['constructor'] += 1 차례가 왔을 때 +=는 객체인 count['constructor']를 문자열로 변환한 후 1과 연결한 것입니다.

이러한 문제를 피하기 위해서는 for in 문에서 이런 유형의 문제를 피하기 위해서 사용했던 hasOwnProperty를 사용하거나 특정한 타입만을 골라서 작업할 수 있습니다. 이번 예제의 경우 아주 특별한 경우가 아니므로 다음과 같이 수정함으로써 문제를 간단히 해결할 수 있습니다.

```
if (typeof count[word] === 'number') {
```



## 나쁜 점들

헌데 당신께 꼭 좀 물어볼 말이 있는데, 애당초 대체 나의 어떤 나쁜 점(bad parts) 때문에 나를 사랑하게 된 것입니까?

– 윌리엄 셰익스피어, 헛소동



이번 부록에서는 자바스크립트에서 문제가 되긴 하지만 쉽게 피 할 수 있는 속성들을 설명할 것입니다. 여기서 얘기하는 부분들의 사용을 피함으로써 자바스크립트를 보다 좋은 언어로 만들 수 있고 더 나은 프로그래머가 될 수 있습니다.

### 01 | ==

자바스크립트는 동등 비교 연산자로 `==`/`!=`과 이것들의 나쁜 쌍등이인 `==!`/`=!` 있습니다. 이것들 중에 앞선 한 쌍 만이 원하는 대로 제대로 동작합니다. 피연산자 두 개가 같은 데이터 타입이면서 같은 값일 때 `==`은 참이 되고 `!=`은 거짓이 됩니다. 나쁜 한 쌍도 두 피연산자의 타입이 같은 경우에는 바르게 동작합니다. 하지만 두 피연산자의 타입이 다른 경우 값을 강제로 변환하여 비교합니다. 값을 변환하여 비교하는 규칙은 복잡하고 외우기도 쉽지 않습니다. 다음은 그 중에서 흥미로운 몇 가지 예입니다.

```
'' == '0'          // 거짓
0 == ''            // 참
0 == '0'          // 참

false == 'false'   // 거짓
false == '0'        // 참

false == undefined // 거짓
false == null      // 거짓
null == undefined // 참

'\t\r\n' == 0      // 참
```

필자가 권하고 싶은 것은 결코 `==/!=`를 사용하지 말고 대신에 항상 `===/!==`를 사용하라는 것입니다. 위에 있는 예들은 `==` 연산자를 사용할 경우 모두 거짓입니다.

## 02 | with 문

자바스크립트는 객체 하나에 속한 속성들을 접근할 때 간편함을 제공할 목적으로 `with` 문이 있습니다. 하지만 이런 의도와는 달리 `with` 문은 때때로 예측 못한 결과를 낳을 수 있기 때문에 사용하지 않는 것이 좋습니다.

다음과 같은 `with` 문은,

```
with (obj) {
  a = b;
}
```

다음과 같은 의미입니다.

```
if (obj.a === undefined) {  
    a = obj.b === undefined ? b : obj.b;  
} else {  
    obj.a = obj.b === undefined ? b : obj.b;  
}
```

그래서 결국 이 with 문은 다음의 문장들 중에 하나와 같게 됩니다.

```
a = b;  
a = obj.b;  
obj.a = b;  
obj.a = obj.b;
```

이 문장들 중에 어떤 문장으로 실행될지는 프로그램을 봐서는 알 수 없습니다. 또한 프로그램을 실행할 때마다 다르게 실행할 수 있고, 심지어는 프로그램을 실행하는 동안에도 달라질 수 있습니다. 프로그램을 제대로 읽을 수 없고 어떻게 실행할지를 예측할 수 없다면, 프로그램이 원하는 대로 제대로 실행할 것이라고 확신할 수 없습니다.

간단하게 언어적인 측면에서 봐도 with 문은 자바스크립트 프로세서의 속도를 현저히 느리게 만듭니다. 왜냐하면 with 문은 변수 이름의 어휘적 바인딩을 어렵게 하기 때문입니다. with 문은 그 의도는 좋았지만, 오히려 없는 것이 더 나을 뻔 했습니다.

## 03 | eval

eval 함수는 문자열을 자바스크립트 컴파일러에 넘긴 후 그 결과를 실행시킵니다. 이 함수는 자바스크립트에서 가장 오용되고 있는 기능 중 하나입니다. eval은 대부분 자바스크립트에 대한 온전한 이해가 없는 사람들이 가장 많이 사용합니다. 예를 들어, 만약 .(마침표) 표기법은 아는데 첨자 표기법은 모른다고 할 경우, 다음과 같이 eval 함수를 사용하려고 할 것입니다.

```
eval("myValue = myObject." + myKey + ";" );
```

하지만 이것은 다음과 같이 하면 됩니다.

```
myvalue = myObject[myKey];
```

eval을 사용한 코드는 매우 읽기 어렵습니다. 또한 eval 함수를 사용하면 단순한 할당문 실행을 위하여 컴파일러를 기동해야 하므로 속도가 매우 느려질 수 있습니다. 또한 eval 함수는 JSLint(부록 C 참조)를 곤란하게 만들어서 문제점을 찾아내는 툴의 기능을 현저히 저하시킵니다.

또한 eval 함수는 인수로 넘어온 텍스트에 너무 많은 권한을 허용하기 때문에 애플리케이션의 보안을 위태롭게 할 수도 있습니다. 그리고 이 함수는 with 문이 하는 것처럼 언어의 성능을 저하시키는 원인이 될 수 있습니다.

함수 생성자도 eval의 또 다른 형태이며 같은 이유로 사용을 피해야만 합니다.

브라우저가 제공하는 setTimeout과 setInterval 함수는 문자열 인수나 함수 인수를 취할 수 있습니다. 이 함수들에 문자열 인수를 제공하면 eval처럼 동작하게 되므로, 문자열 인수를 넘기는 것은 피해야 합니다.

## 04 | continue 문

continue 문은 루프의 처음으로 제어를 이동합니다. 필자는 리팩토링을 통해 continue 문을 제거했을 때 성능이 향상되지 않는 경우를 본 적이 없습니다.

## 05 | 다음 case 절까지 실행하는 switch 문(switch Fall Through)

switch 문은 FORTRAN IV가 go to 문을 다룬 이후에 고안됐습니다. 각각의 case 절은 명시적으로 벗어나게 하지 않으면 다음 case 절까지 계속해서 실행됩니다.

어느 날 어떤 분이 필자에게 JSLint에서 다음 case 절까지 실행되는 case 절에 대해 경고를 해야 한다고 제안하는 글을 쓴 적이 있습니다. 그 분은 이것이 매우 일반적인 오류의 원인이고 코드상으로는 찾기 어려운 오류라고 지적해 주셨습니다. 필자는 모두 맞는 얘기라고 답하면서 하지만 다음 case 절까지 실행시킴으로써 얻는 이득이 그런 문제를 보상할 만큼 더 크다고 얘기했습니다.

다음날, 그 분은 JSLint에 오류가 있다고 알려왔습니다. 그것은 잘못 확인된 오류였지만 왜 그런가 자세히 살펴봤더니, 다음 case 절까지 실행하는 case 문을 사용했기 때문임이 밝혀졌습니다. 이때 큰 깨달음을 얻고 필자는 더 이상 이러한 유형의 case 문을 의도적으로 사용하지 않고 있습니다. 이러한 방침은 의도하지 않은 다음 case 절까지 실행하는 패턴을 훨씬 쉽게 찾을 수 있게 합니다.

언어에서 가장 나쁜 점은 명백하게 위험하거나 불필요한 속성들이 아닙니다. 왜냐하면 이러한 점들은 쉽게 피할 수 있기 때문입니다. 그보다는 매력적으로 보이는 폐단이 가장 나쁜 점입니다. 이러한 속성들은 유용하기도 하면서 위험하기도 하기 때문입니다.

## 06 | 블록이 없는 문장

if/while/do/for 문은 블록을 사용할 수도 있고 문장 하나만을 사용할 수도 있습니다. 문장 하나만을 사용하는 형식이 또 하나의 매력적인 폐단입니다. 이 형식은 문자 두 개를 아끼는 이점이 있지만, 좀 의심스러운 이점입니다. 이 형식은 프로그램의 구조를 애매하게 만들어서 후에 작업하는 사람이 쉽게 버그를 만들 수 있게 합니다.

예를 들어 다음과 같은 코드는,

```
if (ok)
    t = true;
```

다음과 같이 될 수 있습니다.

```
if (ok)
    t = true;
    advance();
```

이 코드는 마치 다음과 같이 보이지만,

```
if (ok) {
    t = true;
    advance();
}
```

실제로는 다음과 같은 의미입니다.

```
if (ok) {
    t = true;
    . . .
}
```

```
    }
    advance(  ) ;
```

이렇게 일을 처리하는 것처럼 보이는데 실제로 다르게 일을 처리하는 프로그램은 바르게 이해하기가 훨씬 어렵습니다. 그러므로 원칙적이고 일관된 블록의 사용은 프로그램의 이해를 훨씬 쉽게 합니다.

## 07 | ++ --

증감 연산자를 사용하면 코드를 매우 간결한 스타일로 작성할 수 있습니다. C 같은 언어에서는 다음과 같이 문자열 복사를 한 줄로 작성할 수 있게 만들어 놨으며, 또한 이런 스타일을 권장하기도 합니다.

```
for (p = src, q = dest; !*p; p++, q++) *q = *p;
```

하지만 밝혀진 바와 같이 이런 것은 부주의한 자세였습니다. 심각한 보안 취약점을 만드는 대부분의 베퍼 오버런 버그는 이와 같은 스타일의 코드 때문에 발생합니다.

필자는 실무에서 ++와 --를 사용하게 되면 코드가 더 빽빽해지고 더 까다로워지며, 보다 더 암호처럼 보이게 된다는 것을 경험하게 됐습니다. 그래서 필자는 규칙처럼 더 이상 증감 연산자를 사용하지 않습니다. 그 결과로 필자의 코딩 스타일은 좀더 깔끔해졌다고 생각합니다.

## 08 | 비트 연산자

자바스크립트는 자바와 같이 다음과 같은 비트 연산자가 있습니다.

```
&      and  
|      or  
^      xor  
~      not  
>>    부호 있는 오른쪽 시프트  
>>>   부호 없는 오른쪽 시프트  
<<    왼쪽 시프트
```

자바에서 비트 연산자는 정수에 대해서 동작합니다. 그런데 자바스크립트에는 정수형은 없고 단지 배 정도의 부동 소수점 숫자형만이 존재합니다. 그래서 비트 연산자는 대상이 되는 숫자를 일단 정수형으로 변환한 다음에 비트 연산을 수행하고 다시 원래 타입으로 되돌립니다. 대부분의 언어에서 비트 연산자는 하드웨어에 친근하고 속도도 매우 빠릅니다. 자바스크립트에서 비트 연산자는 하드웨어와 전혀 동떨어져 있고 속도도 매우 느립니다. 자바스크립트가 비트 연산을 위해 사용되는 경우는 매우 드뭅니다.

그래서 자바스크립트 프로그램에서는 원래 &를 사용하는 경우보다 &&를 사용하려다가 &로 잘못 사용하는 경우가 더 많습니다. 비트 연산자의 존재는 꼭 필요한 기능이라기 보다는 오히려 잠재된 버그를 만들어내기 쉽습니다.

## 09 | 함수 문장 vs 함수 표현식

자바스크립트는 함수 문장과 함수 표현식 모두가 있습니다. 이 둘은 정확히 일치하게 보이기 때문에 혼동이 됩니다. 함수 문장은 var 문장과 함수값 조합의 축약형입니다.

다음과 같은 함수 문장은,

```
function foo( ) { }
```

다음과 같은 의미라고 할 수 있습니다.

```
var foo = function foo( ) { };
```

이 책 전체에 걸쳐 필자는 두 번째 형식을 사용했습니다. 왜냐하면 두 번째 형식은 `foo`가 함수값(function value)을 가진 변수라는 것을 명확히 나타내기 때문입니다. 자바스크립트라는 언어를 잘 사용하기 위해서는 함수도 값(value)이라는 것을 이해하는 것이 중요합니다.

함수 문장은 위로 끌어올려지는 대상이 됩니다(hoisting). 이 말은 함수가 위치한 곳과 관계 없이 함수가 정의된 곳의 유효범위 가장 상위로 이동된다는 뜻입니다.<sup>1</sup> 이러한 특징은 함수를 사용하기 전에 반드시 선언해야 한다는 요구사항을 경감시키는데, 결국 필자 생각에는 구조를 엉성하게 만들 뿐입니다. 또한 이런 특징은 `if` 문에서 함수 문장 사용을 금하게 됩니다. 밝혀진 바에 따르면 대부분의 브라우저에서는 `if` 문 내에서 함수 문장 사용을 허용하고 있습니다. 하지만 이렇게 사용된 함수 문장이 어떻게 해석되는지는 브라우저마다 제각각입니다. 이런 점은 잠재적인 문제를 발생시킵니다.<sup>2</sup>

---

1. 역사 주/ 그렇기 때문에 자바스크립트에서는 함수 호출이 함수를 정의하는 문장보다 앞에 있건 뒤에 있건 유효범위만 같으면 문제가 없습니다.

