Hanbit RealTime 101

You Don't Know JS

타입과 문법

카일 심슨 지음 / 이일웅 옮김

O'REILLY® 한빛미디어 Hanbit Media, Inc.

You Don't Know JS

타입과 문법

카일 심슨 지음 / **이일웅** 옮김

한빛미디어 Hanbit Media, Inc.

You Don't Know JS **타입과 문법**

초판발행 2015년 5월 26일

지은이 카일 심슨 / **옮긴이** 이일웅 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-750-7 15000 / **정가** 15,000원

총괄 배용석 / **책임편집** 김창수 / **기획·편집** 김상민
디자인 표지/내지 여동일, 조판 최송실
마케팅 박상용 / **영업** 김형진, 김진불, 조유미

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

누군가 그랬다. 자바스크립트는 개발자들이 실제로 사용하기 전까진 배우려고 하지 않는 유일한 프로그래밍 언어라고.

이 말을 떠올릴 때마다 나 자신도 같은 생각이었기에 웃음을 참기가 어렵다. 나뿐 아니라 다른 개발자들도 사정은 비슷하리라 본다. 자바스크립트, HTML/CSS는 인터넷 태동기 시절 대학에서 가르치는 필수 컴퓨터 과학 언어가 아니어서, 혼자서 본인의 검색 능력과 브라우저 "소스 보기" 기능에 의지한 채 여기저기서 취합한 코드 조각들을 가지고 개발하는 정도가 고작이었다.

고등학생 시절, 처음으로 웹 프로젝트를 했던 일이 기억난다. 어떤 형태로든 웹 스토어를 구축하는 과제였는데, 제임스 본드 광팬이었던 난 주저 없이 골든아이 스토어를 만들기로 결심했다. 배경 음악으로 흘러나오는 골든아이 MIDI 주제가부터 자바스크립트로 구현한 십자형 마우스 포인터, 클릭할 때마다 귀청을 울려대는 총성까지 정말 없는 게 없었다. 아마 Q(제임스 본드의 장비를 담당)도 내 작품을 봤다면 대견스러워 했을 것이다.

뜬금없이 옛날 얘기를 꺼낸 이유는 오늘날 많은 개발자가 당시 내가 저질렀던 행동, 즉 여기저기 인터넷에서 떠돌던 자바스크립트 코드 뭉치를 대충 긁어다가 뭘 하는 스크립트인지 구체적으로 따지지도 않고 프로젝트에 복사해 넣는 짓을 여전히 하고 있기 때문이다. 제이쿼리JQuery 같은 툴킷이 널리 보급되어 자바스크립트를 간편하게 쓸 수 있게 된 이후로는 더욱 개발자들이 제대로 공부를 하지 않으려는 것 같다.

자바스크립트 툴킷을 폄하하려는 건 아니다(나 역시 지금은 모툴스MooTools 자바스크립트 팀에서 일하고 있다). 하지만 여러분이 알고 있는 툴킷은 그 툴킷을 만든 개발자의 기본기가 탄탄하고 자바스크립트의 함정들은 무엇인지 정확히 알고 있는 상태에서 세심하게 로직을 구사했기에 강력해진 것이다. 그래서 유용한 툴킷을 잘 쓰는 것만큼 카일 톰슨

의 “You Don’t Know JS” 시리즈 같은 책을 읽고 기본 지식을 확고히 갖추는 일이 중요하다. 다른 변명은 통하지 않는다.

시리즈 3번째 도서인 〈타입과 문법〉은 카피 앤 페이스트와 툴킷으로는 절대 배울 수 없는 자바스크립트의 핵심을 훌륭히 간파한 책이다. 강제변환과 관련된 유의 사항, 생성자로서의 네이티브, 그 밖의 자바스크립트에 대한 전반적인 기본 지식을 예제 코드와 함께 빠짐없이 설명했다. 저자 카일 톰슨은 쓸데없이 과장하는 어투를 배제하고 청산유수와 같은 문체로 핵심 포인트를 도출해낸다. 정말 내가 사랑할 수밖에 없는, 바로 그런 종류의 책이다.

이 책을 재미있게 읽고, 여러분의 책상, 손 닿는 곳 가까이 두기 바란다!

– 데이빗 왈쉬David Walsh (http://davidwalsh.name)
모질라Mozilla 수석 웹 개발자

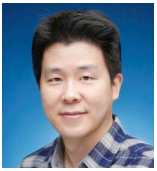
저자 소개

지은이_ **카일 심슨**

텍사스 오스틴 출신의 카일 심슨Kyle Simson은 오픈 웹 전도사로 자바스크립트, HTML5, 실시간 P2P 통신과 웹 성능에 누구 못지않은 열정을 갖고 있다. 안 그랬으면 이미 오래전에 질려 버렸을 것이다. 저술가, 워크숍 강사, 기술 연사이며 오픈 소스 커뮤니티에서도 열심히 활동하고 있다.

역자 소개

옮긴이_ **이일웅**



10년 넘게 국내, 미국 등지에서 대기업/공공기관 프로젝트를 수행한 웹 개발자이자 두 딸의 사랑을 한몸에 받고 사는 행복한 딸바보다. 자바 기반의 서버 플랫폼 구축, 데이터 연계, 그리고 다양한 자바스크립트 프레임워크를 응용한 프론트엔드 화면 개발을 주로 담당해 왔다. 시간이 날 때엔 피아노를 연주한다.

- 개인 홈페이지: http://www.bullion.pe.kr

제가 처음 자바스크립트 언어를 사용하기 시작한 건 1997년 즈음인데(물론 순전히 재미삼아 HTML 웹페이지를 만들던 시절이었습니다), 그때 서점에서 구입한 책이 아직도 책장에 꽂혀 있어 이 책의 번역을 의뢰받고 오랜만에 꺼내 들춰보고는 웃음을 참기가 어려웠습니다. 데이빗 왈쉬 역시 이 책의 추천사에서 비슷한 감회를 털어놓았는데요. '좀 덜 떨어져 보이는' 자바스크립트 언어가 이제는 세상에서 가장 인기 있는 프로그래밍 언어로 확고히 자리를 잡고 브라우저 영역을 벗어나 다양한 무대에서 주류 언어로 활용되는 모습을 보면 참으로 '상전벽해'의 느낌입니다.

그러나 사람들이 많이 사용하기 때문에 그만큼 다양한 방식으로 진화를 거듭해오면서 초기 언어 설계자가 미처 예상치 못했던 부분들이 대단한 오류처럼 인식되어 사실 아직까지도 욕을 많이 얻어먹고 있습니다만, 이 책의 저자 카일 심슨의 기본 사상도 그렇듯이 인류가 만든 도구에 절대적인 선악의 기준을 들이대는 것처럼 덧없는 일은 또 없는 것 같습니다. 특정한 문명의 이기를 어떻게 활용하여 어떠한 결과를 만들어낼 수 있을지는 사용하는 사람의 지혜와 경험, 폭넓은 지식에 따라 달라지는 것 아닐까요?

하지만 여기서 독자 여러분이나 저 같은 프로그래밍을 업으로 삼고 있는 사람들이 명심해야 할 점은 최소한의 원칙과 규정(즉, 자바스크립트 언어라면 ECMAScript 명세)은 알고 있는 상태라야 본인이 처한 상황에 맞게 올바른 방향으로 실전적인 코드를 작성할 수 있다는 겁니다. 명세라는 것이 있는지조차 모르는 개발자가 짠 코드라면 당장은 화면에서 멋진 모습을 고객에게 시연할 수는 있어도 곧 누더기가 되어 아무도 유지 보수할 수 없는 통제 불능의 골칫덩이가 될 것입니다(개발 경험이 풍부한 독자 여러분들은 다른 사람이 이렇게 짜놓고 가버린 코드 때문에 고생한 기억이 새록새록 떠오를 겁니다).

이 책은 기존 자바스크립트 도서와는 완전히 다른 시각에서 자바스크립트 언어의 가장 난해하면서도 논란이 되어 온 주제인 '타입과 문법'에 관한 이야기를 저자가 코앞에

서 들려주듯이 술술 풀어갑니다. 적당히 긴장하고 읽지 않으면 논점을 놓치게 될 수도 있지만, 개발자들이 그간 서로 쉬쉬하며 피해왔던 자바스크립트의 속내를 하도 철저하게 후벼 파는 터라, 몰입도 100% 액션 영화를 감상하는 듯한 짜릿함도 있습니다. 부디 저자의 바람처럼 이 작은 책 한 권이 그동안 잘 몰라 또는 알고 싶지 않아 그냥 지나쳤던, 자바스크립트의 비밀들을 이해하고 깨우치는 훌륭한 계기가 되었으면 좋겠습니다.

제가 저술한 책은 아니지만 번역은 사실 또 다른 창작이란 생각이 듭니다. 부족한 실력이나마 번역을 의뢰하신 한빛미디어 김창수 팀장님과 스마트미디어팀 여러분, 그리고 주말과 휴일 내내 많은 시간 함께 해주지 못한, 사랑하는 제 아내와 두 딸 제이와 솔이에게 이 역서를 바칩니다. 그리고 언제나 아들에게 변함없는 믿음과 사랑을 보내주신 부모님께 진심으로 감사의 말씀 올립니다.