2. 역사 주/ 함수 `var` 문과 함수값을 사용하는 문장과 관련해서 IE는 조금 다르게 동작합니다.

다음의 예를 보기 바랍니다.

```
notPreDefined();  
  
var notPreDefined = function notPreDefined () {  
    alert("ok");  
};
```

이 예는 `notPreDefined()` 부분에 함수가 정의되지 않았다는 오류가 발생해야 정상입니다. 왜냐하면 아래에 있는 함수 정의가 함수 문장으로 정의된 것이 아니기 때문입니다. 하지만 IE에서는 함수 표현식에 함수 이름을 지정안하면 함수 문장으로 정의한 것처럼 해석하는 것으로 보입니다. 그래서 위의 코드를 IE에서 확인하면 오류가 나지 않습니다. 하지만 함수 정의 부분을 다음과 같이 하여 함수 이름을 빼면 오류가 발생합니다.

```
var notPreDefined = function () {  
    alert("ok");  
};
```

공식적인 문법은 function이라는 단어로 시작하는 문장을 함수 문장이라고 가정하고 있기 때문에 문장의 첫 번째 부분에 함수 표현식을 사용할 수 없습니다. 이를 위한 대안은 함수 표현식을 다음의 예처럼 괄호로 묶는 것입니다.

```
(function () {
    var hidden_variable;

    // 이 함수는 환경에 영향을 미치는 점들이 있지만
    // 새로운 전역변수를
    // 추가하지는 않음.
})();
```

## 10 | 데이터 타입 랙퍼

자바스크립트에는 일련의 데이터 타입 랙퍼들이 있습니다. 예를 들어 다음과 같은 문장은 랙핑된 값을 반환하는 valueOf 메소드를 가진 객체를 생성합니다.

```
new Boolean(false)
```

이러한 랙퍼들은 완전히 필요가 없으며 때때로 혼란을 줄 수 있다는 것이 판명됐습니다. new Boolean, new Number, new String 등을 사용하지 마십시오.

또한 new Object와 new Array 사용도 피하고 {}와 []를 사용하기 바랍니다.

## 11 | new

자바스크립트의 new 연산자는 피연산자의 프로토타입 멤버들을 상속하는 객체를 생성하고 이 객체를 this에 바인딩하면서 피연산자를 호출합니다. 이러한 호출은 new의 피연산자(생성자 함수)가 새로 만들어진 객체를 원하는 대로 커스터마이징하여 반환할 수 있는 기회를 제공합니다.

만약 new 연산자를 빼먹게 되면 일반적인 함수 호출을 하게 되고 이때 this는 새로운 객체가 아니라 전역객체에 바인딩됩니다. 이렇게 되면 새로운 속성을 초기화 할 때 전역변수에 접근하게 됩니다. 아주 안 좋은 결과입니다. new를 안 써도 컴파일 시나 실행 시에 어떠한 경고도 없습니다.

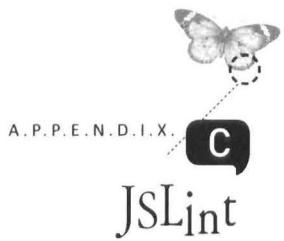
new와 함께 사용하기 위해서 만든 함수의 이름은 각 단어의 첫 글자를 대문자로 표기하고 그 외의 것들에는 이러한 표기법을 사용하지 않는 것이 좋습니다(파스칼 표기법). 이러한 코딩 규칙을 사용함으로써 new를 생략한 실수를 찾을 수 있는 단서를 제공할 수 있습니다.

물론 더 나은 정책은 new 연산자를 사용하지 않는 것입니다.

## 12 | void

많은 언어에서 void는 아무 값도 없는 데이터 타입입니다. 자바스크립트에서 void는 피연산자를 취한 후 undefined를 반환하는 연산자입니다. 이는 유용하지도 않고 매우 혼동됩니다. void 역시 피하는 것이 좋습니다.





무슨 착오(error)가 있어서 보는 것, 듣는 것이 뒤죽박죽 된 것일까?

- 윌리엄 세익스피어, 착오 희극



C 언어가 나온지 얼마 안 됐을 때, 초기의 컴파일러가 잡아내지 못하는 몇몇 공통적인 프로그래밍 오류가 있었습니다. 그래서 소스 파일을 검색하여 이러한 오류들을 찾아주는 lint라는 부가적인 프로그램이 개발됐습니다.

C가 점점 발전하면서, 언어의 정의는 몇몇 안전하지 않은 요소들을 제거함으로써 강화되었고, 컴파일러는 경고를 알려주는 부분에서 더욱 발전하였습니다. 결국 lint는 더 이상 필요하지 않게 됐습니다.

자바스크립트는 나이가 어린 언어입니다. 자바스크립트는 무거운 자바를 쓰기에 적합하지 않은 웹 페이지 내의 작은 일들을 처리하도록 고안됐습니다. 하지만 자바스크립트는 매우 재능 있는 언어이고 현재 꽤 큰 프로젝트에서도 사용되고 있습니다. 언어를 쉽게 사용할 수 있도록 고안된 많은 속성들은 큰 프로젝트에서는 골치거리로 작용하고 있습니다. 그래서 자바스크립트를 위한 lint가 필요한데, 자바스크립트의 구문을 확인하고 검증해주는 JSLint가 바로 그것입니다.

JSLint는 자바스크립트의 코드 품질을 높이기 위한 툴입니다. 이 툴은 소스 텍스트를 검사하고, 만약 문제점을 발견하면, 문제점을 설명하는 메시지와 소스의 대략적인 위치를 반환합니다. JSLint가 보고하는 문제점은 구문 오류뿐만이 아닙니다. JSLint는 코딩 스타일과 구조적인 문제도 살펴봅니다. 이 툴은 검사하는 프로그램 전체가 온전히 문제 없다는 것을 증명하지는 못합니다. 다만 문제가 있을만한 부분을 바라보는 또 다른 시각을 제공합니다.

JSLint는 자바스크립트의 전문적인 부분집합을 정의하는데 이 부분집합은 ECMAScript 언어 명세 3번째 판에 정의돼 있는 것보다 훨씬 엄격한 부분집합입니다. 이 부분집합은 9장에서 권고한 스타일과 거의 유사합니다.

자바스크립트는 영성해 보이지만, 그 내부에는 우아하고 훨씬 좋은 언어가 내재돼 있습니다. JSLint는 이러한 좋은 점들만을 사용하고 대부분의 영성한 부분은 피할 수 있게 도와줍니다.

JSLint는 <http://www.JSLint.com/>에서 사용할 수 있습니다.

## 01 | 정의되지 않은 변수와 함수

자바스크립트의 가장 큰 문제점은 전역변수에 의존하고 있다는 점입니다. 특히 묵시적으로 선언되는 전역변수들은 더욱 문제입니다. 변수가 명시적으로 선언되지 않으면(var 문을 사용하여 선언되지 않으면), 자바스크립트는 이 변수를 전역변수로 간주합니다. 이런 특성은 오탏가 난 이름이나 그 외 다른 문제들을 덮어버릴 수 있습니다.

JSLint는 모든 변수나 함수를 사용하거나 호출하기 전에 반드시 선언해야 함을 원칙으로 하고 있습니다. 이러한 원칙은 묵시적인 전역변수를 찾을 수 있게 해줍니다. 또한 프로그램을 보다 읽기 쉽게 만들어줍니다.

때때로 프로그램 파일 하나가 다른 파일에 정의돼 있는 전역변수나 함수에 의존적일 수 있습니다. 이런 경우에는 자신의 프로그램이나 스크립트 파일 내에 정의돼 있지 않고 다른 곳에 있는 전역변수나 객체를 주석으로 표기하여 JSLint에게 알려줄 수 있습니다.

전역 선언 주석은 의도적으로 사용하는 모든 전역변수의 이름을 열거하는데도 사용할 수 있습니다. JSLint는 var 문을 생략했거나 잘못 입력한 이름을 구분할 때 이 정보를 참조합니다. 전역 선언 주석은 다음과 같은 형태입니다.

```
/*global getElementByAttribute, breakCycles, hanoi */
```

전역 선언 주석은 /\*global로 시작합니다. \*와 g 사이에 빈 칸이 없다는 것에 주의해야 합니다. /\*global 주석은 원하는 만큼 사용할 수 있습니다. 이 주석은 반드시 명시한 변수를 사용하기 전에 나타나야 합니다.

일부 전역 요소들은 정의돼 있는 것으로 가정할 수 있습니다(03. 옵션 절 참조). 브라우저라고 가정하는 browser 옵션은 웹 브라우저에서 기본으로 제공하는 window, document, alert 같은 표준 전역 속성들이 정의돼 있다고 가정하고 검사합니다. Rhino로 가정하게 하는 rhino 옵션은 Rhino 환경에서 제공하는 전역 속성들이 정의돼 있다고 가정하고, Yahoo! 위젯으로 가정하는 widget 옵션은 Yahoo! 위젯 환경에서 제공하는 전역 요소들이 정의돼 있다고 가정합니다.

## 02 | 잘못 사용한 이름 찾기

자바스크립트는 엄격하지 않은 타입 체크를 하는 동적 객체 언어이기 때문에 컴파일 시에 속성 이름이 오타 없이 제대로 사용됐는지 알 수가 없습니다. JSLint는 이러한 점과 관련하여 어느 정도 도움이 됩니다.

JSLint는 보고서의 마지막에 `/*members*/` 주석을 표시합니다. 이 주석에는 .(마침표) 표기법과 배열첨자 표기법, 객체 리터럴에서 속성의 이름 등에 사용된 모든 이름과 문자열 리터럴들이 포함됩니다. 이 목록을 통해서 오타가 난 이름이 없나 확인할 수 있습니다. 한 번만 사용된 이름은 이탤릭체로 표기해서 잘못된 부분을 쉽게 찾을 수 있게 합니다.

`/*members*/` 주석을 아래와 같이 스크립트 파일 내에 복사할 수 있습니다. 그러면, JSLint는 이 목록과 프로그램 코드에 사용된 속성 이름의 철자를 대조하게 됩니다. 이러한 방법으로 JSLint가 잘못된 이름을 찾을 수 있습니다.

```
/*members doTell, iDoDeclare, mercySakes,  
myGoodness, ohGoOn, wellShutMyMouth */
```

## 03 | 옵션

JSLint는 사용자에게 맞는 자바스크립트 부분집합을 선택할 수 있게 옵션 객체를 받도록 구현돼 있습니다. 옵션은 스크립트의 소스 내에서도 지정할 수 있습니다.

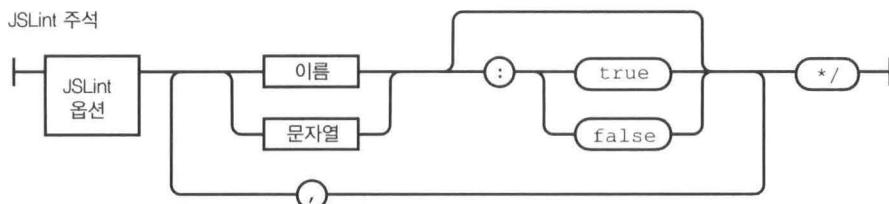
스크립트 내에서의 옵션은 다음과 같은 형태로 지정할 수 있습니다.

```
/*jslint nomen: true, evil: false */
```

옵션의 설정은 `/*jslint`로 시작합니다. \*와 j 사이에 빈칸이 없습니다. 설정 내용은 이름/값 쌍이 이어지는 형태입니다. 이름은 JSLint 옵션의 이름이고 값은 `true`나 `false`입니다. 주석을 통한 옵션 설정은 옵션 객체보다 우선 순위를 갖습니다. 모든 옵션의 기본값은 `false`입니다. [표 C-1]은 JSLint가 사용할 수 있는 옵션들을 보여 줍니다.

[표 C-1] JSLint 옵션

옵션	의미
adsafe	true면 ADsafe.org의 규칙이 강제로 적용됨
bitwise	true면 비트 연산자를 허용하지 않음
browser	true면 표준 브라우저 전역 속성들이 정의돼 있다고 가정
cap	true면 대문자 HTML을 허용
debug	true면 debugger 문을 허용
eqlqqeq	true면 ===를 사용해야 함
evil	true면 eval 허용
for in	true면 필터링하지 않은 for in 문 허용
fragment <sup>1</sup>	true면 HTML의 일부분에 들어있는 자바스크립트 검사를 허용
glovar	true면 전역변수를 선언하는데 var를 사용하지 않아도 됨
laxbreak	true면 if 문이 분리되는 것을 확인하지 않음
nomen	true면 이름들을 확인
on	true면 HTML 이벤트 핸들러 허용
passfail	true면 첫 번째 오류에서 검사를 멈춤
plusplus	true면 ++와 --를 허용하지 않음
rhino	true면 Rhino 환경 전역 요소들이 있다고 가정
undef	true면 정의되지 않은 전역변수는 오류
white	true면 엄격한 공백 whitespace 규칙 적용
widget	true면 Yahoo! 위젯의 전역 요소들이 정의돼 있다고 가정



- 역자 주/ fragment 옵션은 다음과 같이 HTML에서 자바스크립트가 포함된 일부분만을 검사할 때 설정하는 옵션입니다.

```
<HTML태그>
<script>
    자바스크립트 코드
<script>
</HTML태그>
```



## 04 | 세미콜론

자바스크립트는 C 유형의 구문을 사용합니다. 그래서 문장을 구분하기 위해서 세미콜론을 사용합니다. 자바스크립트는 세미콜론 삽입 메커니즘으로 세미콜론이 사용되지 않았다고 생각되는 곳에 알아서 삽입합니다. 그런데 이러한 메커니즘은 위험합니다.

C처럼 자바스크립트도 ++, --, ( 연산자가 있는데 이 연산자들은 피연산자의 앞에 올 수도 있고 뒤에 올 수도 있습니다. 앞에 쓰인 것인지 뒤에 쓰인 것인지를 명확히 하는 것은 세미콜론입니다.

자바스크립트에서 라인피드(LF)는 공백 문자 whitespace)일 수도 있고 세미콜론으로 동작할 수도 있습니다. 이러한 모호함은 의도와는 달리 서로가 뒤바뀔 여지가 있습니다.

JSLint는 for, function, if, switch, try, while을 제외한 모든 문장의 끝에는 :가 붙어 있어야 한다고 규정합니다. JSLint는 불필요한 세미콜론이나 빈 문장은 잘못된 것이라고 간주합니다.

## 05 | 줄 바꿈

세미콜론 자동 삽입 메커니즘으로 인해 발생하는 오류에 대해 좀더 방어적이 되기 위해서, JSLint는 긴 문장을 줄 바꿈할 때는 반드시 다음과 같은 구두점 문자나 연산자 다음에 해야 하는 것을 원칙으로 하고 있습니다.

```
, . ; : { } ( [ = < > ? ! + - * / % ^ | &
== != <= >= += -= *= /= %= ^= |= &= << >> || &&
==== !== <<= >>= >>> >>>=
```

JSLint는 긴 문장이 식별자, 문자열, 숫자, 클로저 및 다음과 같은 접미 연산자 다음에 줄 바꿈하는 것을 허용하지 않습니다.

```
) ] ++ --
```

이러한 규칙들 없이 자유로운 줄 바꿈을 사용하려면 laxbreak 옵션을 켜면 됩니다.

세미콜론 자동 삽입은 카피 앤 페이스트(copy/paste)로 발생하는 오류를 덮어버릴 수가 있습니다. 항상 연산자 다음에 줄 바꿈을 하면 JSLint가 이러한 오류를 찾는데 도움이 됩니다.

## 06 | 쉼표

쉼표 연산자는 지나치게 까다로운 표현식의 원인이 될 수 있습니다. 이 연산자는 몇몇 프로그래밍 오류를 숨길 수도 있습니다.

JSLint는 쉼표를 연산자가 아니라 구분자로 사용한 것만을 허용합니다(for 문의 초기화 부분과 증가 부분은 예외입니다). JSLint는 배열 리터럴에서 쉼표만 있고 배열 요소를 생략하는 것을 허용하지 않습니다([1,2,3,]나 [1,2,,3] 같은 경우). 불필요한 쉼표도 사용해서는 안 됩니다. 쉼표는 배열이나 객체 리터럴의 마지막 요소 다음에 위치해서는 안 됩니다. 이렇게 하면 몇몇 브라우저에서는 제대로 해석하지 못합니다.

## 07 | 필수 블록

JSLint는 if 문과 for 문이 실행하는 문장들을 반드시 중괄호로 묶어 블록으로 만들어야 한다고 규정합니다.

자바스크립트는 다음과 같은 코드를 허용합니다.

```
if (condition)
    statement;
```

이러한 유형은 여러 프로그래머가 같은 코드로 함께 작업하는 프로젝트에서 오류를 발생시키기 쉽다는 것이 판명됐습니다. 그래서 JSLint는 다음과 같은 형식을 고수합니다.

```
if (condition) {
    statements;
}
```

이러한 형식이 훨씬 좋다는 것은 경험이 말해주고 있습니다.

## 08 | 금지된 블록

수많은 언어에서 블록은 유효범위(scope)를 갖습니다. 블록 내에서 정의된 변수는 블록 바깥에서 볼 수 없습니다.

자바스크립트에서는 블록이 유효범위를 갖지 않습니다. 단지 함수 유효범위만 존재합니다. 함수 내에서 정의된 변수는 어디에서 정의됐던지 상관 없이 함수 내 모든 곳에서 볼 수 있습니다. 이러한 자바스크립트 블록은 경험 많은 프로그래머들에게 혼란을 주고, 블록 유효범위를 가질 거 같아 보이지만 실제로는 그렇지 않기 때문에 오류를 유발할 수 있습니다.

JSLint는 function 문, if 문, switch 문, while 문, for 문, do 문, try 문에서만 블록을 허용하고 그 외 다른 곳에는 사용하는 것을 허용하지 않습니다. 물론 else if 문이나 for in 문에도 블록을 허용합니다.

## 09 | 표현식 문장

표현식 문장은 할당, 함수나 메소드 호출, delete에만 허용합니다. 그 외의 다른 모든 표현식 문장은 오류로 간주합니다.

## 10 | for in 문

for in 문은 객체가 가진 모든 속성의 이름들을 열거합니다. 불행히도 for in 문은 프로토타입 체인을 통해 상속된 모든 멤버들도 열거합니다. 또한 for in 문은 데이터 속성들에만 초점을 맞춰 열거하고자 할 때 메소드들까지 모두 열거해주는 안 좋은 점이 있습니다.

모든 for in 문장의 몸체는 필터링 기능을 하는 if 문에 뷰이는 것이 좋습니다. If 문을 사용하여 필터링을 하면 특정한 타입이나 범위의 값만 선택할 수도 있고, 함수들을 제외할 수도 있으며, 프로토타입에 있는 속성들을 제외할 수 있습니다. 예를 들면 다음과 같이 할 수 있습니다.

```
for (name in object) {
    if (object.hasOwnProperty(name)) {
        ...
    }
}
```

## 11 | switch 문

switch 문을 사용하면서 발생하는 가장 일반적인 오류는 각각의 case 절 마지막에 break 문을 두는 것을 잊어서 의도하지 않았는데도 다음 case 절까지 실행(fall-through)을 하는 것입니다. JSLint는 다음에 오는 case나 default 바로 전의 문장은 break, return, throw 중에 하나만을 허용합니다.

## 12 | var 문

자바스크립트는 함수 내의 어느 곳에서도 var를 사용하여 변수를 정의하는 것을 허용합니다. 하지만 JSLint는 보다 엄격합니다.

JSLint가 허용하는 사항은 다음과 같습니다.

- 변수는 한 번만 선언하고, 사용하기 전에 var 문을 사용하여 선언할 것.
- 함수는 사용 전에 선언할 것.
- 매개변수를 일반 변수로 재차 선언하지 말 것.

그리고 JSLint가 허용하지 않는 사항은 다음과 같습니다.

- arguments 배열을 var 문으로 선언하는 것.
- 변수를 블록 내에서 선언하는 것. 이유는 자바스크립트의 블록은 블록 유효 범위를 갖지 않기 때문. 블록 내에서 변수를 선언하면 예기치 못한 결과를 가져올 수 있기 때문에 모두 변수는 함수 몸체의 상단에서 선언하는 것이 좋음.

## 13 | with 문

with 문의 의도는 깊게 중첩된 객체의 속성들을 간단하게 접근하는 것입니다. 그런데 이런 의도와는 달리 새로운 객체 속성을 설정할 때 매우 안 좋게 동작합니다. with 문을 결코 사용하지 마십시오. 대신에 변수를 직접 사용해야 합니다.

JSLint는 with 문을 허용하지 않습니다.

## 14 | =

JSLint는 if 문이나 while 문의 조건절에서 할당문을 사용하는 것을 허용하지 않습니다. 왜냐하면 다음과 같은 코드가,

```
if (a = b) {  
    ...  
}
```

실제로는 다음과 같은 조건절을 의도했는데 실수로 할당문을 사용한 것처럼 보이는 경우가 많기 때문입니다.

```
if (a == b) {  
    ...  
}
```

만약 조건절에서 할당문이 필요한 경우에는 다음과 같이 괄호로 묶어줍니다.

```
if ((a = b)) {  
    ...  
}
```

## 15 | ==과 !=

==과 != 연산자는 비교 전에 타입 변환을 합니다. 이러한 특성은 ‘\f\r\n\t’ == 0과 같은 경우를 참으로 인식하기 때문에 좋지 않습니다. 이 연산자들은 데이터 타입 오류를 덮어버릴 수 있습니다.

다음과 같은 값들을 비교할 때는 항상 ===나 !==를 사용해야 합니다. 이 연산자들은 타입을 변경하지 않고 비교합니다.

```
0 '' undefined null false true
```

만약 ==나 !=를 사용하고 싶다면 짧은 형식을 사용하는 것이 좋습니다. 즉 다음과 같이 하는 대신에,

```
(foo != 0)
```

다음과 같이 사용합니다.

```
(foo)
```

그리고 다음과 같이 하는 대신에,

```
(foo == 0)
```

다음과 같이 하면 됩니다.

```
(!foo)
```

항상 ==과 !== 연산자 사용을 우선 순위에 두는 것이 좋습니다. eqeqeq 옵션(==과 !=를 허용하지 않음)을 설정하면 모든 경우에 ===와 !==만 허용합니다.

## 16 | 라벨

자바스크립트에서는 모든 문장이 라벨을 가질 수 있고 라벨은 별도의 이름공간(namespace)을 갖습니다. JSLint는 이와 관련하여 좀더 엄격합니다.

JSLint는 break 문으로 지정되는 switch 문, while 문, do 문, for 문에서만 라벨 사용을 허용합니다. 라벨은 변수나 매개변수와 구분되어야 합니다.

## 17 | 다음에 오는 코드가 정해진 문장

return 문, break 문, continue 문, throw 문 등은 그 다음에 반드시 }나 case 또는 default가 와야 합니다.

## 18 | +와 -의 혼동

JSLint는 + 다음에 +나 ++가 오거나 - 다음에 -나 --가 오는 것을 허용하지 않습니다. 잘못 위치한 빈칸은 + +를 ++로 변경할 수 있고 이렇게 발생한 오류는 찾기가 어렵습니다. 이러한 혼동을 피하기 위해서 괄호를 사용하는 것이 좋습니다.

## 19 | ++와 --

++(증가)와 --(감소) 연산자는 까다로운 부분을 야기하는 나쁜 코드의 원인으로 잘 알려져 있습니다. 이러한 연산자는 바이러스나 그 외 다른 보안적 위험을 낳는 잘못된 아키텍처에 벼금갈 정도로 안 좋은 것입니다. JSLint의plusplus 옵션을 설정하면 이 연산자들 사용을 금합니다.

## 20 | 비트 연산자

자바스크립트는 정수형 데이터 타입이 없음에도 불구하고 비트 연산자가 있습니다. 비트 연산자는 피연산자를 부동 소수점에서 정수형으로 돌려 연산을 한 후 다시 되돌립니다. 그래서 자바스크립트의 비트 연산은 C나 그 밖의 다른 언어들에서처럼 효율적이지 않습니다. 이들은 브라우저 애플리케이션에서는 거의 유용하지 않습니다. 논리 연산자와 비슷한 점은 오히려 일부 프로그래밍 오류를 감출 수 있습니다. `bitwise` 옵션을 설정하면 비트 연산자를 허용하지 않습니다.

## 21 | eval

`eval` 함수와 그의 사촌들(`Function`, `setTimeout`, `setInterval`)은 자바스크립트 컴파일러를 접근하게 합니다. 이 함수들은 때때로 유용하지만, 대부분의 경우 아주 나쁜 코딩의 단초가 될 수 있습니다. `eval` 함수는 자바스크립트에서 가장 오용하고 있는 기능 중 하나입니다.

## 22 | void

C 계열의 언어 대부분에서 `void`은 데이터 타입입니다. 자바스크립트에서 `void`은 항상 `undefined`를 반환하는 전치 연산자입니다. `void`은 혼란만을 가중하고 유용하지 않기 때문에 JSLint는 이 연산자를 허용하지 않습니다.