2015년 4월, 어느 빗발이 흩날리는 밤에

이일웅

1. 국어 표준 맞춤법, 외래어 표기법 및 띄어쓰기 규정을 준수한다.
2. 기술 용어, 제품명 등 고유 명사 형태의 원어는 최대한 한글로 음차하여 옮긴다(예: JavaScript → 자바스크립트). 약자 형태로 축약된 원어나 그밖에 한글 음차로 옮기는 것보다 원어 그대로 표기하는 편이 독자의 이해에 도움이 된다면 원어를 표기한다(예: PK).
3. 2번에서 한글 음차 시 최초 1회 영문을 윗첨자 형태로 병기하고, 이후 반복하여 등장할 경우 이를 생략한다. 그러나 이렇게 병기한 용어가 다시 나올 경우에도 독자의 이해를 위해 필요할 경우 다시 영문 병기를 한다.
4. 'You don't know JS' 시리즈 도서는 대체로 일상생활에서 대화를 나누는 듯한 구어적인 표현이 많은데, 이러한 특성을 한글 번역본에도 최대한 반영하려고 노력하였다.
5. 이미 업계나 기술자들 사이에서 많이 사용되어 외래어처럼 굳어진 용어는, 어차피 이 도서의 대상 독자가 일반인이 아니기 때문에, 굳이 우리말로 번역하지 않고 원어를 그대로 음차한다(예: type → 형 타입, copy and paste → 복사 후 붙여넣기 카피 앤 페이스트). 그리고 저자가 즐겨 쓰는 일부 비표준 용어(예: coercion)는 가장 가까운 한글 용어로 번역하여 그 의미를 최대한 반영하고(예: coercion → 강제변환), 독자가 혼동할 우려가 있는 경우 각주에서 그 이유를 밝힌다.
6. 저자가 기술한 원문이 직역 시 이해가 어렵다고 판단하면 문장 구조를 재배열하거나 관련 문구를 추가하는 식으로 의역을 병행한다. 필요하다면 번안 수준의 번역을 일부 적용한다.
7. 예제 코드의 주석 및 기술적인 내용과는 무관한 상수 문자열은 한글 번역을 하되, 프로그램 로직을 파악하는 데 오히려 방해가 되거나 코드 가독성을 떨어뜨릴 수 있는 경우 원래 코드를 유지한다.

이미 눈치챘겠지만, 본 시리즈 제목 일부인 "JS"는 자바스크립트를 폄하할 의도로 쓴 약어가 아니다. 물론 자바스크립트 언어에 숨겨진 기벽quirk이 만인이 비난하는 대상임은 부인할 수 없겠지만!

웹 초장기 시절부터 자바스크립트는 사람들이 대화하듯 웹 콘텐츠를 소비할 수 있게 해준 기반 기술이었다. 마우스 트레일을 깜빡이거나 팝업 알림창을 띄워야 할 수요에서 비롯되어 20년 가까이 흐른 지금, 자바스크립트는 엄청난 규모로 기술적 역량이 성장하였고, 세계에서 가장 널리 사용되는 소프트웨어 플랫폼이라 불리는, 웹의 심장부를 형성하는 핵심 기술이 되었다.

그러나 프로그래밍 언어로서의 자바스크립트는 끊임없는 비난과 논란의 대상이기도 했는데, 부분적으론 과거로부터 전해 내려온 폐해 탓이기도 하지만, 그보다 설계 철학 자체가 문제 시 되기도 했다. 브렌단 아이크Brendan Eich[01]의 표현을 빌자면 '자바스크립트'란 이름 자체가 좀 더 성숙하고 나이 많은 형인 '자바' 아래의 "바보 같은 꼬마 동생dumb kid brother" 같은 느낌을 준다. 하지만 이름은 정치와 마케팅 사정상 우연히 그렇게 붙여진 것일 뿐, 두 언어는 여러 중요한 부분에서 이질적이다. "자바스크립트"와 "자바"는 "카메라"와 "카car" 만큼이나 무관하다.

C 스타일의 절차 언어에서 미묘하며 불확실한 스킴Scheme/리스프Lisp 스타일의 함수형 언어에 이르기까지 자바스크립트는 서너 개 언어로부터 근본 개념과 구문 체계를 빌려왔기 때문에 꽤 폭넓은 개발자층을 확보하는 데 대단히 유리했고, 심지어 프로그래밍 경력이 별로 없는 사람들도 쉽게 배울 수 있었다. "Hello World"를 자바스크립트로 출력하는 코드는 너무 단순해서 출시 당시엔 나름의 매력이 있었고 금방 익숙해졌다.

01 역자주_ 자바스크립트의 창시자. 1995년 넷스케이프 근무 당시 열흘 만에 자바스크립트 언어를 고안했습니다.

자바스크립트는 처음 시작하고 실행하기는 가장 쉬운 언어지만 독특한 기벽 탓에 다른 언어들에 비해 언어 자체를 완전히 익히고 섭렵한 달인은 찾아보기 매우 어려운 편이다. C/C++ 등으로 전체 규모full-scale의 프로그램을 작성하려면 언어 자체를 깊이 있게 알고 있어야 하지만, 자바스크립트는 언어 전체의 능력 중 일부를 대략 수박 겉핥기 정도만 알고 사용해도 웬만큼 서비스를 운영할 수 있다.

언어 깊숙이 뿌리를 내려 자리 잡은 정교하고 복잡한 개념이 외려 (콜백 함수를 다른 함수에 인자로 넘기는 것처럼) 겉보기에 단순한 방식으로 사용해도 괜찮게끔 유도하고, 그러다 보니 자바스크립트 개발자는 내부에서 무슨 일들이 벌어지든, 있는 그대로의 언어 자체를 사용하여 개발할 수 있다.

그러나 간단하고 쓰기 쉬운 언어일수록 여러 가지 의미와 복잡하고 세밀한, 다양한 기법들이 결집되어 있기 때문에 꼼꼼하게 학습하지 않으면 제아무리 노련한 개발자라 할지라도 올바르게 이해하지 못한다.

이것이 바로 자바스크립트의 역설이자 아킬레스건이며, 이 책을 읽고 여러분이 넘어야 할 산이다. 다 알지 못해도 사용하는 데 문제가 없다 보니 끝내 자바스크립트를 제대로 이해하지 못하고 넘어가는 경우가 비일비재하다.

목표

자바스크립트의 놀랍거나 불만스런 점들을 마주할 때마다 자신의 블랙리스트에 추가하여 금기시한다면(이런 일에 익숙한 사람들이 더러 있다), 자바스크립트란 풍성함의 빈 껍데기에 머무르게 될 것이다.

누군가 "The Good Parts"이란 유명한 별칭을 달아놓았는데[02], 부디 독자 여러분들! "좋은 부분"이라기보단 차라리 "쉬운 부분", "안전한 부분", 또는 "불완전한 부분"이라고 하는 편이 더 정확할 것이다.

"You Don't Know JS" 시리즈는 정반대의 방향으로 접근한다. 자바스크립트의 모든 것, 그 중 특히 "어려운 부분The Tough Part"을 심층적으로 이해하고 학습할 것이다!

나는 자바스크립트 개발자들이 정확히 언어가 어떻게, 왜 그렇게 작동하는지 알려 하지 않고, "그냥 이 정도면 됐지" 식으로 대충 이해하고 때우려는 자세를 직접 거론할 것이다. 험한 길을 마주한 상황에서 쉬운 길로 돌아가라는 식의 조언은 절대 하지 않을 것이다.

코드가 일단 잘 돌아가니 이유는 모른 채 그냥 지나치는 건 내 성격상 용납할 수 없다. 여러분도 그래야 한다. 여러분이 나와 함께 험난한 "가시밭길"을 탐험하면서 자바스크립트가 무엇인지, 자바스크립트로 뭘 할 수 있을지 포괄적으로 배우기 바란다. 이런 지식을 확실히 보유하고 있으면 테크닉, 프레임워크, 금주의 인기 있는 머리글자 따위는 여러분 손바닥 위에서 벗어나지 않을 것이다.

본 시리즈는 자바스크립트에 대해 가장 흔히 오해하고 있거나 잘못 이해하고 있는, 특정한 핵심 언어 요소를 선정하여 아주 깊고 철저하게 파헤친다. 여러분은 이론적으로만 알고 넘어갈 것이 아니라, 실전적으로 "내가 알고 있어야 할" 내용을 분명히 다 알고 간다는 확신을 갖고 책장을 넘기기 바란다.

아마도 지금 여러분이 알고 있는 자바스크립트는 다른 사람들이 불완전한 이해로 구

02 역자주_『더글라스 크락포드의 자바스크립트 핵심 가이드』(한빛미디어, 2008.)의 원서명 『JavaScript: The Good Parts』(O'Reilly, 2008.)를 의미합니다.

워낸 단편적인 지식들을 물려받은 정도일 것이다. 이런 자바스크립트는 진정한 자바스크립트의 그림자에 불과하다. 여러분은 지금 자바스크립트를 제대로 알고 있지 못하지만, 본 시리즈를 열독하면 완벽히 알게 될 것이다. 동료, 선후배 여러분들, 포기하지 말고 계속 읽기 바란다. 자바스크립트가 여러분의 두뇌를 기다리고 있다.

정리하기

자바스크립트는 굉장한 언어다. 다만 적당히 아는 건 쉬워도 완전히(충분히) 다 알기는 어렵다. 헷갈리는 부분이 나오면 개발자들은 대부분 자신의 무지를 탓하기 전에 언어 자체를 비난하곤 한다. 본 시리즈는 이런 나쁜 습관을 바로잡고 이제라도 여러분이 자바스크립트를 제대로, 깊이 있게 이해할 수 있도록 도와주는 것을 목표로 한다.

이 책의 예제 코드를 실행하려면 현대적인 자바스크립트 엔진(예: ES6)이 필요하다. 구 엔진(ES6 이전)에서는 코드가 작동하지 않을 수 있다.

이 책의 표기법

팁, 제안은 여기에 적습니다.

일반적인 내용은 여기에 적습니다.

경고나 유의 사항은 여기에 적습니다.

예제 코드 내려받기

보조 자료(예제 코드, 연습 문제 등)는 http://bit.ly/ydkjs-types-code에서 내려받을 수 있습니다.

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾기도 쉽지 않습니다. 또한, 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1 eBook First – 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 좀 더 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한, 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2 무료로 업데이트되는 전자책 전용 서비스입니다

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3 독자의 편의를 위해 DRM-Free로 제공합니다

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 어려운 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

부록 다양한 환경의 자바스크립트 197

chapter 1

타입

자바스크립트 같은 동적 언어는 타입type 개념이 없다고 생각하는 개발자가 많다. ECMA 표준 명세서 5.1[01] (이하 'ES5'로 줄임) "8장. 타입"을 보자.

> 이 명세에 수록된 알고리즘에서 사용되는 모든 값은 이 절에서 정의한 타입 목록 중 하나에 해당한다. 타입은 ECMAScript 언어 타입과 명세 타입으로 하위 분류된다.
>
> ECMAScript 프로그래머가 ECMAScript 언어를 이용하여 직접 조작하는 값들의 타입이 바로 ECMAScript 언어 타입이다. ECMAScript 언어 타입에는 `Undefined`, `Null`, `Boolean`, `String`, `Number`, `Object`가 있다.