## 23 | 정규 표현식

정규 표현식은 간결하고 암호처럼 보이는 표기법으로 작성됩니다. JSLint는 잠재적인 문제를 야기할 수 있는 문제들을 찾습니다. 또한 JSLint는 명시적인 이스케이프를 권고함으로써 가시적인 모호함을 해결하려고 합니다.

정규 표현식 리터럴을 위한 자바스크립트의 구문은 /를 너무 많이 사용하게 합니다. 모호성을 피하기 위해 JSLint는 정규 표현식 리터럴 앞에 올 수 있는 문자로 ( = :, , 만을 허용합니다.

## 24 | 생성자와 new

생성자는 new와 함께 사용되도록 고안된 함수입니다. 전치어 new는 함수의 프로토타입에 근거하여 새로운 객체를 생성하고 이를 함수의 묵시적인 this 매개변수에 바인딩합니다. 만약 new를 빼먹게 되면 새로운 객체는 생성되지 않고 this는 전역객체에 바인딩됩니다. 이것은 심각한 실수입니다.

JSLint는 머리 글자를 대문자로 사용하는 표기법으로 생성자 함수의 이름을 짓도록 강요합니다. JSLint는 이름의 머리 글자가 대문자인 함수를 new 없이 호출하는 것을 허용하지 않습니다. 또한 이와는 반대로 이름의 머리 글자가 대문자가 아닌 함수가 new와 함께 호출되는 것도 허용하지 않습니다.

JSLint는 랙퍼 형식인 new Number, new String, new Boolean을 허용하지 않습니다.

JSLint는 new Object도 허용하지 않습니다(대신에 {} 사용).

JSLint는 new Array도 허용하지 않습니다(대신에 [] 사용).

## 25 | 예측하지 않음

JSLint는 변수가 사용되기 전에 값이 할당되는지를 분석하지 않습니다. 왜냐하면 많은 프로그램에서 당연한 기본값으로 생각하는 undefined가 기본값으로 할당되기 때문입니다.

JSLint는 전역적인 분석을 하지 않습니다. new와 같이 사용된 함수가 실제 생성자인지 또는 메소드 이름의 철자가 맞았는지 등을 확인하려고 하지 않습니다(생성자와 관련해서는 대문자 표기법은 확인합니다).

## 26 | HTML

JSLint는 HTML 텍스트를 다룰 수 있습니다. <script> ... </script> 태그 안과 이벤트 핸들러로 지정된 자바스크립트를 분별해낼 수 있습니다. 또한 다음과 같은 내용들을 조사하여 자바스크립트를 방해하는 것으로 알려진 문제들을 찾을 수 있습니다.

- 모든 태그 이름이 소문자인지 확인
- 닫는 태그가 있는 모든 태그에 닫는 태그가 있는지 확인
- 모든 태그가 제대로 중첩됐는지 확인
- 리터럴 <를 위해 엔티티 &lt;가 사용됐는지 확인

JSLint는 XHTML이 요구하는 수준까지 확인하지는 않지만 유명한 브라우저들보다 더 엄격합니다.

JSLint는 또한 문자열 리터럴에서 </가 나타나는지도 검사합니다. 항상 </ 대신에 <\>를 사용해야 합니다. 부가적인 \를 자바스크립트 컴파일러는 무시하지만 HTML 파서는 그렇지 않습니다.

옵션 중에는 대문자 태그를 허용하는 옵션이 있습니다. 또한 인라인 HTML 이벤트 핸들러를 허용하는 옵션도 있습니다.

## 27 | JSON

JSList는 또한 JSON 데이터 구조가 제대로 구성됐는지 확인할 수 있습니다. JSList가 살피는 첫 글자가 {나 [면 JSList는 JSON 규칙을 엄격하게 적용합니다. JSON은 부록 E에 자세히 설명합니다.

## 28 | 보고서

JSList가 분석을 모두 마치면 함수 보고서를 생성합니다. 이 보고서는 각각의 함수에 대해 다음의 항목들을 열거합니다.

- 함수가 시작하는 줄 번호
- 함수 이름. 함수 이름이 주어지지 않은 함수(anonymous)이면 함수 이름을 예측하여 표시
- 매개변수
- Closure: 함수 내에 선언되어 내부 함수에서 사용되는 변수와 매개변수.
- Variables: 함수 내에서 선언되어 함수에서만 사용되는 변수
- Unused: 함수 내에서 선언됐지만 사용되지 않은 변수. 이로 인해 오류를 발견할 수도 있음
- Outer: 함수 내에서 사용되지만 다른 함수에서 선언된 변수
- Global: 함수에서 사용된 전역변수
- Label: 함수에서 사용된 라벨

또한 보고서에는 사용된 모든 속성 이름이 포함됩니다.





A.P.P.E.N.D.I.X.

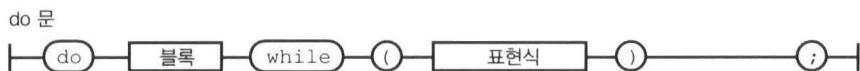
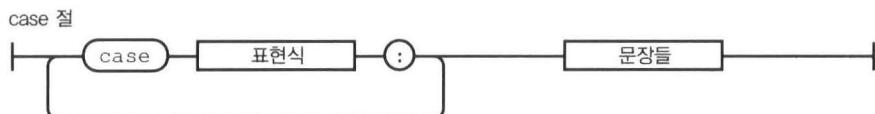
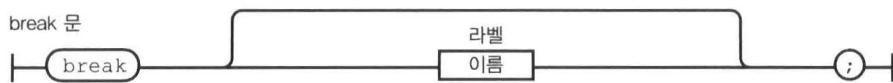
D

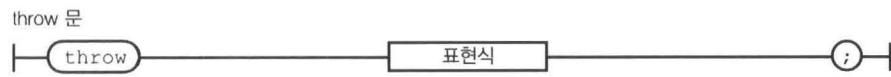
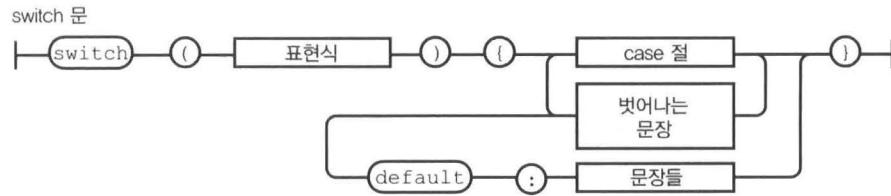
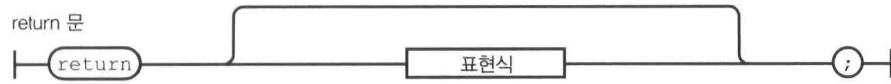
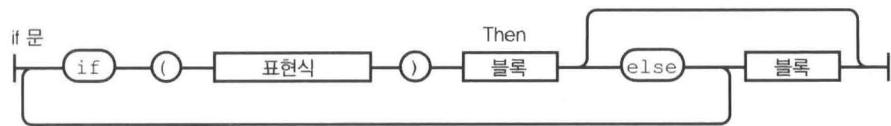
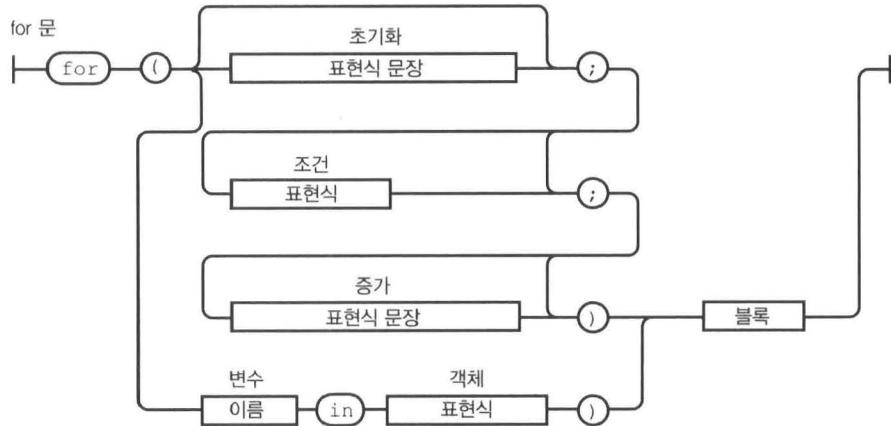
## 구문 다이어그램

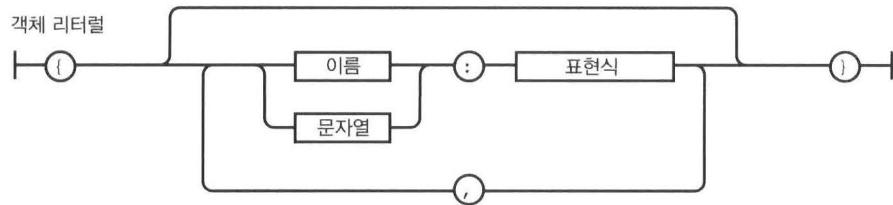
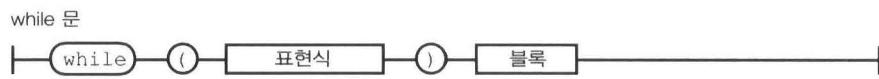
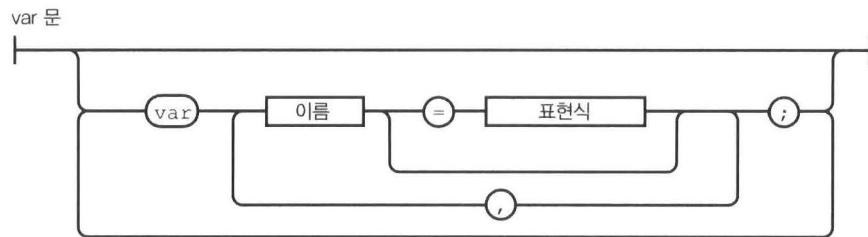
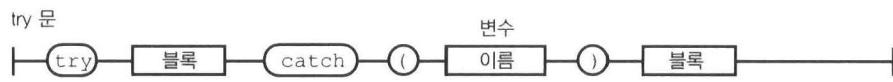
그렇게 몸짓만으로 말을 하는 비탄의 도면 같은 너는!

- 윌리엄 셰익스피어, 타이터스 앤드로니커스

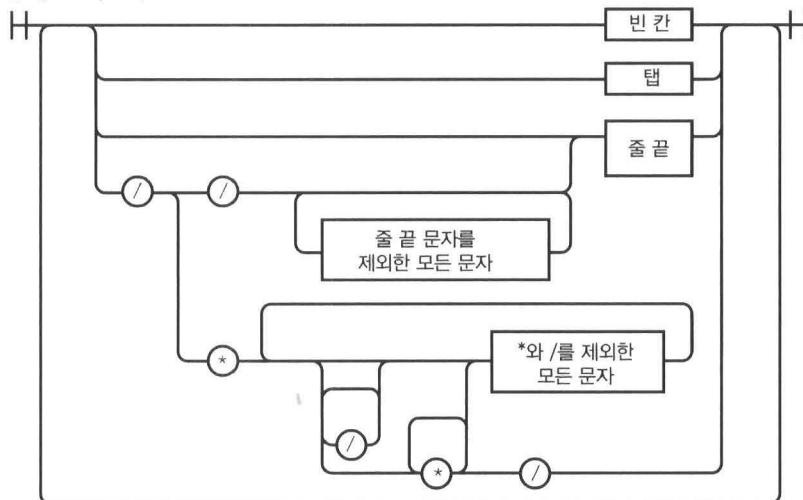
구문 다이어그램을 참고하기 쉽게 A~Z 그리고 가~하 순서로 배치했습니다.



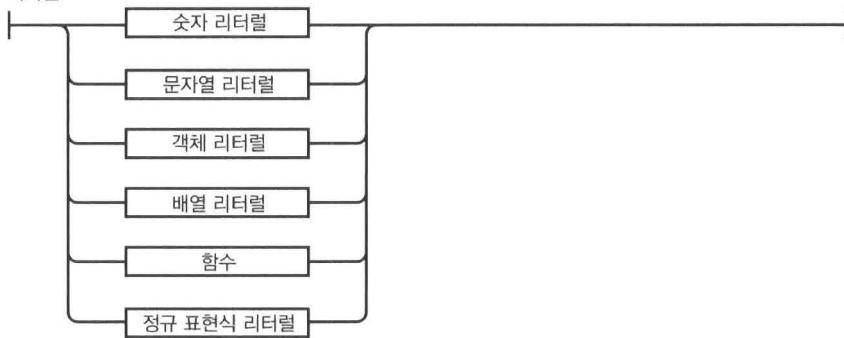




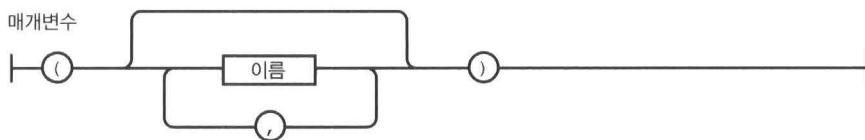
### 공백 whitespace)

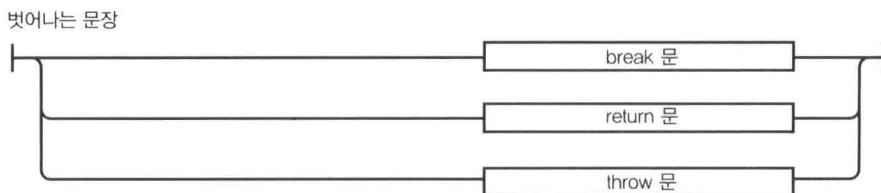
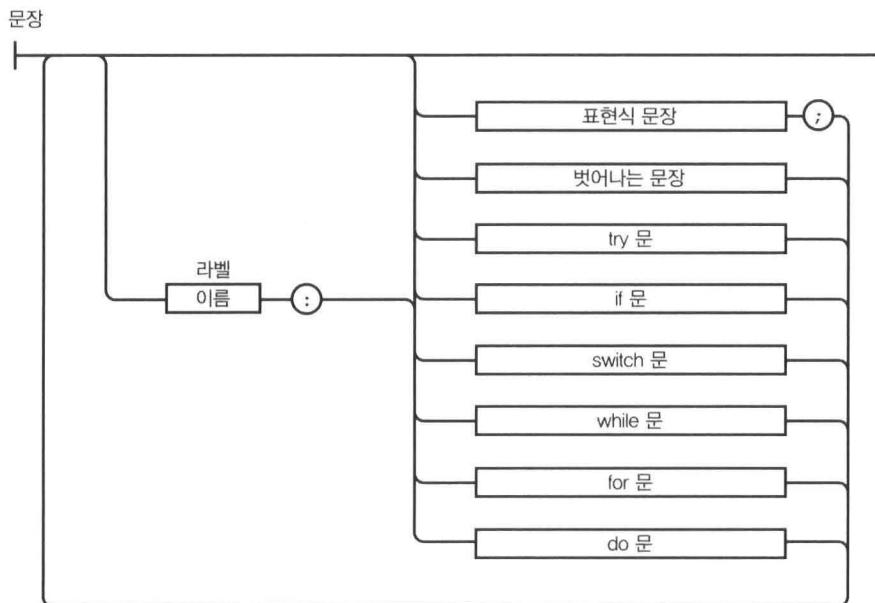
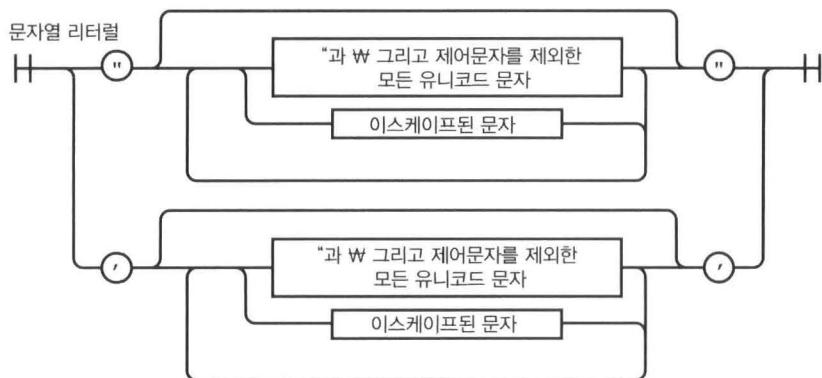


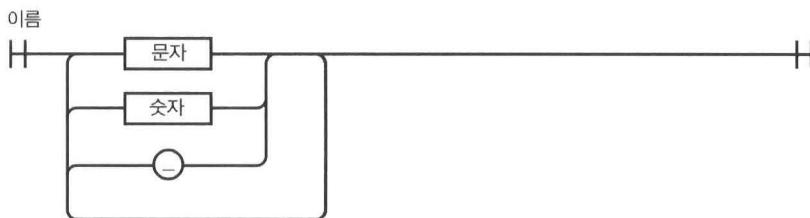
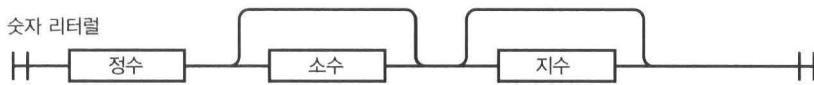
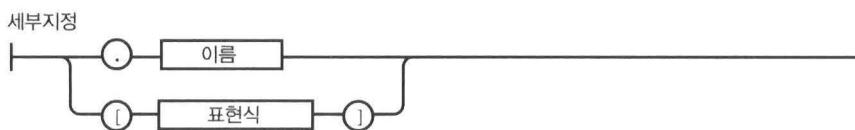
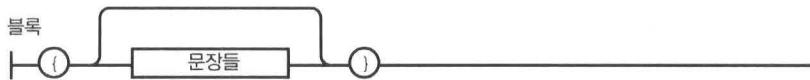
### 리터럴



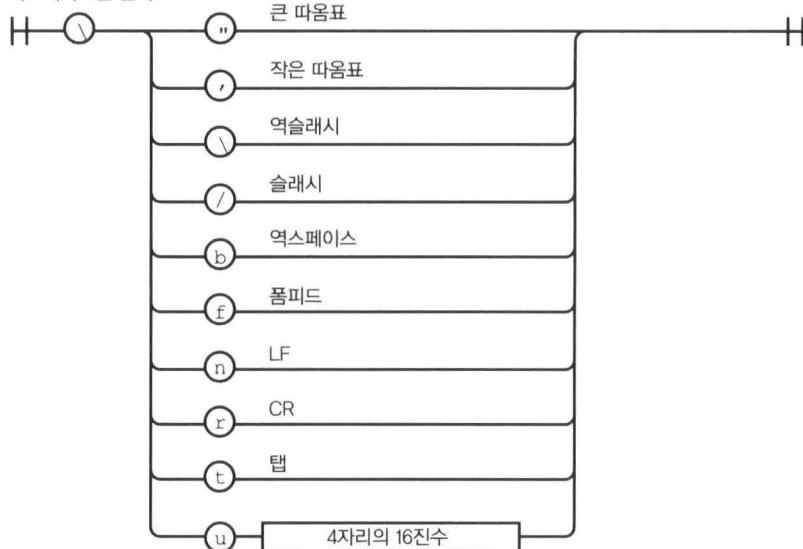
### 매개변수



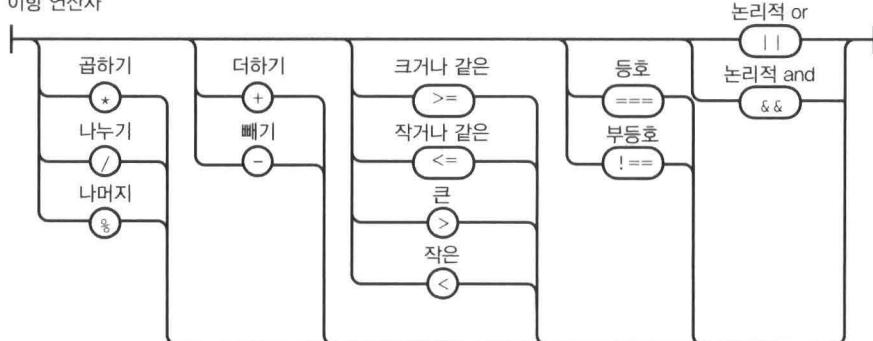




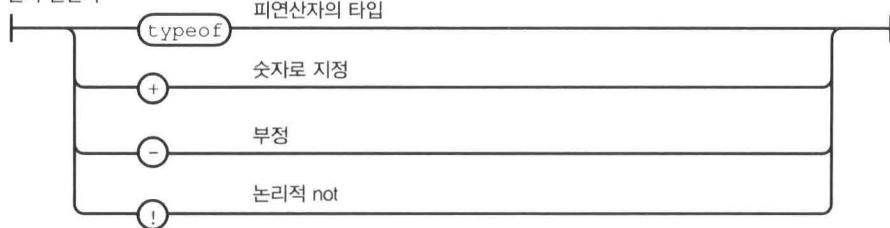
### 이스케이프된 문자

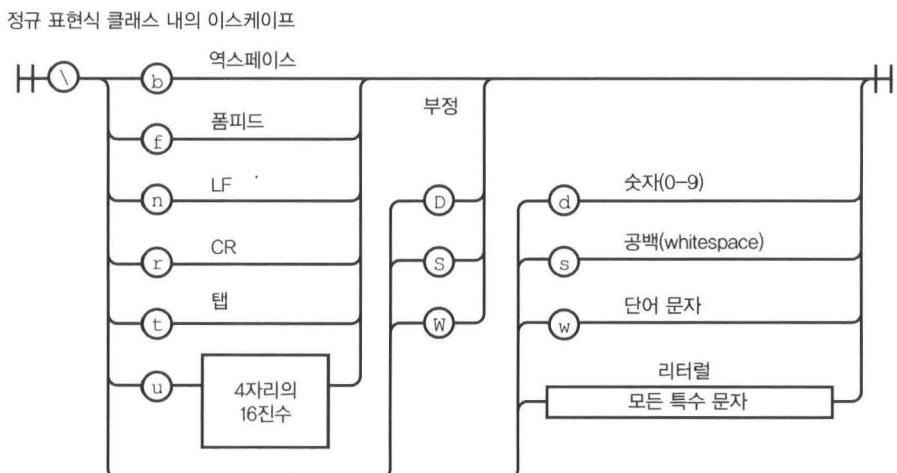
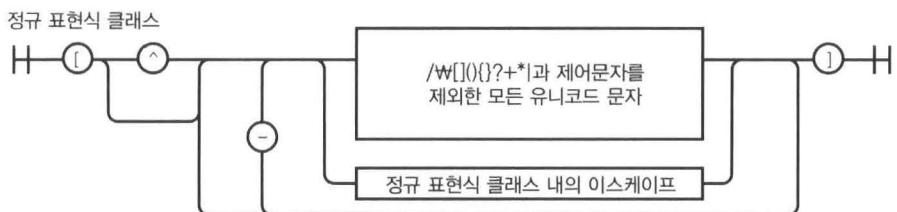
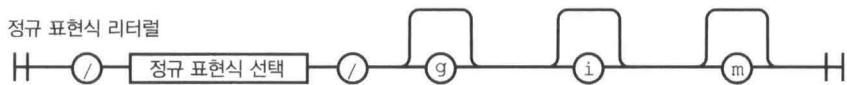
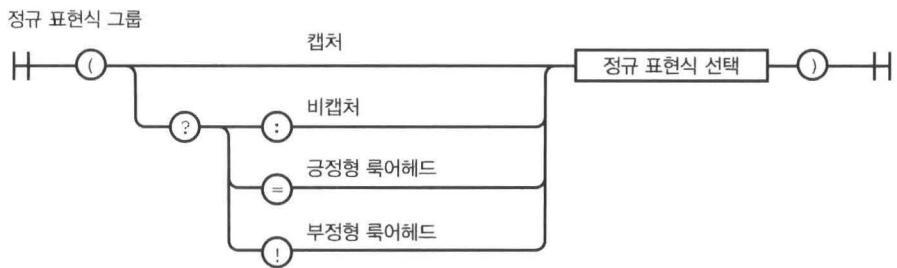