엄격 타입strong type(정적 타입statical type) 형 언어의 광팬들은 "타입"이란 말의 이러한 용도를 반대할지도 모른다. 그런 언어에서 "타입"이란 말은 자바스크립트보다 훨씬 더 많은 의미를 내포하고 있기 때문이다.

한술 더 떠 자바스크립트에 "타입"이란 없으니 "태그tag"나 "하위타입subtype"이라 호칭해야 한다는 이들도 있다.

허걱! 명세의 대략적인 정의를 따르자면(명세에 선택된 단어를 따라가는 것과 같다), "타입"이란 자바스크립트 엔진, 개발자 모두에게 어떤 값을 다른 값과 분별할 수 있는, 고유한 내부 특성의 집합이다.

---

01 http://www.ecma-international.org/ecma-262/5.1/

다시 말해 기계(엔진)와 사람(개발자)이 42(숫자)란 값을 " 42 "(문자열)란 값과 다르게 취급한다면, 두 값은 타입(즉, 숫자와 문자열)이 서로 다르다. 42는 수학 연산 등 계산을 하려는 의도지만 " 42 "는 필경 페이지에 출력할 문자열 비슷한 것으로 쓸 의도로 만든 값이다. 어쨌든 이 두 값은 상이한 타입을 가지고 있다.

완벽한 정의라고 할 수는 없지만, 이 정도면 책장을 넘기는 데 지장은 없다. 자바스크립트가 실제 타입을 다루는 방식도 이와 비슷하다.

1.1 타입, 그 실체를 이해하자

사전적인 정의는 그렇다 치고 자바스크립트 언어에 타입이 있든 없든 뭐 대수란 말인가?

타입별로 내재된 특성을 제대로 알고 있어야 값을 다른 타입으로 변환하는 방법을 정확히 이해할 수 있다(4장 강제변환 참고). 어떤 형태로든 거의 모든 자바스크립트 프로그램에서 강제변환coercion이 일어나므로 타입을 확실하게 인지하고 사용하는 것이 중요하다.

42를 문자열로 보고 위치 1에서 " 2 "라는 문자를 추출하려면, 먼저 숫자 42 → 문자열 " 42 "로 변경(강제변환)해야 한다.

너무 간단한 얘기다.

하지만 강제변환은 정말 다양한 방식으로 일어난다. 삼척동자도 알 정도로 쉽고 분명하게 바뀔 때도 있지만, 조심하지 않으면 아주 이상하고 기막한 결과가 나오기도 한다.

자바스크립트 개발자에게 강제변환은 정말 헷갈리는 주제다. 오죽하면 강제변환이 자바스크립트의 언어 설계 오류이므로 무조건 피해야 한다고 혹평을 일삼는 사람들이 있겠는가.

이 책은 자바스크립트 타입 지식으로 완전 무장한 여러분이 타입 변환에 관한 터무니없고 과장된 이야기들을 내치고 강제변환의 강력함과 유용함을 십분 활용하도록 충실히 안내할 것이다. 자, 먼저 값/타입을 확실히 알고 가자.

1.2 내장 타입

자바스크립트에는 다음 7가지 내장 타입이 있다.

- `null`
- `undefined`
- `boolean`
- `number`
- `string`
- `object`
- `symbol` (ES6[02]부터 추가!)

(`object`를 제외한) 이들을 "원시 타입primitives"이라 한다.

값 타입은 `typeof` 연산자로 알 수 있다. 그럼, `typeof` 반환 값은 항상 7가지 내장 타입 중 하나일까? 놀랍게도 목록의 7가지 내장 타입과 1:1로 정확히 매치되지는 않는다.

```
typeof undefined === "undefined"; // true
typeof true === "boolean"; // true
typeof 42 === "number"; // true
typeof "42" === "string"; // true
typeof { life: 42 } === "object"; // true
```

02 역자주_ ECMAScript 6 명세는 2015년 6월 출시 예정이며, 이 책을 번역하는 현재 최신 버전은 5.1입니다.

```
// ES6부터 추가!
typeof Symbol() === "symbol"; // true
```

예제의 6개 타입은 자신의 명칭과 동일한 문자열을 반환한다. Symbol은 ES6에서 새로 추가된 데이터 타입으로 3장 네이티브에서 다룬다.

눈치챘겠지만, null은 typeof 연산 결과가 꼭 버그처럼 보인다. 특별한 녀석이라 따로 뺐다.

```
typeof null === "object"; // true
```

" null "을 반환했으면 좋겠지만(이게 정답이다!), 거의 20년 동안 이 버그는 끈덕지게 버텨왔고, 이제 와서 손을 대자니 다른 버그가 생겨 잘 돌아가던 웹 소프트웨어가 멈춰버릴 경우가 너무 많아 앞으로도 해결될 가능성은 좀처럼 없어 보인다.

그래서 타입으로 null 값을 정확히 확인하려면 조건이 하나 더 필요하다.

```
var a = null;
(!a && typeof a === "object"); // true
```

null은 " falsy[03] "한(false나 다름없는, 4장 강제변환 참고) 유일한 원시 값이지만, 타입은 " object "인 특별한 존재다.

typeof가 반환하는 문자열은 하나 더 있다.

03 역자주_ truthy/falsy는 각각 true/false 뒷부분에 y를 붙여 변형한 형태로 '~스럽다'는 의미의 영어 특유의 뉘앙스가 있습니다. 이 단어를 많은 자바스크립트 도서에서 '참 같은 참', '거짓 같은 거짓', '참스러움', '거짓스러움' 등으로 번역을 시도했는데, 제 생각에는 truthy/falsy 역시 자바스크립트의 예약어처럼 true/false와 동일한 레벨에서 원어를 있는 그대로 표기하는 편이 독자 여러분이 의미를 받아들이는 데 도움이 될 것 같습니다. true/false를 일일이 참/거짓이라고 옮기지 않는 것과 같은 이치며, '불리언 문맥 상 true/false로 봐야 하는' 값으로 이해하기 바랍니다.

```
typeof function a(){ /* .. */ } === "function"; // true
```

typeof 반환 값을 보면, 마치 function이 최상위 레벨의 내장 타입처럼 보이지만 명세를 읽어보면 실제로는 object의 "하위 타입"이다. 구체적으로 설명하면 함수는 "호출 가능한 객체callable object"(내부 프로퍼티 [[Call]]로 호출할 수 있는 객체)라고 명시되어 있다.

사실 함수는 객체라서 유용하다. 무엇보다 함수에 프로퍼티를 둘 수 있다.

```
function a(b,c) {
  /* .. */
}
```

함수에 선언된 파라미터 개수는 함수 객체의 length 프로퍼티로 알 수 있다.

```
a.length; // 2
```

함수 a는 두 개(b, c)의 파라미터를 가지므로 "함수의 길이length"는 2다.

배열은 어떨까? 자바스크립트의 터줏대감답게 독특한 타입일까?

```
typeof [1,2,3] === "object"; // true
```

그냥 객체다. 배열은 (키가 문자열인 객체와 반대로) 숫자 인덱스를 가지며, length 프로퍼티가 자동으로 관리되는 등의 추가 특성을 지닌, 객체의 "하위 타입"이라 할 수 있다.

1.3 값은 타입을 가진다

값에는 타입이 있지만, 변수엔 따로 타입이란 없다. 변수는 언제라도, 어떤 형태의 값이라도 가질 수 있다.

자바스크립트는 "타입 강제type enforcement"를 하지 않는다. 변숫값이 처음에 할당된 값과 동일한 타입일 필요는 없다. 문자열을 넣었다가 나중에 숫자를 넣어도 상관없다.

42는 내장된 숫자 타입의 값이고, 이 타입은 절대로 바꿀 수 없다. " 42 " 는 문자열 타입의 값이지만 숫자 42에서 (강제변환(4장 강제변환 참고) 과정을 거쳐) 생성할 수 있다.

변수에 typeof 연산자를 대어보는 건 "이 변수의 타입은 무엇이니?"라는 질문과 같지만, 실은 타입이란 개념은 변수에 없으므로 정확히는 "이 변수에 들어있는 값의 타입은 무엇이니?"라고 묻는 것이다.

```
var a = 42;
typeof a; // "number"

a = true;
typeof a; // "boolean"
```

typeof 연산자의 반환 값은 언제나 문자열이다.

```
typeof typeof 42; // "string"
```

따라서 typeof 42는 " number " 를 반환하고, typeof " number " 의 결과값은 " string " 이다.

1.3.1 값이 없는 vs 선언되지 않은

값이 없는 변수의 값은 undefined이며, typeof 결과는 " undefined " 다.

```
var a;

typeof a; // "undefined"

var b = 42;
var c;

// 그러고 나서,
b = c;

typeof b; // "undefined"
typeof c; // "undefined"
```

" undefined " (값이 없는)와 " undeclared " (선언되지 않은)를 동의어처럼 생각하기 쉬운데, 자바스크립트에서 둘은 완전히 다른 개념이다.

" undefined " 는 접근 가능한 스코프에 변수가 선언되었으나 현재 아무런 값도 할당되지 않은 상태를 가리키는 반면, " undeclared " 는 접근 가능한 스코프에 변수 자체가 선언조차 되지 않은 상태를 의미한다.

```
var a;

a; // undefined
b; // ReferenceError: b is not defined (참조에러: b가 정의되지 않았습니다)
```

여기서 브라우저 에러 메시지가 다소 헷갈린다. " b is not defined " 란 말은 결국 " b is undefined " 란 말 아닌가? 하지만 " undefined " 와 " is not defined " 는 의미가 완전히 다르다는 걸 상기하자. 처음부터 " b is not found " 나 " b is not declared " 라고 했으면 명쾌하고 좋으련만!

선언되지 않은undeclared 변수의 typeof 연산 결과는 더 헷갈린다.

```
var a;

typeof a; // "undefined"

typeof b; // "undefined"
```

선언되지 않은 변수도 typeof하면 “ undefined ”로 나온다. b는 분명 선언조차 하지 않은 변수인데 typeof b를 해도 브라우저는 오류 처리를 하지 않는다. 바로 이것이 typeof만의 독특한 안전 가드safety guard다.

이것도 “typeof 선언되지 않은 변수”를 “ undeclared ” 정도로 했으면, 진짜 “ undefined ”인 변수와 쓸데없이 충돌할 일도 없었을 텐데 참 아쉽다.

1.3.2 선언되지 않은 변수

그런데 브라우저에서 자바스크립트 코드를 처리할 때, 특히 여러 스크립트 파일의 변수들이 전역 명칭공간namespace을 공유할 때, typeof의 안전 가드는 의외로 쓸모가 있다.

간단한 예로, 프로그램의 “디버그 모드”를 DEBUG라는 전역 변수(플래그)로 조정한다고 치자. 콘솔 창에 메시지 로깅 등 디버깅 작업을 수행하기 전, 이 변수의 선언 여부를 체크해야 할 것이다. 최상위 전역 스코프에 var DEBUG = true라고 “ debug.js ” 파일에만 선언하고, 개발/테스트 단계(운영 단계는 제외)에서 이 파일을 브라우저가 로딩하기만 하면 될 것이다.

그러나 나머지 애플리케이션 코드에서 ReferenceError가 나지 않게 하려면 조심해서 DEBUG 전역 변수를 체크해야 한다. 바로 이럴 때 우리의 친구, typeof 안전 가드가 제 몫을 해낸다.

자신이 작성한 코드의 모든 변수는 전역 명칭공간에는 전혀 없고, 오직 전용private 또는 별도의 명칭공간에만 있다고 자신 있게 말하는 개발자들이 있다. 이론적으로는 그럴 듯하지만 실제는 거의 불가능한 소리다. 물론 그런 방향으로 코딩하려는 자세는 좋다! 다행히 ES6부터는 모듈을 일급first-class 개념으로 지원하기 때문에 현실적으로 가능할 것 같다.[04]

```
// 헉, 이렇게 하면 에러가 난다!
if (DEBUG) {
  console.log( "디버깅을 시작합니다" );
}

// 이렇게 해야 안전하게 존재 여부를 체크할 수 있다.
if (typeof DEBUG !== "undefined") {
  console.log( "디버깅을 시작합니다" );
}
```

(DEBUG 같은) 임의로 정의한 변수를 쓰지 않더라도 이런 식으로 체크하는 것이 편리하며, 내장 API 기능을 체크할 때에도 에러가 나지 않게 도와준다.

04 역자주_ ES6부터는 import/export 키워드로 외부 모듈의 함수/변수를 사용할 수 있습니다. 다음 예제를 참고하시기 바랍니다.

```
[utility.js]
function sum(a, b) {
  return a + b;
}
function product(a, b) {
  return a * b;
}

export { product, sum }

[app.js]
import { product, sum } from 'utility';

console.log(product(1, 2)); //2
console.log(sum(1, 2)); //3
```

```
if (typeof atob === "undefined") {
  atob = function() { /*..*/ };
}
```

존재하지 않는 기능을 추가하기 위해 "폴리필polyfill"을 정의하려면 atob 선언문에서 var 키워드를 빼는 편이 좋다. if 문 블록에 var atob로 선언하면, (전역 atob는 이미 존재하므로) 코드 실행을 건너뛰더라도 선언 자체가 최상위 스코프로 호이스팅 hoisting된다[05] (『You Don't Know JS: 스코프와 클로저』[06] 참고). 이렇게 (보통 호스트 객체host object라고 부르는) 특수한 타입의 전역 내장 변수를 중복 선언하면 에러를 던지는 브라우저가 있다. 명시적으로 var를 빼야 선언문이 호이스팅되지 않는다.

typeof 안전 가드 없이 전역 변수를 체크하는 다른 방법은 전역 변수는 모두 전역 객체(브라우저는 window)의 프로퍼티라는 점을 이용하는 것이다. 그래서 다음과 같이 (꽤 안전하게) 체크할 수 있다.

```
if (window.DEBUG) {
  // ..
}

if (!window.atob) {
  // ..
}
```

05 역자주_ 자바스크립트에서 변수 선언은 항상 최상위로 끌어올려집니다 → 호이스팅(hoisting)됩니다. 즉, var atob로 선언하면 자바스크립트 엔진은 다음 코드로 해석합니다.

```
var atob; // 선언문이 호이스팅된다!
if (typeof atob === "undefined") {
  atob = function() { /*..*/ };
}
```

06 역자주_ 이 글을 번역하는 현재 본 시리즈 중 유일하게 한글판 번역서가 출간된 책입니다. http://goo.gl/YCqO7K

선언되지 않은 변수 때와는 달리, 어떤 객체(전역 window 객체도 포함해서)의 프로퍼티를 접근할 때 그 프로퍼티가 존재하지 않아도 ReferenceError가 나지는 않는다.

하지만 window 객체를 통한 전역 변수 참조는 가급적 삼가는 것이 좋다. 전역 변수를 꼭 window 객체로만 호출하지 않는, 다중 자바스크립트 환경(브라우저뿐만 아니라 서버에서 실행되는 노드JS[node.js]가 일례다)이라면 더욱 그렇다.

엄밀히 말해서 typeof 안전 가드는 (비록 일반적인 경우는 아니지만) 전역 변수를 사용하지 않을 때에도 유용한데, 일부 개발자들은 이런 설계 방식이 그다지 바람직하지 않다고 말한다. 이를테면 다른 개발자가 여러분이 작성한 유틸리티 함수를 자신의 모듈/프로그램에 카피 앤 페이스트[copy-and-paste]하여 사용하는데, 가져다 쓰는 프로그램에 유틸리티의 특정 변숫값이 정의되어 있는지 체크해야 하는 상황을 가정해보자.

```
function doSomethingCool() {
  var helper =
    (typeof FeatureXYZ !== "undefined") ?
    FeatureXYZ :
    function() { /*.. 기본 XYZ 기능 ..*/ };

  var val = helper();
  // ..
}
```

doSomethingCool 함수는 FeatureXYZ 변수가 있으면 그대로 사용하고 없으면 함수 바디를 정의한다. 이렇게 해야 다른 사람이 카피 앤 페이스트를 해도 안전하게 FeatureXYZ가 존재하는지를 체크할 수 있다.

```
// IIFE (즉시호출함수표현식, 『You Don't Know JS: 스코프와 클로저』[07] 참조)
(function(){
  function FeatureXYZ() { /*.. 나의 XYZ 기능 ..*/ }

  // 'doSomethingCool(..)'를 포함
  function doSomethingCool() {
    var helper =
      (typeof FeatureXYZ !== "undefined") ?
      FeatureXYZ :
      function() { /*.. 기본 XYZ 기능 ..*/ };

    var val = helper();
    // ..
  }

  doSomethingCool();
})();
```

FeatureXYZ는 전역 변수가 아니지만, typeof 안전 가드를 이용하여 안전하게 체크하고 있다. 그리고 이 코드에선 (window.—— 식으로 전역 변수에 했던 것처럼) 체크 용도로 사용할 만한 객체가 없기 때문에 typeof가 꽤 요긴하다.

"의존성 주입dependency injection" 설계 패턴을 선호하는 개발자들도 있다. FeatureXYZ가 doSomethingCool( )의 바깥이나 언저리에 정의되었는지 암시적으로 조사하는 대신, 다음 코드처럼 명시적으로 의존 관계를 전달하는 것이다.

```
function doSomethingCool(FeatureXYZ) {
  var helper = FeatureXYZ ||
    function() { /*.. 기본 XYZ 기능 ..*/ };

  var val = helper();
  // ..
}
```

07 http://goo.gl/YCqO7K

다양한 설계 옵션이 가능하지만 접근 방식에 따라 장, 단점이 고루 있어서 어떤 것이 완전히 "맞다"고 "틀리다"고 할 수는 없다. 하지만 대체로 typeof 안전 가드가 선택할 수 있는 옵션이 많아서 좋다.

1.4 정리하기

자바스크립트에는 7가지 내장 타입(null, undefined, boolean, number, string, object, symbol)이 있으며, typeof 연산자로 타입명을 알아낸다.

변수는 타입이 없지만 값은 타입이 있고, 타입은 값의 내재된 특성을 정의한다.

"undefined"와 "undeclared"가 대충 같다고 보는 개발자들이 많은데, 자바스크립트 엔진은 둘을 전혀 다르게 취급한다. undefined는 선언된 변수에 할당할 수 있는 값이지만, undeclared는 변수 자체가 선언된 적이 없음을 나타낸다.

불행히도 자바스크립트는 이 두 용어를 대충 섞어버려, 에러 메시지("ReferenceError: a is not defined")뿐만 아니라 typeof 반환 값도 모두 "undefined"로 뭉뚱그린다.

그래도 (에러를 내지 않는) typeof 안전 가드 덕분에 선언되지 않은 변수에 사용하면 제법 쓸 만하다.

값

배열, 문자열, 숫자는 모든 프로그램의 가장 기본적인 구성 요소지만 자바스크립트에서는 독특한 특성을 갖고 있어 개발자를 웃게도 울게도 만든다. 자바스크립트에 내장된 값 타입과 작동 방식을 살펴보고 정확하게 사용할 수 있도록 완전히 이해하자.

2.1 배열

타입이 엄격한 다른 언어에 비해 자바스크립트 배열은 문자열, 숫자, 객체는 물론, 심지어 다른 배열에 이르기까지(이런 식으로 다차원 배열을 만든다), 어떤 타입의 값이라도 담을 수 있는 그릇이다.

```
var a = [ 1, "2", [3] ];

a.length; // 3
a[0] === 1; // true
a[2][0] === 3; // true
```

배열 크기는 미리 정하지 않고도 선언할 수 있으며 원하는 값을 추가하면 된다.

```
var a = [ ];

a.length; // 0
```

```
a[0] = 1;
a[1] = "2";
a[2] = [ 3 ];

a.length; // 3
```

배열 값에 delete 연산자를 적용하면 슬롯slot을 제거할 수 있지만, 마지막 원소까지 제거해도 length 프로퍼티 값까지 바뀌지 않는다는 점을 주의하자! delete 연산자는 5장 문법에서 자세히 다룬다.