### 이항 연산자



### 전치 연산자

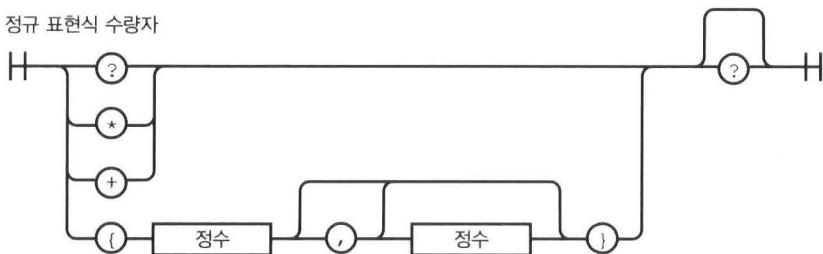




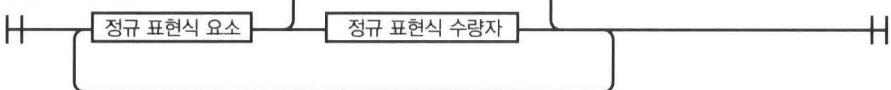
정규 표현식 선택



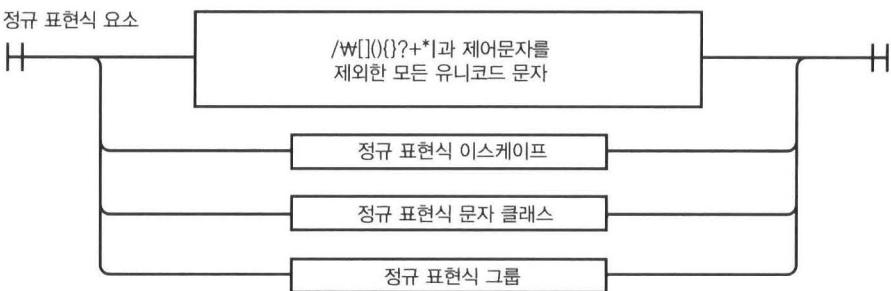
정규 표현식 수량자



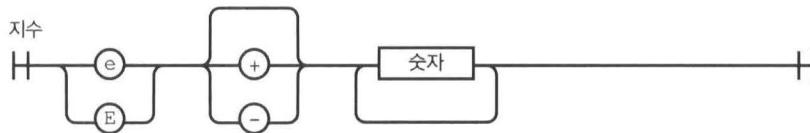
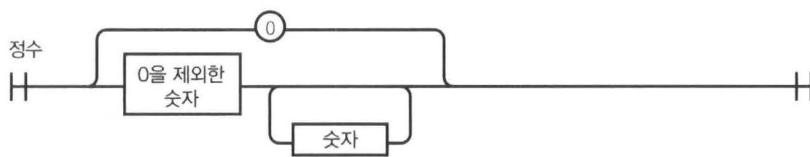
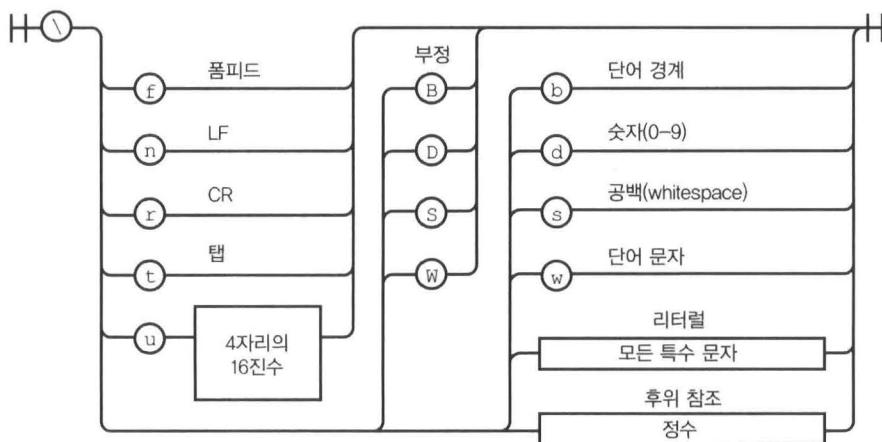
정규 표현식 시퀀스

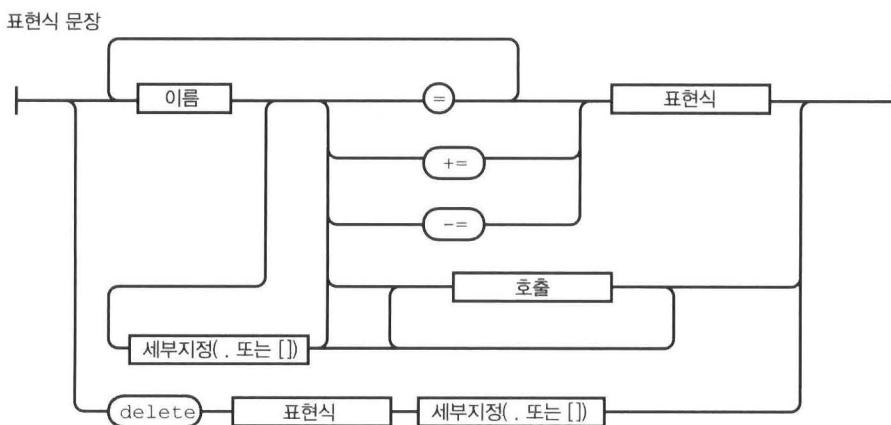
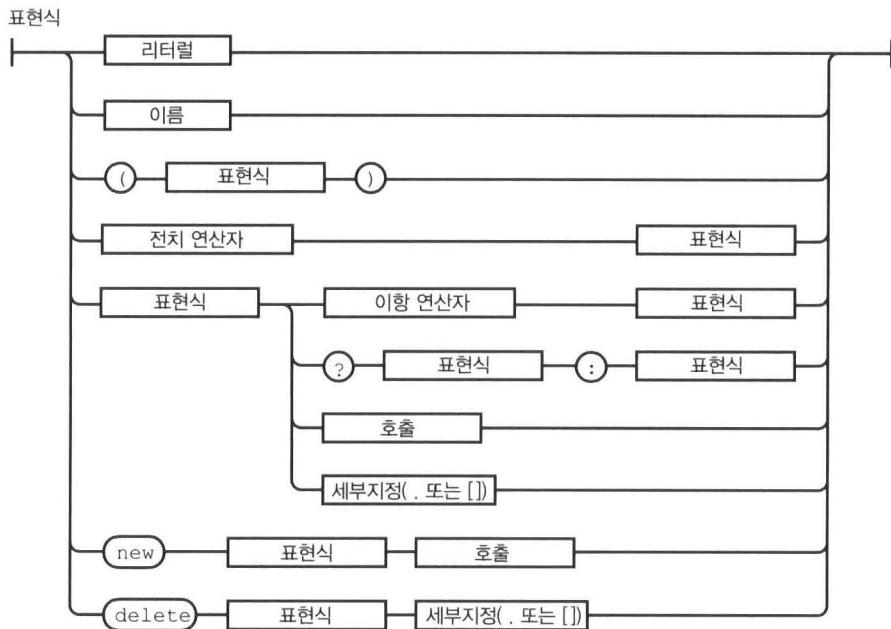


정규 표현식 요소

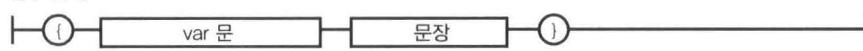


정규 표현식 이스케이프

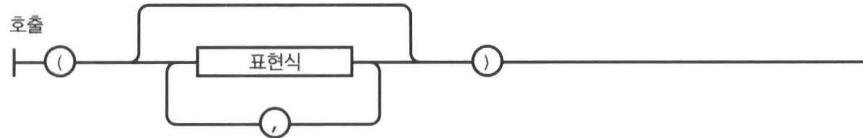




함수 몸체



호출





A.P.P.E.N.D.I.X.

E

## JSON

그만 가보겠다. 이렇게 절박한 비상 사태 하에서는 오랜만에 만난 친구로서 흥금을 털어놓고 즐겁게 얘기를 할래야 할 수가 없구나. 신은 그러한 단란을 후일에 마련해 주시겠지! 자, 한 번 더 잘 있거라. 용감히 싸워라. 성공을 빈다!

– 윌리엄 셰익스피어, 리차드 3세

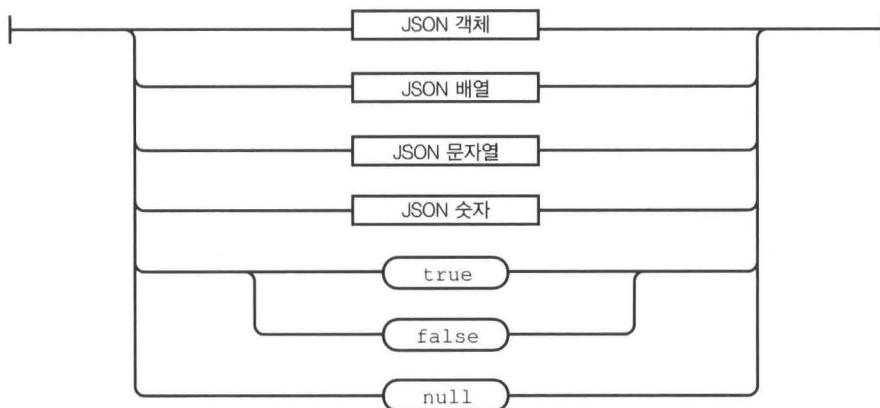


JSON(JavaScript Object Notation)은 경량화된 데이터 교환 형식입니다. 이 형식은 자바스크립트의 가장 좋은 점 중에 하나인 객체 리터럴 표기법에 기초하고 있습니다. JSON이 자바스크립트의 부분집합이기는 하지만 언어에 독립적입니다. JSON은 프로그램이 (오늘날 사용되는) 어떤 프로그래밍 언어로 작성됐던지 관계없이 서로 데이터를 교환하는데 사용할 수 있습니다. JSON은 텍스트 형식이기 때문에 사람이나 기계가 모두 읽을 수 있고 구현이나 사용이 쉽습니다. JSON과 관련된 자료들은 <http://www.JSON.org>에 있습니다.

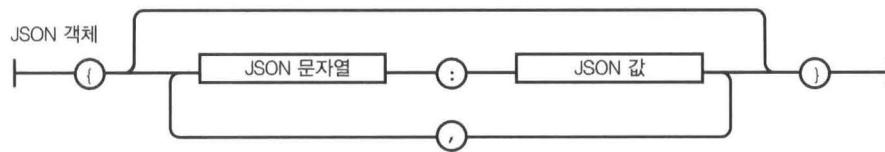
## 01 | JSON 구문

JSON은 객체, 배열, 문자열, 숫자, 불리언(true/false), null 이렇게 여섯 종류의 값이 있습니다. 공백 문자(빈 칸, 탭, CR, LF)는 값의 앞이나 뒤에 삽입할 수 있습니다. 이런 규칙 하에 공백 문자를 적절히 사용하면 JSON 텍스트를 사람이 보다 쉽게 읽을 수 있습니다. 공백 문자는 전송이나 저장 비용을 줄이기 위해 생략할 수 있습니다.

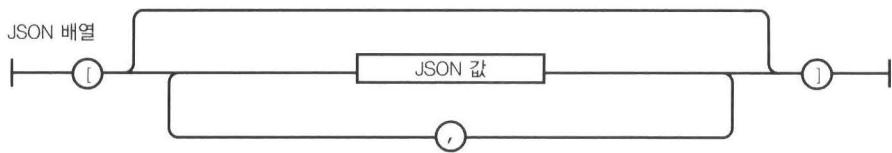
### JSON 값



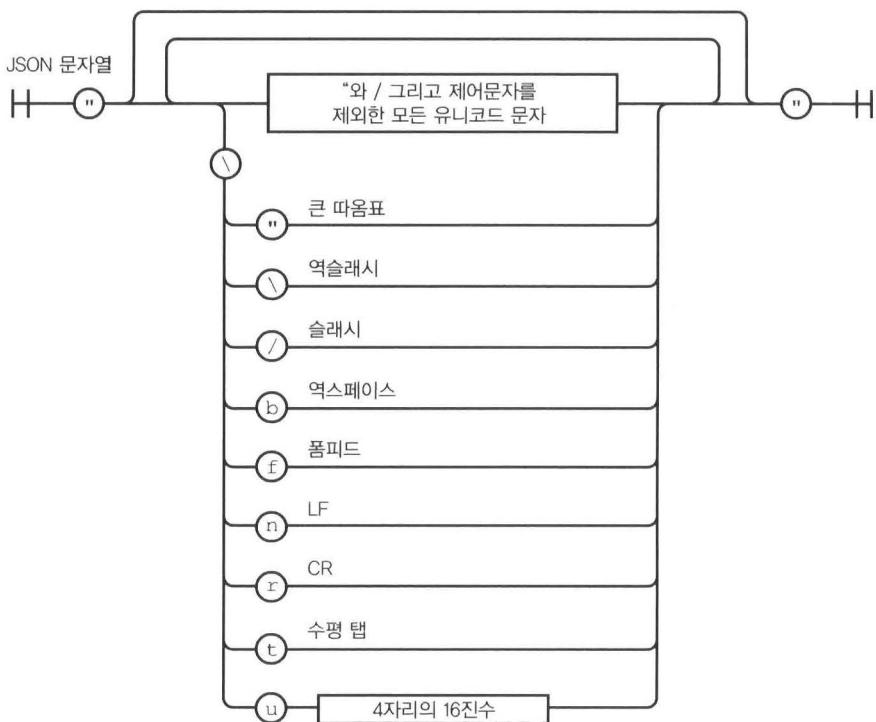
JSON 객체는 이름/값 쌍들을 순서 없이 가지고 있는 컨테이너입니다. 이름은 문자열이고 값은 배열과 객체를 포함한 모든 JSON 값이 가능합니다. JSON 객체는 원하는 만큼 중첩해서 사용할 수 있습니다. 하지만 중첩 단계를 상대적으로 얕게 유지하는 것이 일반적으로 가장 효율적입니다. 대부분의 언어는 JSON 객체를 쉽게 사상(mapping)할 수 있는 객체, 구조체, 레코드, 사전, 해시 테이블, 속성 리스트, 연관 배열 등이 있습니다.