(빈/빠진 슬롯이 있는) "구멍 난sparse" 배열을 다룰 때는 조심해야 한다.

```
var a = [ ];

a[0] = 1;
// 'a[1]' 슬롯을 건너뛰었다!
a[2] = [ 3 ];

a[1]; // undefined

a.length; // 3
```

실행은 되지만 이런 코드에서 중간에 건너뛴 "빈 슬롯"은 혼란을 부추길 수 있다. a[1] 슬롯 값은 응당 undefined가 될 것 같지만, 명시적으로 a[1] = undefined 세팅한 것과 똑같지는 않다.

배열 인덱스는 숫자인데, 배열 자체도 하나의 객체여서 키/프로퍼티 문자열을 추가할 수 있다(하지만 배열 length가 증가하지는 않는다)는 점이 다소 까다롭다.

```
var a = [ ];

a[0] = 1;
a["foobar"] = 2;

a.length; // 1
```

```
a["foobar"]; // 2
a.foobar; // 2
```

그런데 키로 넣은 문자열 값이 표준 10진수 숫자로 타입 변환되면, 마치 문자열 키가 아닌, 숫자 키를 사용한 것 같은 결과가 초래된다는 점은 정말 주의해야 할 함정이다!

```
var a = [ ];

a["13"] = 42;

a.length; // 14
```

일반적으로 배열에 문자열 타입의 키/프로퍼티를 두는 건 추천하고 싶지 않다. 그렇게 해야 한다면 객체를 대용하고, 배열 원소의 인덱스는 확실히 숫자만 쓰자.

2.1.1 유사 배열

유사 배열 값(숫자 인덱스가 가리키는 값들의 집합)을 진짜 배열로 바꾸고 싶을 때가 더러 있다. 이럴 때는 배열 유틸리티 함수(`indexOf( .. )`, `concat( .. )`, `forEach( .. )` 등)를 사용하여 해결하는 것이 일반적이다.

예를 들어, DOM 쿼리 작업을 수행하면, 비록 배열은 아니지만 변환 용도로는 충분한, 유사 배열 형태의 DOM 원소 리스트가 반환된다. 다른 예로, 함수에서 (배열 비슷한) `arguments` 객체를 사용하여 인자를 리스트로 가져오는 것(ES6부터 비 권장 deprecated)도 마찬가지다.

이런 변환은 `slice( .. )` 함수의 기능을 차용하는 방법을 가장 많이 쓴다.

```
function foo() {
  var arr = Array.prototype.slice.call( arguments );
```

```
  arr.push( "bam" );
  console.log( arr );
}

foo( "bar", "baz" ); // ["bar","baz","bam"]
```

예제 코드에서 알 수 있듯이 `slice( )` 함수에 인자가 없으면 기본 파라미터 값으로 구성된 배열(여기서는 유사 배열)을 복사한다.[01]

ES6부터는 기본 내장 함수 `Array.from( .. )`가 이 일을 대신 한다.

```
...
var arr = Array.from( arguments );
...
```

`Array.from( .. )`에는 다른 강력한 기능도 있는데, 『You Don't Know JS: ES6 & Beyond』[02]에서 자세히 다룬다.

2.2 문자열

문자열은 흔히 단지 문자의 배열이라고 생각한다. 엔진이 내부적으로 배열을 쓰도록 구현되었는지는 모르겠지만 자바스크립트 문자열은 실제로 생김새만 비슷할 뿐, 문자 배열과 같지 않다는 사실을 알아야 한다.

다음 두 값을 보자.

01 역자주_ 즉, Array.prototype.slice.call(arguments)는 Array.prototype.slice.call(arguments, 0)와 같습니다. 첫 번째 원소부터(인덱스는 0부터 시작) 끝까지 잘라내므로(slice) 배열을 복사하는 것이나 다름없습니다.

02 역자주_ 이 글을 번역하는 현재 출간되지 않았고, 2015년 6월 12일 이후 원서가 출간될 예정입니다. http://goo.gl/ODcZu4

```
var a = "foo";
var b = ["f","o","o"];
```

문자열은 배열과 겉모습은 닮았다(유사 배열이다). 이를테면 둘 다 length 프로퍼티, indexOf(..) 메서드(ES5 배열에만 있음), concat(..) 메서드를 가진다.

```
// 앞 코드에서 계속됨

a.length; // 3
b.length; // 3

a.indexOf( "o" ); // 1
b.indexOf( "o" ); // 1

var c = a.concat( "bar" ); // "foobar"
var d = b.concat( ["b","a","r"] ); // ["f","o","o","b","a","r"]

a === c; // false
b === d; // false

a; // "foo"
b; // ["f","o","o"]
```

그럼, 어쨌든 기본적으로는 둘 다 "문자의 배열"이라고 할 수 있을까? 그렇지 않다.

```
a[1] = "0";
b[1] = "0";

a; // "foo"
b; // ["f","0","o"]
```

문자열은 불변 값immutable이지만 배열은 가변 값mutable이다. a[1]처럼 문자열의 특정 문자를 접근하는 형태는 모든 자바스크립트 엔진에서 유효한 것은 아

니다. 실제로 IE 구버전은 이를 문법 에러로 인식한다(요즘 버전은 그렇지 않다)[03]. `a.charAt(1)`으로 접근해야 맞다.

한 가지 더, 문자열은 불변 값이므로 문자열 메서드는 그 내용을 바로 변경하지 않고 항상 새로운 문자열을 생성한 후 반환한다. 반면에, 대부분의 배열 메서드는 그 자리에서 곧바로 원소를 수정한다.

```
c = a.toUpperCase();
a === c; // false
a; // "foo"
c; // "FOO"

b.push( "!" );
b; // ["f","O","o","!"]
```

그리고 문자열을 다룰 때 유용한 대부분의 배열 메서드는 사실상 문자열에 쓸 수 없지만, 문자열에 대해 불변 배열 메서드를 빌려 쓸 수는 있다.

```
a.join; // undefined
a.map; // undefined

var c = Array.prototype.join.call( a, "-" );
var d = Array.prototype.map.call( a, function(v){
  return v.toUpperCase() + ".";
} ).join( "" );

c; // "f-o-o"
d; // "F.O.O."
```

다음은 문자열의 순서를 거꾸로 뒤집는 코드다(자바스크립트와 관련된 면접 시 자주 출제되는 문제다). 배열에는 `reverse()`라는 가변 메서드가 준비되어 있지만, 문자열

03 역자주_ 정확히는 IE 7까지 a[1] 값은 undefined로 나옵니다. IE 8부터 a[1]에 o가 할당됩니다.

은 그렇지 않다.

```
a.reverse; // undefined

b.reverse(); // ["!","o","O","f"]
b; // ["!","o","O","f"]
```

불행히도 문자열은 불변 값이라 바로 변경되지는 않으므로 배열의 가변 메서드는 통하지 않고, 그래서 "빌려쓰는 것" 또한 안 된다.

```
Array.prototype.reverse.call( a );[04]
// 여전히 String 객체 래퍼를 반환한다(3장 네이티브 참고)
// for "foo" :(
```

일단 문자열을 배열로 바꾸고 원하는 작업을 수행한 후 다시 문자열로 되돌리는 것이 또 다른 꼼수(전문 용어로는 핵hack이라고 함)다.

```
var c = a
  // 'a'를 문자의 배열로 분할한다
  .split( "" )
  // 문자 배열의 순서를 거꾸로 뒤집는다
  .reverse()
  // 문자 배열을 합쳐 다시 문자열로 되돌린다
  .join( "" );

c; // "oof"
```

영 내키지는 않지만 단순 문자열에 대해 좀 지저분하더라도 빠르게 써먹을 방법이

04 역자주_ 파이어폭스는 '타입에러: Array.prototype.reverse.call(..)은 읽기 전용입니다', IE는 9 버전부터 '개체는 이 기능을 지원하지 않습니다', IE 8 버전 이하와 크롬은 에러 없이 그냥 무시합니다.

필요하다면 이런 방법도 나쁘지는 않다.

하지만 조심하라! 복잡한 (유니코드) 문자가 섞여 있는 경우(특수 문자, 멀티바이트multibyte 문자 등), 이 방법은 통하지 않는다. 제대로 처리하려면 유니코드를 인식하는 정교한 라이브러리 유틸리티가 필요하다. 마티아스 바이넨Mathias Bynens이 만든 에스레베르를 참고하자.[05]

"문자열" 자체에 어떤 작업을 빈번하게 수행하는 경우라면 관점을 달리하여 문자열을 문자 단위로 저장하는 배열로 취급하는 것이 더 나을 수도 있다. '문자열 ↔ 배열' 변환을 매번 번거롭게 신경쓰지 않아도 되니 시간과 노력을 아낄 수 있다. 문자열로 나타내야 할 때는 언제나 문자 배열에 `join( " " )` 메서드를 호출하면 된다.

2.3 숫자

자바스크립트의 숫자 타입은 number가 유일하며, "정수integer", "부동 소수점 숫자fractional decimal number"를 모두 아우른다. "정수"에 따옴표를 친 건, 다른 언어와 달리 자바스크립트에는 진정한 정수가 없다는 이유로 오랫동안 욕을 먹어왔기 때문이다. 언젠가 개선될 날이 오긴 하겠지만, 일단 현재는 모든 숫자를 number 타입 하나로만 표시한다.

따라서 자바스크립트 "정수"는 부동 소수점 값이 없는 값이다(예: `42.0`은 "정수" 42와 같다).

사실상 모든 스크립트 언어를 통틀어 대부분의 현대 프로그래밍 언어는 " IEEE 754 " 표준(부동 소수점 표준)을 따른다. 자바스크립트 number도 IEEE 754 표준을

05 역자주_ 에스레베르(Esrever)는 리버스(Reverse)를 거꾸로 한 것입니다. 프로그램 이름만 보아도 무슨 기능을 하는지 알 수 있도록 재미있게 명명한 것입니다.
https://github.com/mathiasbynens/esrever

따르며, 그 중에서도 정확히는 "배 정도double precision" 표준 포맷(64비트 바이너리)"을 사용한다.

웹 서핑을 하다 보면 이진 부동 소수점 숫자가 메모리에 저장되는 방식과 그렇게 설계된 의미는 무엇인지 시시콜콜 자세히 설명한, 좋은 참고 자료가 많다. 자바스크립트 number의 정확한 사용 방법을 이해하고자 메모리 비트 패턴까지 알아야 할 필요는 없으니 IEEE 754 상세 내용을 알고 싶은 독자들은 따로 자료를 검색하여 읽어보자.

2.3.1 숫자 구문

자바스크립트 숫자 리터럴은 다음과 같이 10진수 리터럴로 표시한다.

```
var a = 42;
var b = 42.3;
```

소수점 앞 정수가 0이면 생략 가능하다.

```
var a = 0.42;
var b = .42;
```

소수점 이하가 0일 때도 생략 가능하다.

```
var a = 42.0;
var b = 42.;
```

42.처럼 표기하는 건 일반적이지도 않고 다른 사람이 코드를 읽을 때 혼동을 일으킬 수 있으니 별로 좋은 생각은 아니다. 어쨌든 틀린 코드는 아니다.

대부분의 숫자는 10진수가 디폴트이고 소수점 이하 0은 뗀다.

```
var a = 42.300;
var b = 42.0;

a; // 42.3
b; // 42
```

아주 크거나 아주 작은 숫자는 지수형exponent form으로 표시하며, toExponential( ) 메서드의 결과값과 같다.

```
var a = 5E10;
a; // 50000000000
a.toExponential(); // "5e+10"

var b = a * a;
b; // 2.5e+21

var c = 1 / a;
c; // 2e-11
```

숫자 값은 Number 객체 래퍼wrapper로 박싱boxing할 수 있기 때문에 Number.prototype 메서드로 접근할 수도 있다(3장 네이티브 참고). 예를 들면, toFixed(..) 메서드는 지정한 소수점 이하 자릿수까지 숫자를 나타낸다.

```
var a = 42.59;

a.toFixed( 0 ); // "43"
a.toFixed( 1 ); // "42.6"
a.toFixed( 2 ); // "42.59"
a.toFixed( 3 ); // "42.590"
a.toFixed( 4 ); // "42.5900"
```

실제로는 숫자 값을 문자열 형태로 반환하며, 원래 값의 소수점 이하 숫자보다 더 많은 자릿수를 지정하면 그만큼 0이 우측에 붙는다. toPrecision(..)도 기능

은 비슷하지만 유효 숫자 개수를 지정할 수 있다.

```
var a = 42.59;

a.toPrecision( 1 ); // "4e+1"
a.toPrecision( 2 ); // "43"
a.toPrecision( 3 ); // "42.6"
a.toPrecision( 4 ); // "42.59"
a.toPrecision( 5 ); // "42.590"
a.toPrecision( 6 ); // "42.5900"
```

두 메서드는 숫자 리터럴에서 바로 접근할 수 있으므로 굳이 변수를 만들어 할당하지 않아도 된다. 하지만 .이 소수점일 경우엔 프로퍼티 접근자accessor가 아닌 숫자 리터럴의 일부로 해석되므로 . 연산자를 사용할 때는 조심하자.

```
// 잘못된 구문
42.toFixed( 3 ); // SyntaxError

// 모두 올바른 구문
(42).toFixed( 3 ); // "42.000"
0.42.toFixed( 3 ); // "0.420"
42..toFixed( 3 ); // "42.000"
```

42.toFixed(3)에서 구문 에러가 난 이유는 .이 42. 리터럴(맞는 표현이다!)의 일부가 되어 버려 .toFixed 메서드에 접근할 수단이 없기 때문이다.

42..toFixed(3)의 경우, 첫 번째 .은 숫자 리터럴의 일부, 두 번째 .은 프로퍼티 연산자로 해석되므로 문제가 없다. 하지만 보기에 어색하고 불편하므로 이런 코드는 잘 쓰지 않는다. 사실, 원시 값이 메서드를 직접 호출할 일이 거의 없다(물론 드물다고 해서 나쁘거나 틀렸다는 소리는 아니다).

Number.prototype을 확장하여 숫자 값 연산 기능을 추가한 라이브러리들도 있다. 하늘에서 10초 동안 돈다발이 비처럼 쏟아지게 하는 10..makeItRain() 같은 코드가 있을지 모를 일이다. 어쨌든 이런 라이브러리를 구해 써도 된다.

다음 코드 역시 옳다(. 앞에 공란이 있다).

```
42 .toFixed(3); // "42.000"
```

그러나 숫자 리터럴에 이런 코딩을 해놓으면 다른 개발자들의 (작성자 본인까지도) 안구를 메마르게 할 뿐이니 그러지 말자.

큰 숫자는 보통 다음과 같이 지수형으로 표시한다.

```
var onethousand = 1E3; // 1 * 10^3
var onemilliononehundredthousand = 1.1E6; // 1.1 * 10^6
```

숫자 리터럴은 2진, 8진, 16진 등 다른 진법으로도 나타낼 수 있다.

```
0xf3; // 243의 16진수
0Xf3; // 위와 같음
0363; // 243의 8진수
```

ES6+ 엄격 모드strict mode에서는 0363처럼 0을 앞에 붙여 8진수를 표시하지 못한다(새 방식은 바로 다음에 설명한다). 느슨한 모드non-strict mode에서는 과거 형식을 계속 쓸 수는 있지만 미래를 생각해서 쓰지 않는 게 좋다(이제부턴 엄격 모드에서 코딩해야 하므로!).

ES6부터는 다음과 같이 쓸 수도 있다.

```
0o363; // 243의 8진수
0O363; // 위와 같음
```

```
0b11110011; // 243의 이진수
0B11110011; // 위와 같음
```

옆에 동료 개발자가 앉아 있다면, 제발 00363처럼 코딩하지 말아달라고 정중히 부탁해보자. 0 다음 O는 쓸데없이 헷갈리니까 언제나 소문자로 0x, 0b, 0o와 같이 표기하기 바란다.

2.3.2 작은 소수 값

다음은 널리 알려진 이진 부동 소수점 숫자의 부작용을 알아보자(다른 언어는 모두 IEEE 754 표준을 준수하지만, 많은 이들의 예상과는 달리 자바스크립트는 그렇지 않다).

```
0.1 + 0.2 === 0.3; // false
```

수식만 보면 분명 true다. 그런데 결과는 false다?

간단히 말하면, 이진 부동 소수점으로 나타낸 0.1과 0.2는 원래의 숫자와 일치하지 않는다. 그래서 둘을 더한 결과 역시 정확히 0.3이 아니다. 실제로는 0.30000000000000004에 가깝지만, "가깝다고" 해도 "같은" 것은 아니다.

그럼 자바스크립트가 모든 값을 정확하게 표시하도록 숫자 구현 방식을 바꾸어야 맞을까? 그렇게 생각하는 사람들도 있다. 그래서 지난 수십 년 동안 갖가지 대안이 제시되었지만 어느 것도 채택되지 않았고 앞으로도 사정은 비슷할 것 같다. 누군가 손을 번쩍 들고 "제가 버그를 고쳤습니다!"라고 쉽게 말할 수 있을 것 같지만 그리 간단치가 않다. 간단했다면 머리 좋은 사람들이 벌써 오래 전에 다 풀어놨을 것이다.

그렇다면 한번 생각해보자. 숫자 값이 정확하리라는 보장이 없다면 숫자를 쓰지 말라는 소린가? 물론 그렇지 않다.

부동 소수점 숫자를 조심히 다루어야 할 애플리케이션도 있겠지만, 많은 애플리케

이션이 (아마 대부분?) 전체수whole number ("정수integer")[06]만을, 그것도 기껏해야 백만이나 조 단위 규모의 숫자를 다룬다. 이런 상황이라면 언제나 안심하고 자바스크립트의 숫자 연산 기능을 믿고 써도 된다.

그럼 `0.1 + 0.2`와 `0.3`, 두 숫자는 어떻게 비교해야 할까?

가장 일반적으로는 미세한 "반올림 오차"를 허용 공차tolerance로 처리하는 방법이 있다. 이렇게 미세한 오차를 "머신 입실론machine epsilon[07]"이라고 하는데, 자바스크립트 숫자의 머신 입실론은 2^{-52} (`2.220446049250313e-16`)다.

ES6부터는 이 값이 `Number.EPSILON`으로 미리 정의되어 있으므로 필요 시 사용하면 되고, ES6 이전 브라우저는 다음과 같이 폴리필을 대신 사용한다.

```
if (!Number.EPSILON) {
  Number.EPSILON = Math.pow(2,-52);
}
```

`Number.EPSILON`으로 두 숫자의 (반올림 허용 오차 이내의) "동등함equality"을 비교할 수 있다.

```
function numbersCloseEnoughToEqual(n1,n2) {
  return Math.abs( n1 - n2 ) < Number.EPSILON;
}

var a = 0.1 + 0.2;
```

06 역자주_ 전체수(whole number)란 0과 양수를 포함한 숫자, 정수(integer)는 다들 알고 계신 것처럼 음수와 0, 양수를 포함한 숫자를 말합니다. 자연수(natural number)는 양수만을 가리킵니다.

07 역자주_ 머신 입실론은 컴퓨터가 이해할 수 있는 가장 작은 숫자 단위를 말한다. 컴퓨터는 실수를 이진수의 형태로 저장하기 때문에 1/3과 같은 숫자를 저장하는 것은 불가능하다. 하지만 대부분의 경우에 무한히 긴 0.33333333…과 같이 표현할 수 있다. 그러나 마지막 값은 일반적으로 3이 아니라 2나 4가 된다. 이와 같이 컴퓨터가 다룰 수 있는 가장 작은 수, 즉 임계 값을 머신 입실론이라 한다(출처: 컴퓨터인터넷IT용어대사전, 전산용어사전편찬위원회, 2011.1.20, 일진사 http://goo.gl/y5bocj).

```
var b = 0.3;

numbersCloseEnoughToEqual( a, b ); // true
numbersCloseEnoughToEqual( 0.0000001, 0.0000002 ); // false
```

부동 소수점 숫자의 최댓값은 대략 `1.798e+308`이고(정말 엄청, 엄청 큰 수다!) `Number.MAX_VALUE`로 정의하며, 최솟값은 `5e-324`로 음수는 아니지만 거의 `0`에 가까운 숫자고, `Number.MIN_VALUE`로 정의한다.