JSON 배열은 순서를 가진 값들의 연속체입니다. 배열 요소의 값은 JSON 배열이나 객체를 포함한 모든 JSON 값이 가능합니다. 대부분의 언어는 JSON 배열을 쉽게 사상(mapping)할 수 있는 배열, 벡터, 리스트, 시퀀스 등이 있습니다.

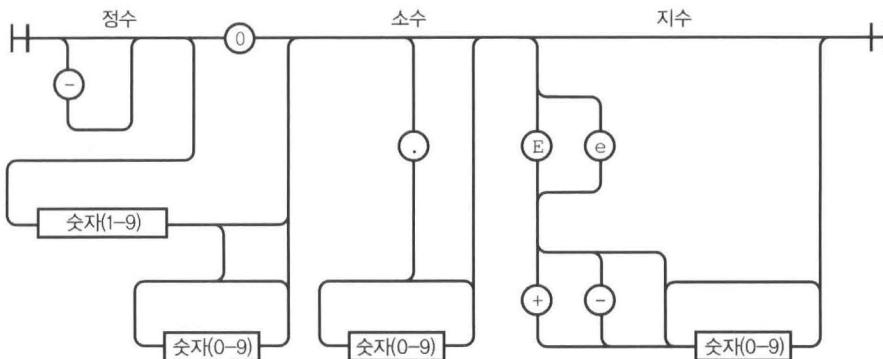


JSON 문자열은 큰 따옴표로 묶입니다. 이스케이프를 위해서는 \를 사용합니다. JSON은 /가 이스케이프되는 것을 허용하기 때문에 JSON을 HTML <script> 태그 내에 넣을 수 있습니다. <script> 태그 내에 JSON을 넣는 경우 주의할 점은, HTML이 </script>를 제외하고는 </>를 허용하지 않는다는 것입니다. 그래서 JSON에 </>가 포함된 경우 문제가 될 수 있지만, JSON은 </>를 <\>로 표기하는 것을 허용하기 때문에 HTML에 위배되는 </>를 <\>로 이스케이프하여 사용할 수 있습니다.



JSON 숫자는 자바스크립트의 숫자와 유사합니다. 정수에서 앞에 0을 붙이는 것을 허용하지 않는데 일부 언어에서는 앞에 0을 붙이는 것이 8진수를 나타내기 때문입니다. 이러한 종류의 기수(진법) 혼동은 데이터 교환 형식에는 바람직하지 않습니다. 숫자는 정수, 실수 또는 그보다 더 정밀한 값 모두 가능합니다.

JSON 숫자



여기까지가 JSON의 전부입니다. JSON을 설계할 때 목표로 삼은 것은 최소화, 이식성, 텍스트 기반, 자바스크립트의 부분집합이었습니다. 상호 운용을 위해 동의해야 할 것들이 적으면 적을수록 더 쉽게 운용될 수 있습니다.

```
[
  {
    "first": "Jerome",
    "middle": "Lester",
    "last": "Howard",
    "nick-name": "Curly",
    "born": 1903,
    "died": 1952,
    "quote": "nyuk-nyuk-nyuk!"
  },
  {
    "first": "Harry",
    "middle": "Moses",
    "last": "Howard",
    "nick-name": "Moe",
  }
]
```

```
        "born": 1897,  
        "died": 1975,  
        "quote": "Why, you!"  
    },  
    {  
        "first": "Louis",  
        "last": "Feinberg",  
        "nick-name": "Larry",  
        "born": 1902,  
        "died": 1975,  
        "quote": "I'm sorry. Moe, it was an accident!"  
    }  
]
```

## 02 | JSON을 안전하게 사용하기

JSON은 자바스크립트이기 때문에 특별히 웹 애플리케이션에서 사용하기가 쉽습니다. JSON 텍스트는 다음과 같이 eval 함수를 통해 유용한 데이터 구조로 전환될 수 있습니다.

```
var myData = eval('(' + myJSONText + ')');
```

(JSON 텍스트에 괄호를 붙이는 것은 자바스크립트 문법에서 모호함을 피하기 위해서입니다.)

eval 함수는 보안에 있어서 치명적인 문제점이 있습니다. 그렇다면 eval 함수로 JSON 텍스트를 파싱하는 것은 안전할까요? 현재 웹 브라우저에서 서버로부터 데이터를 얻는 가장 좋은 기술은 XMLHttpRequest를 사용하는 것입니다. XMLHttpRequest는 해당 HTML이 속하는 서버로부터만 데이터를 수신할 수 있습니다. 서버로부터 보내져서 eval되는 텍스트는 원래 HTML 만큼이나 안전합니다. 하지만 서버가 문제가 있는 경우를 생각해 보겠습니다. 만약, 서버가 기능을 제대로 못하는 불완전한 경우라면 어떻게 될까요?

불완전한 서버는 JSON 형식을 제대로 만들지 못할 수 있습니다. 서버가 적절한 JSON 인코더를 사용하여 만들지 않고 아무렇게나 문자열을 합쳐 JSON 텍스트를 만든다면 의도하지는 않았지만 결과적으로 위험한 물건을 클라이언트로 전송하게 될 수 있습니다. 또한 서버가 프록시 서버 역할을 하여 JSON 형식이 제대로 갖춰졌는지 확인하지 않고 받은 것을 그대로 전송한다면 이 역시도 위험 요소를 보내는 것이 될 수 있습니다.

이러한 위험은 eval 대신 JSON.parse 메소드를 사용하여 피할 수 있습니다 (<http://www.JSON.org/json2.js> 참조). JSON.parse는 텍스트 안에 위험해 보이는 것이 있으면 예외를 발생합니다. 불완전한 서버에 대응하기 위해서 항상 eval 대신 JSON.parse를 사용할 것을 권해드립니다. 이러한 방법은 브라우저가 다른 서버에서 데이터를 직접 받는 것을 허용하는 때가 되면 안전을 위해 더욱 유용합니다.

외부 데이터와 innerHTML 사이에 상호작용을 할 때 또 다른 위험이 있습니다. 일반적인 Ajax 패턴은 서버에서 HTML 텍스트를 보내고 이를 HTML 요소의 innerHTML 속성에 할당하는 것입니다. 이러한 방법은 아주 나쁜 관행입니다. HTML 텍스트에 <script> 태그나 그에 상응하는 부분이 속해 있는 경우 악의적인 스크립트가 실행될 수 있습니다. 이 역시도 물론 불완전한 서버로 인한 문제입니다.

특별히 어떤 점이 위험할까요? 악의적인 스크립트가 자신의 페이지에서 실행되면, 페이지의 모든 상태와 기능에 접근할 수 있습니다. 이 스크립트는 서버와 상호작용을 할 수도 있는데 서버는 요청이 제대로만 오면 이 요청이 악의적인 스크립트가 한 요청인지 그렇지 않은지를 구별할 수 없습니다. 악의적인 스크립트는 전역 객체에도 접근할 수 있는데 이렇게 되면 클로저(closure)로 숨겨진 변수를 제외한 애플리케이션의 모든 변수를 접근할 수 있게 됩니다. 악의적인 스크립트는 document 객체에도 접근할 수 있습니다. document 객체에 접근할 수 있다는 의미는 사용자가 보는 모든 것에 접근할 수 있다는 의미입니다. document 객체에 접근할 수 있으면 사용자에게 원하는 내용을 (경고창 등을 통해) 보여줄 수도 있는데 이러한 내용은 주소가 이동돼서 보이는 것이 아니기 때문에 피싱 필터에도 감지될 수 없고 결국 사용자는 이 내용을 바른 내용으로 믿게 될 수 있습니다. 또한

document 객체는 네트워크를 사용할 수 있게도 해서 보다 악의적인 스크립트를 로딩하거나 방화벽 내의 사이트들을 검색하고 이렇게 알게 된 정보들을 외부로 보낼 수 있습니다.

이러한 위험은 모두 자바스크립트의 전역객체로 인한 직접적인 결과인데 자바스크립트의 많은 나쁜 점들 중에서도 단연 가장 나쁜 점입니다. 이러한 위험은 Ajax나 JSON 또는 XMLHttpRequest나 Web 2.0(혹은 뭐라 지칭하든) 때문이 아닙니다. 이러한 위험은 자바스크립트가 소개된 이후로 계속해서 브라우저에 내재돼 있었고 자바스크립트가 대체되기 전까지는 계속 있게 될 위험입니다. 그러므로 주의해야 합니다.

## 03 | JSON 파서

다음은 자바스크립트로 구현한 JSON 파서입니다.

```
var json_parse = function () {  
    // JSON 텍스트를 파싱하여 자바스크립트 데이터 구조를 반환하는 함수.  
    // 재귀적 하향 방식의 파서.  
  
    // 전역변수 생성을 피하기 위해서 함수 내에 또 다른 함수를 정의할 것임.  
  
    var at,      // 현재 문자의 인덱스  
        ch,      // 현재 문자  
        escapee = {  
            "'': "'",  
            '\\': '\\\\',  
            '/': '/',  
            b: 'b',  
            f: '\f',  
            n: '\n',  
            r: '\r',  
            t: '\t'  
        }  
};
```

```

},
text,

error = function (m) {

// 잘못된 것이 있을 때는 error 호출.
throw {
    name:      'SyntaxError',
    message:   m,
    at:        at,
    text:      text
};
}

next = function (c) {

// 매개변수가 주어지면, 현재 문자와 일치하는지 검사.

if (c && c !== ch) {
    error("Expected '" + c + "' instead of '" + ch + "'");
}
}

// 다음 문자를 취함. 더 이상 문자가 없는 경우 빈 문자열 반환.

ch = text.charAt(at);
at += 1;
return ch;
}

number = function () {

// 숫자값 파싱.

var number,
    string = '';

if (ch === '-') {
    string = '-';
    next('-');
}
}

```

```

        }
        while (ch >= '0' && ch <= '9') {
            string += ch;
            next();
        }
        if (ch === '.') {
            string += '.';
            while (next() && ch >= '0' && ch <= '9') {
                string += ch;
            }
        }
        if (ch === 'e' || ch === 'E') {
            string += ch;
            next();
            if (ch === '-' || ch === '+') {
                string += ch;
                next();
            }
            while (ch >= '0' && ch <= '9') {
                string += ch;
                next();
            }
        }
        number = +string;
        if (isNaN(number)) {
            error("Bad number");
        } else {
            return number;
        }
    },
    string = function () {
        // 문자열 파싱.
        var hex,
            i,
            string = '',
            uffff;

```

```

// 문자열을 파싱할 때, "와 \"를 찾아야 함.
if (ch === '') {
    while (next()) {
        if (ch === '') {
            next();
            return string;
        } else if (ch === '\\\\') {
            next();
            if (ch === 'u') {
                uffff = 0;
                for (i = 0; i < 4; i += 1) {
                    hex = parseInt(next(), 16);
                    if (!isFinite(hex)) {
                        break;
                    }
                    uffff = uffff * 16 + hex;
                }
                string += String.fromCharCode(uffff);
            } else if (typeof escapee[ch] === 'string') {
                string += escapee[ch];
            } else {
                break;
            }
        } else {
            string += ch;
        }
    }
    error("Bad string");
},
white = function () {
    // 공백 문자를 건너뜀.
    while (ch && ch <= ' ') {
        next();
    }
},

```

```

word = function () {
    // true나 false나 null

    switch (ch) {
        case 't':
            next('t');
            next('r');
            next('u');
            next('e');
            return true;
        case 'f':
            next('f');
            next('a');
            next('l');
            next('s');
            next('e');
            return false;
        case 'n':
            next('n');
            next('u');
            next('l');
            next('l');
            return null;
    }
    error("Unexpected '" + ch + "'");
},
value, // value 함수를 위한 변수

array = function () {
    // 배열 파싱.

    var array = [];

    if (ch === '[') {
        next('[');

```

```

white();
if (ch === ']') {
    next(']');
    return array; // 빈 배열
}
while (ch) {
    array.push(value());
    white();
    if (ch === ']') {
        next(']');
        return array;
    }
    next(',');
    white();
}
error("Bad array");
},
object = function () {
// 객체 파싱.

var key,
object = {};

if (ch === '{') {
    next('{');
    white();
    if (ch === '}') {
        next('}');
        return object; // 빈 객체
    }
    while (ch) {
        key = string();
        white();
        next(':');
        object[key] = value();
        white();
    }
}