2.3.3 안전한 정수 범위

숫자를 표현하는 방식이 이렇다 보니, 정수는 `Number.MAX_VALUE`보다 훨씬 작은 수준에서 "안전safe" 값의 범위가 정해져 있다.

"안전하게" 표현할 수 있는(즉, 표현한 값과 실제 값이 정확하게 일치한다고 장담할 수 있는) 정수는 최대 $2^{53} - 1$ (`9007199254740991`)다. 세 자리 콤마를 찍어보면 얼추 9천조가 넘는다. 아주 끝내주게 널널한 수치다.

이 값은 ES6에서 `Number.MAX_SAFE_INTEGER`로 정의한다. 최솟값은 `Number.MIN_SAFE_INTEGER`로 정의하며, `-9007199254740991`다.

자바스크립트 프로그램에서 이처럼 아주 큰 숫자에 맞닥뜨리는 경우는 데이터베이스 등에서 64비트 ID를 처리할 때가 대부분이다. 64비트 숫자는 숫자 타입으로 정확하게 표시할 수 없으므로 (보내고 받을 때) 자바스크립트 `string` 타입으로 저장해야 한다.

다행히 그렇게 큰 ID 값을 숫자 연산할 일은 흔치 않다. 하지만 아주 큰 수를 다룰 수밖에 없는 상황이라면, 지금으로선 큰 수big number 유틸리티[08] 사용을 권하고 싶

08 역자주_ 여러 라이브러리가 있지만, 우선 https://github.com/peterolson/BigInteger.js를 참고하기 바랍니다.

다. 차기 자바스크립트 버전에서 큰 수를 공식 지원할 가능성도 있다.

2.3.4 정수인지 확인

ES6부터는 `Number.isInteger( .. )`로 어떤 값의 정수 여부를 확인한다.

```
Number.isInteger( 42 ); // true
Number.isInteger( 42.000 ); // true
Number.isInteger( 42.3 ); // false
```

ES6 이전 버전을 위한 폴리필은 다음과 같다.

```
if (!Number.isInteger) {
  Number.isInteger = function(num) {
    return typeof num == "number" && num % 1 == 0;
  };
}
```

안전한 정수 여부는 ES6부터 `Number.isSafeInteger( .. )`로 체크한다.

```
Number.isSafeInteger( Number.MAX_SAFE_INTEGER ); // true
Number.isSafeInteger( Math.pow( 2, 53 ) ); // false
Number.isSafeInteger( Math.pow( 2, 53 ) - 1 ); // true
```

폴리필은 다음과 같다.

```
if (!Number.isSafeInteger) {
  Number.isSafeInteger = function(num) {
    return Number.isInteger( num ) &&
      Math.abs( num ) <= Number.MAX_SAFE_INTEGER;
  };
}
```

2.3.5 32비트 (부호 있는) 정수

정수의 "안전 범위"가 대략 9천 조(53비트)에 이르지만, (비트 연산bitwise operation 처럼) 32비트 숫자에만 가능한 연산이 있으므로 실제 범위는 훨씬 줄어든다.

따라서 정수의 안전 범위는 `Math.pow(-2,31)` (-2147483648, 약 -21억)에서 `Math.pow(2,31)-1` (2147483647, 약 +21억)까지다.

`a | 0`와 같이 쓰면 '숫자 값 → 32비트 부호 있는 정수'로 강제변환한다. | 비트 연산자는 32비트 정수 값에만 쓸 수 있기 때문에 (즉, 32비트까지만 관심을 갖기 때문에 그 상위 비트는 소실됨) 가능한 방법이다. 0과의 OR 연산은 본질적으로 NOOP[09] 비트 연산과 같다.

(다음 절에서 배울) NaN, Infinity 등 일부 특수 값은 "32비트에서 안전하지 않다". 이들을 비트 연산하면 ToInt32(4장 강제변환 참고) 추상 연산을 통해 비트 연산 본연의 기능을 수행하고 결과는 +0이 된다.

2.4 특수 값

타입별로 자바스크립트 개발자들이 조심해서 사용해야 할 특수한 값들이 있다.

2.4.1 값 아닌 값

Undefined 타입의 값은 undefined밖에 없다. null 타입도 값은 null뿐이다. 그래서 이 둘은 타입과 값이 항상 같다.

undefined와 null은 종종 "빈empty" 값과 "값 아닌nonvalue" 값을 나타낸다. 이와 다른 의미로 사용하는 개발자도 있다. 예를 들면,

09 역자주_ NOP 또는 NOOP는 어셈블리 언어의 명령어 중 하나로, 명령 자체의 길이만큼 프로그램 카운터를 증가시킬 뿐 아무런 실행도 하지 않습니다. 0과의 OR 연산 역시 값은 변하지 않으므로 이에 비유한 것입니다.

- null은 빈 값이다.
- undefined는 실종된[missing] 값이다.

또는,

- null은 예전에 값이 있었지만 지금은 없는 상태다.
- undefined는 값을 아직 가지지 않은 것이다.

undefined와 null의 의미를 어떻게 "정의"하여 쓰든지, null은 식별자[identifier]가 아닌 특별한 키워드이므로 null이라는 변수에 뭔가 할당할 수는 없다(뭐 하러 이런 짓을!?). 오 이런! 그런데 (불행히도) undefined은 식별자로 쓸 수 있다.

2.4.2 Undefined

느슨한 모드에서는 전역 스코프에서 undefined란 식별자에 값을 할당할 수 있다 (절대 추천하지는 않는다!).

```
function foo() {
undefined = 2; // 정말 좋은 생각이 아니다!
}

foo();

function foo() {
  "use strict";
  undefined = 2; // 타입 에러 발생!
}

foo();
```

그런데 모드에 상관없이 undefined란 이름을 가진 지역 변수는 생성할 수 있다. 가능하기는 하지만 생각만 해도 끔찍하다!

```
function foo() {
  "use strict";
  var undefined = 2;
```

```
  console.log( undefined ); // 2
}

foo();
```

좋은 친구라면 undefined를 여러분이 재정의override하도록 내버려두지 않을 것이다. 결단코!

void 연산자

undefined는 내장 식별자로, (앞의 예처럼 수정하지만 않으면) 값은 undefined지만, 이 값은 void 연산자로도 얻을 수 있다.

표현식 `void` ——는 어떤 값이든 "무효로 만들어void", 항상 결과값을 undefined로 만든다. 기존 값은 건드리지 않고 연산 후 값은 복구할 수 없다.

```
var a = 42;

console.log( void a, a ); // undefined 42
```

(명확하게 `void true`라고 하든가, 동일한 기능의 다른 void 표현이 있는데도, 대개 C 언어 프로그래밍에서 유래된) 관례에 따라 void만으로 undefined 값을 나타내려면 `void 0`이라고 쓴다. `void 0`, `void 1`, undefined 모두 같다.

void 연산자는 어떤 표현식의 결과값이 없다는 걸 확실히 밝혀야 할 때 긴요하다. 예를 들어,

```
function doSomething() {
// 참고: 'APP.ready'는 이 애플리케이션에서 제공한 값이다.
if (!APP.ready) {
// 나중에 다시 해보자!
return void setTimeout( doSomething,100 );
}
```

```
var result;

// 별도 처리 수행
return result;
}

// 제대로 처리했나?
if (doSomething()) {
// 다음 작업 바로 실행
}
```

`setTimeout( .. )` 함수는 숫자 값(타이머를 취소할 때 사용할 타이머의 고유 식별자)을 반환하지만, 예제에서는 이 숫자 값을 무효로 만들어 `doSomething( )` 함수의 결과값이 `if` 문에서 긍정 오류false positive[10]를 일으키지 않게 하려는 것이다.

이때 다음 코드처럼 두 줄로 분리해 쓰는 걸 선호하는 개발자들이 많고 `void` 연산자는 잘 쓰지 않는다.

```
if (!APP.ready) {
// 나중에 다시 해보자!
setTimeout( doSomething,100 );
return;
}
```

정리하면 `void` 연산자는 (어떤 표현식으로부터) 값이 존재하는 곳에서 그 값이 `undefined`가 되어야 좋을 경우에만 사용하자. 아마도 그렇게 해야 할 경우도 거의 없고 극히 제한적으로 쓰이겠지만, 제법 쓸모는 있다.

10 역자주_ 정상을 비정상으로 판단하는 것을 긍정 오류(false positive), 반대로 비정상을 정상이라고 판단하는 것을 부정 오류(false negative)라고 합니다.

2.4.3 특수 숫자

숫자 타입에는 몇 가지 특수한 값이 있다. 하나씩 자세히 알아보자.

The not number, number not number, number

수학 연산 시 두 피연산자가 전부 숫자 (또는 평범한 숫자로 해석 가능한 10진수 또는 16진수)가 아닐 경우 유효한 숫자가 나올 수 없으므로 그 결과는 NaN이다.

NaN은 글자 그대로 "숫자 아님not a number"이다. 그런데 이 명칭과 설명이 아주 형편없고 오해의 소지가 다분하다. NaN은 "숫자 아님"보다는 "유효하지 않은invalid 숫자", "실패한failed 숫자", 또는 "몹쓸 숫자"라고 하는 게 차라리 더 정확하다.

```
var a = 2 / "foo"; // NaN

typeof a === "number"; // true
```

즉, "숫자 아님의 typeof는 숫자다!"란 뜻이다. 이 무슨 해괴망측한 이름/의미란 말인가!

NaN은 경계 값sentinel value의 일종으로 (또는 특별한 의미를 부여한 평범한 값으로) 숫자 집합 내에서 특별한 종류의 에러 상황("난 당신이 내준 수학 연산을 해봤지만 실패했어, 그러니 여기 실패한 숫자를 도로 가져가!")을 나타낸다.