```

```

        if (ch === '}') {
            next('}');
            return object;
        }
        next(',');
        white();
    }
}

error("Bad object");
};

value = function () {

// JSON 값 파싱. 유효한 값은 객체, 배열, 문자열, 숫자, 단어.

white();
switch (ch) {
case '{':
    return object();
case '[':
    return array();
case '"':
    return string();
case '-':
    return number();
default:
    return ch >= '0' && ch <= '9' ? number() : word();
}
};

// json_parse 함수를 반환. 위에 정의한 모든 함수와 변수에 접근 가능.

return function (source, reviver) {
    var result;

    text = source;
    at = 0;
    ch = ' ';
    result = value();
}

```

```

white();
if (ch) {
    error("Syntax error");
}

// reviver 함수가 있으면, 파싱한 결과를 재귀적으로 탐색하여
// 각각의 이름/값 쌍을 reviver 함수로 보내 변환하게 하는
// walk 함수를 호출하여 반환. walk 함수를 호출할 때 처음 값으로 빈 키에 값은
// 반환값(result)을 가진 임시 객체를 넘김. reviver 함수가 없으면 단순히 result를 반환.

return typeof reviver === 'function' ?
    function walk(holder, key) {
        var k, v, value = holder[key];
        if (value && typeof value === 'object') {
            for (k in value) {
                if (Object.hasOwnProperty.call(value, k)) {
                    v = walk(value, k);
                    if (v !== undefined) {
                        value[k] = v;
                    } else {
                        delete value[k];
                    }
                }
            }
        }
        return reviver.call(holder, key, value);
    }({': result'}, '') : result;

};

}();

```



찾아보기

**기호**

!= 연산자 181  
 !== 연산자 181  
 & 188  
 && 34  
 + 34,173  
 ++ 187  
 += 32  
 << 188  
 = 32  
 == 177,181  
 >> 188  
 >>> 188  
 ? 33  
 [] 100  
 ^ 188  
 | 188  
 || 42  
 ~ 188  
 / 34  
 /\* \*/ 21

**가**

계승 64, 80  
 객체 리터럴 36, 100  
 객체 기술자 87  
 객체  
   || 연산자 42  
   객체 생성 40  
   위임 44  
   delete 연산자 47  
   열거 46  
   for in 문 46  
   함수 49  
   전역변수 47  
   hasOwnProperty 메소드 45  
   리터럴 40  
   속성 41  
   프로토타입 체인상의 속성 44  
   prototype 22  
   프로토타입 43  
     연결 44  
   참조 43  
   리플렉션 45  
   속성 값 읽기 41

undefined 41, 44

값 갱신 42  
 구문 다이어그램 211-222  
 꼬리 재귀 최적화 64  
 공백 20

**다**

동등 연산자 181

**라**

라벨 205  
 리플렉션 45  
 랙퍼 190

**마**

문자 타입 25  
 문법 19-38  
   표현식 33  
   함수 37  
   리터럴 36  
   이름 21  
   숫자 23  
   메소드 24  
   음수 23  
 객체 리터럴 36  
 다이어그램 해석 방법 19  
 문장 26  
 문자열 24  
   length 속성 25  
   공백 20  
   목시적 전역변수 168  
 메모이제이션 78  
 메소드 호출 패턴 52  
 메소드  
   array.concat( ) 131  
   array.join( ) 132  
   array.pop( ) 132  
   array.push( ) 133  
   array.reverse( ) 133  
   array.shift( ) 134  
   array.slice( ) 134  
   array.sort( ) 135-138  
   array.splice( ) 139-140  
   array.unshift( ) 141  
   function.apply( ) 141

찾아보기

237

number.toExponential( ) 142  
number.toFixed( ) 143  
number.toPrecision( ) 143  
number.toString( ) 144  
object.hasOwnProperty( ) 144  
regexp.exec( ) 145  
regexp.test( ) 147  
string.charAt( ) 148  
string.charCodeAt( ) 149  
string.concat( ) 149  
String.fromCharCode( ) 156  
string.indexOf( ) 149  
string.lastIndexOf( ) 149  
string.localeCompare( ) 150  
string.match( ) 150  
string.replace( ) 151  
string.search( ) 153  
string.slice( ) 153  
string.split( ) 154  
string.substring( ) 155  
string.toLocaleLowerCase( ) 155  
string.toLocaleUpperCase( ) 155  
string.toLowerCase( ) 156  
string.toUpperCase( ) 156  
모듈 72-74  
문장  
    블록 27  
    break 29, 31  
    case 절 29  
    catch 절 31  
    do 27, 30  
    for 27, 29  
    for in 30  
    if 28  
    라벨 31  
    return 31  
    switch 27, 28, 31  
    then 블록 28  
    throw 31  
    try 31  
    var 26  
    while 27, 29  
문자열 24, 39  
    빈 문자열 28  
    length 속성 25

## 바

    배열 리터럴 37  
    배열  
        새로운 항목 추가 102  
        배열의 크기와 차원 108  
        객체와 배열의 혼동 104  
        delete 연산자 103

    다차원 배열 108  
    열거 104  
    length 속성 101  
    리터럴 100  
    메소드 106  
        Object.create 메소드 107  
        splice 메소드 103  
        typeof 연산자 105  
        undefined 값 108  
    비트 연산자 187, 206  
    블록 주석 21, 160  
    블록 유효범위 66, 165  
    블록 없는 문장 186  
    블록 28, 200  
    불리언 39  
    빈 문자열 28  
    부동 소수점 174  
    반복문 29, 30

## 사

    생성자 55  
        정의 83  
    생성자 함수  
        위험 86  
        new 전치 연산자 86  
    생성자 호출 패턴 54  
    실행문 26  
    상속  
        함수를 사용한 방식 90-95  
        캡쳐 기술자 87  
        부속품 96-98  
        프로토타입 방식 88  
        의사 클래스 방식 82-86, 95  
    숫자 23, 39  
    숫자 객체 101

## 아

    아름다운 속성에 대한 단상 163-166  
    연속 호출 75  
    위임(delegation) 44  
    열거 46  
    이스케이프 문자 24  
    이스케이프 시퀀스 25  
    예외 58  
    이름 21  
    연산자 우선순위 34  
    의사 클래스 방식 82, 86, 95  
    예약어 22, 170  
    유효범위 28, 65, 169  
    세미콜론 169, 198  
    스타일 157-162  
    유니코드 171

**자**

증감 연산자 187, 199, 205  
 주석 21, 161  
 자바스크립트  
     분석 14-17  
     표준 16  
     사용해야하는 이유 13  
 채귀호출  
     DOM 63  
     피보나치 수열 78  
     꼬리 재귀 최적화 64  
     하노이의 탑 62  
 정규 표현식 111-129

**차**

철도 다이어그램 19

**카**

콜백 70  
 캐스팅 81  
 클로저 66-70

**타**

테스트 환경 17

**파**

표현식 33-36  
     ? 삼항 연산자 33  
     이항 연산자 35  
     호출 35  
     연산자 우선순위 34  
     세부지정 36  
     변수 33  
     피보나치 수열 78  
     프로토타입 방식 88

**하**

함수 호출 패턴 53  
 함수 객체 47  
 함수 문장 vs. 함수 표현식 188  
 함수를 사용한 방식(상속) 90-95  
 함수  
     인수 배열 57  
     기본 타입에 기능 추가 59  
     콜백 71  
     연속 호출 75  
     클로저 66-70  
     커링 76  
     예외 58  
     모듈 72  
     호출 51-56  
         apply 호출 패턴 56  
         생성자 호출 패턴 54

함수 호출 패턴 53  
 메소드 호출 패턴 52  
 new 전치 연산자 54  
 호출 연산자 52  
 생성자로 호출 54  
 리터럴 50  
 메모이제이션 78  
 모듈 72-75  
 객체 49  
 채귀호출 62-64  
     DOM 63  
     피보나치 수열 78  
     꼬리 재귀 64  
     하노이의 탑 62  
 return 문 58  
 유효범위 65  
 호출 연산자 34, 52  
 하노이 탑 61

**A**

apply 호출 패턴 56  
 arguments 57  
 array.concat() 131  
 array.join() 132  
 array.pop() 132  
 array.push() 133  
 array.reverse() 133  
 array.shift() 134  
 array.slice() 134  
 array.sort() 135-138  
 array.splice() 139-140  
 array.unshift() 141

**B**

break 문 29, 31

**C**

case 절 29  
 catch 절 31  
 constructor 속성 82  
 continue 문 185  
 curry 메소드 77

**D**

deentityify 메소드 72  
 delete 연산자 103  
 do 문 27, 30  
 DOM 63

**E**

ECMAScript 언어 명세 194  
 eval 함수 184  
     보안 문제점 227

**F**

for in 문 30  
 객체 46  
 for 문 27, 29  
 function.apply( ) 메소드 141

**H**

hasOwnProperty 메소드 45, 178, 179  
 HTML  
 <script> 태그 (JSON) 225  
 innerHTML 속성 228  
 JSLint 193

**I**

if 문 28  
 Infinity 24, 33  
 inherits 메소드 85  
 innerHTML 속성 228

**J**

JSON 15, 223, 236  
 / 문자 225  
 배열 225  
 eval 함수 227  
 HTML <script> 태그 226  
 innerHTML 속성 228  
 JSLint 193  
 숫자 226  
 객체 224  
 문자열 225  
 예제 226  
 안전하게 사용 227  
 JSON.parse 메소드 228  
 JSLint 16, 193-209

**K**

K&R 스타일 160

**L**

length 속성 (배열) 101

**M**

Math 객체 24  
 method 메소드 85

**N**

NaN 22, 24, 28, 33, 174  
 new 연산자  
     사용을 잊은 경우 86  
     함수 54  
 null 28, 33, 177

**O**

Object.create 메소드 90, 107

Object.hasOwnProperty( ) 메소드 144  
 Object.prototype 106

**P**

parseInt 함수 173  
 private 메소드 91  
 prototype 속성 82

**R**

RegExp 객체 145  
 regexp.exec( ) method 145  
 regexp.test( ) 메소드 147  
 return 문 31, 58

**S**

setInterval 함수 184  
 setTimeout 함수 184  
 splice 메소드 103  
 string.charCodeAt( ) 148  
 string.charCodeAtAt( ) 149  
 string.concat( ) 149  
 String.fromCharCode( ) 156  
 string.indexOf( ) 149  
 string.lastIndexOf( ) 149  
 string.localeCompare( ) 150  
 string.match( ) 150  
 string.replace( ) 151  
 string.search( ) 153  
 string.slice( ) 153  
 string.split( ) 154  
 string.substring( ) 155  
 string.toLocaleLowerCase( ) 155  
 string.toLocaleUpperCase( ) 155  
 string.toLowerCase( ) 156  
 string.toUpperCase( ) 156

**T**

then 블록 28  
 throw 문 31  
 trim 메소드 60  
 try 문 31  
 TypeError 예외 42  
 typeof 연산자 34, 104, 171, 176

**U**

undefined 41, 44, 108

**V**

var 26  
 void 연산자 191

**W**

while 문 27, 29  
 with 문 182

# 꼭 알아야 하는 자바스크립트의 좋은 점 그리고 나쁜 점



대부분의 프로그래밍 언어가 좋은 점과 나쁜 점이 모두 있지만 자바스크립트는 언어가 잘 정제되기 전에 급하게 출시된 태생적 한계로 나쁜 점이 더 많습니다. 이 책은 이러한 나쁜 점을 모두 제거하고, 신뢰할 수 있으면서도, 읽기 편하고 유지보수가 편한 요소만을 모아 소개합니다. 이러한 요소만을 사용하면 여러분은 확장이 쉽고 효율적인 코드를 작성할 수 있습니다.

개발 커뮤니티에서 많은 사람에게 자바스크립트 전문가로 인정받고 있는 이 책의 저자 더글라스 크락포드는 자바스크립트를 우수한 객체지향 언어로 만들 수 있는 수많은 장점을 찾아냈습니다. 하지만 애석하게도 이러한 장점들(함수, 느슨한 변수 타입 검사, 동적 객체, 객체 리터럴 표현식)은 자바스크립트의 대표적인 단점들(예를 들어 전역변수에 근거한 프로그래밍 모델)과 섞여 사용됨으로써 그 진가가 드러나지 않고 있습니다.

이 책에서 크락포드는 다음과 같은 부분들을 설명하면서 언어의 장단점을 심도 있게 다루고 있습니다.

- 구문
- 객체
- 함수
- 상속
- 배열
- 정규식
- 메소드
- 코딩 스타일
- 훌륭한 기능

이 책으로 여러분은 효율적인 코드를 작성하는 훌륭하고, 우아하며, 가볍고, 상당히 표현적인 언어를 발견할 수 있을 것입니다. 여러분이 Ajax 개발이든, 라이브러리 관리든 어떠한 작업을 하든지 상관 없이 이 책을 적극 권장할 수 있지만 특히 웹에서 사이트나 애플리케이션을 개발하고 있다면 이 책은 필독서라고 할 수 있습니다.

웹프로그래밍 | 자바스크립트



9 788979 145984

ISBN 978-89-7914-598-4

정가 22,000원