어떤 변숫값이 특수한 실패 숫자, 즉 NaN인지 여부를 확인할 때 null, undefined처럼 NaN도 직접 비교하고 싶은 충동이 생기겠지만... 땡, 틀렸다!

```
var a = 2 / "foo";

a == NaN; // false
a === NaN; // false
```

NaN은 너무 귀하신 몸이라 다른 어떤 NaN과도 동등하지 않다(즉, 자기 자신과도 같지

않다). 사실상 반사성reflexive이[11] 없는(x === x로 식별되지 않는) 유일무이한 값이다. 따라서 NaN !== NaN이다. 좀 이상하긴 하다.

비교 불능이라면(비교 결과가 반드시 실패라면) 그럼 NaN 여부는 어떻게 확인할 수 있을까?

```
var a = 2 / "foo";
isNaN( a ); // true
```

간단하다. 내장 전역 유틸리티 isNaN(..) 함수가 NaN 여부를 말해준다. 문제 해결!

잠깐, 아직이오!

isNaN(..)는 치명적인 결함이 있다. 이 함수는 NaN("숫자 아님")의 의미를 너무 글자 그대로만 해석해서 실제로 "인자 값이 숫자인지 여부를 평가"하는 기능이 전부다. 하지만 이래서는 결과가 정확할 수 없다.

```
var a = 2 / "foo";
var b = "foo";

a; // NaN
b; // "foo"

window.isNaN( a ); // true
window.isNaN( b ); // true  -  허걱!
```

" foo "는 당연히 숫자가 아니지만, 그렇다고 NaN는 아니다! 이 버그는 자바스크립트 탄생 이후 (19년이 넘도록, 이런!) 오늘까지 계속됐다.

드디어 ES6부터는 해결사 Number.isNaN(..)이 등장한다. ES6 이전 브라우저

11 역자주_ 수학에서 어떤 집합 S에 대해 관계 ~이 S에 속한 모든 원소 x에 대해 성립할 때 반사적(reflexive)이라고 합니다. 즉, ∀x∈S: x~x입니다.

에서는 다음 폴리필을 쓰면 안전하게 NaN 여부를 체크할 수 있다.

```
if (!Number.isNaN) {
Number.isNaN = function(n) {
return (
typeof n === "number" &&
window.isNaN( n )
);
};
}

var a = 2 / "foo";
var b = "foo";

Number.isNaN( a ); // true
Number.isNaN( b ); // false - 휴, 다행이다!
```

NaN이 자기 자신과도 동등하지 않는 독특함을 응용하여 폴리필을 더 간단히 구현할 수도 있다. NaN은 세상의 모든 언어를 통틀어 "자기가 아닌 다른 어떤 값도 항상 자신과 동등한" 유일한 값이다.

```
if (!Number.isNaN) {
Number.isNaN = function(n) {
return n !== n;
};
}
```

되게 이상해 보이지만 잘 작동한다!

실제로 많은 자바스크립트 코드에 NaN은 고의로/실수로 박혀있다. 의미를 오해하지 않고 바르게 쓰려면 Number.isNaN(..) 같은 (또는 폴리필) 내장 유틸리티를 사용하자.

만약 지금 여러분이 isNaN(..)를 사용하여 코딩 중이라면, 유감스럽게도 아직

터지지 않은 지뢰(버그)를 묻어놓은 셈이다!

무한대

C와 같은 전통적인 컴파일 언어의 개발자들은 "0으로 나누기divide by zero" 비슷한 컴파일/런타임 에러를 숱하게 보았을 것이다.

```
var a = 1 / 0;
```

그러나 자바스크립트에서는 0으로 나누기 연산이 잘 정의되어 있어서 에러 없이 Infinity(Number.POSITIVE_INFINITY)라는 결과값이 나온다.

```
var a = 1 / 0; // Infinity
var b = -1 / 0; // -Infinity
```

분자가 음수면 0으로 나누기 결과값은 -Infinity (Number.NEGATIVE_INFINITY)다.

자바스크립트는 유한 숫자 표현식finite numeric representations(앞에서 언급한 IEEE 754 부동 소수점)을 사용하므로 수학 교과서와는 다르게 더하기, 뺄셈 같은 연산 결과가 +무한대/-무한대가 될 수 있다. 예를 들면,

```
var a = Number.MAX_VALUE; // 1.7976931348623157e+308
a + a; // 무한대
a + Math.pow( 2, 970 ); // 무한대
a + Math.pow( 2, 969 ); // 1.7976931348623157e+308
```

IEEE 754 명세에 따르면, 덧셈 등의 연산 결과가 너무 커서 표현하기 곤란할 때 "가장 가까운 수로 반올림round-to-nearest" 모드가 결과값을 정한다. 대략적으로 이야기하면 Number.MAX_VALUE + Math.pow(2, 969)는 무한대보다는

Number.MAX_VALUE에 가깝기 때문에 "버림round-down" 처리하고 Number.MAX_VALUE + Math.pow(2, 970)는 무한대에 더 가깝기 때문에 "올림round-up" 처리한다.

너무 골똘히 생각하다간 두뇌에 상처가 날 수도 있으니, 여기서 그만! 제발 그만하자!

그런데 일단 어느 한쪽 무한대의 늪에 빠지고 나면 다신 돌아올 수 없다. 시적 감수성을 살려 표현하면 "님은 유한에서 무한으로 가시지만 무한에서 유한으로 돌아오진 않으시네!".

"무한을 무한으로 나누면?" 이거 너무 철학적인가? 단순히 생각하면 "1"이나 "무한대"가 될 것 같지만 모두 틀렸다. 수학책, 자바스크립트 공히 무한대/무한대는 "정의되지 않은 연산"이며, 결과값은 NaN이다.

유한한 양수를 무한대로 나누면? 당연 0, 이건 쉽다! 유한한 음수를 무한대로 나누면? 자, 채널 고정!

영(0)

철저한 수학 마인드로 무장한 독자라면 자바스크립트엔 보통의 영(+0이라고 함)과 음의 영(-0)이 있다는 사실 자체가 자못 혼란스러울 것이다. -0의 존재 이유를 설명하기 전에 자바스크립트가 영을 다루는 방식을 먼저 짚어보자.

음의 영은 표기만 -0으로 하는 것이 아니다. 특정 수식의 연산 결과 또한 -0으로 떨어진다. 예를 들면,

```
var a = 0 / -3; // -0
var b = 0 * -3; // -0
```

덧셈과 뺄셈에는 -0이 나올 일이 없다.

개발자 콘솔 창에서 확인해보면 -0으로 나오겠지만, 비교적 최근까지 자주 나오

는 연산은 아니어서 아직도 0으로 표시되는 브라우저도 간혹 있을 것이다.[12]

그러나 명세에 의하면 -0을 문자열화stringify하면 항상 "0"이다.

```
var a = 0 / -3;

// (일부 브라우저에 한하여) 제대로 표시한다.
a; // -0

// 하지만 명세는 여러분에게 거짓말을 하라고 시킨다!
a.toString(); // "0"
a + ""; // "0"
String( a ); // "0"

// 이상하게도 JSON조차 속아 넘어간다.
JSON.stringify( a ); // "0"
```

신기하게도 반대로 하면(문자열에서 숫자로 바꾸면) 있는 그대로 보여준다.

```
+"-0"; // -0
Number( "-0" ); // -0
JSON.parse( "-0" ); // -0
```

JSON.parse("-0")는 예상대로 -0인데 JSON.stringify(-0) = "0"은 정말 앞뒤가 맞지 않는 대목이다.

-0을 문자열화할 때 진짜 값을 감춰버리는 것 말고도 비교 연산 결과 역시 (고의로) 거짓말을 한다.

```
var a = 0;
var b = 0 / -3;
```

12 역자주_ 역자가 확인한 결과, IE는 버전에 상관없이 모두 0으로만 콘솔 창에 표시됩니다.

```
a == b; // true
-0 == 0; // true

a === b; // true
-0 === 0; // true

0 > -0; // false
a > b; // false
```

확실하게 -0과 0을 구분하고 싶다면 콘솔 창 결과에만 의존할 게 아니라 조금 더 영리해야 한다.

```
function isNegZero(n) {
n = Number( n );
return (n === 0) && (1 / n === -Infinity);
}

isNegZero( -0 ); // true
isNegZero( 0 / -3 ); // true
isNegZero( 0 ); // false
```

그런데 무슨 수학 경시 대회도 아니고 대관절 -0은 왜 만든 것일까?

값의 크기로 어떤 정보(예: 애니메이션 프레임당 넘김 속도)와 그 값의 부호로 또 다른 정보(예: 넘김 방향)를 동시에 나타내야 하는 애플리케이션이 있기 때문이다.

+0, -0 개념이 없다면 어떤 변숫값이 0에 도달하여 부호가 바뀌는 순간, 그 직전까지 이 변수의 이동 방향은 무엇인지 알 수가 없으므로 부호가 다른 두 0은 유용하다. 즉, 잠재적인 정보 소실을 방지하기 위해 0의 부호를 보존한 셈이다.

2.4.4 특이한 동등 비교

앞서 설명했듯이 NaN과 -0의 동등 비교는 독특하다. NaN은 자기 자신과도 동등하

지 않아 ES6의 `Number.isNaN(..)` 또는 폴리필을 사용해야 하며, 마찬가지로 -0도 거짓말쟁이라서 보통의 0과 동등한 척하므로(심지어는 엄격 동등 연산자 ===에 대해서도 - 4장 강제변환 참고), 방금 전 살펴본 `isNegZero` 같은 함수를 꼼수로 써야 한다.

ES6부터는 잡다한 예외를 걱정하지 않아도 두 값이 절대적으로 동등한지를 확인하는 새로운 유틸리티를 지원한다. 바로 `Object.is(..)`다.

```
var a = 2 / "foo";
var b = -3 * 0;

Object.is( a, NaN ); // true
Object.is( b, -0 ); // true
Object.is( b, 0 ); // false
```

ES6 이전 환경에서도 `Object.is(..)` 폴리필을 만들어 간단히 쓸 수 있다.

```
if (!Object.is) {
Object.is = function(v1, v2) {
// '-0' 테스트
if (v1 === 0 && v2 === 0) {
return 1 / v1 === 1 / v2;
}
// 'NaN' 테스트
if (v1 !== v1) {
return v2 !== v2;
}
// 기타
return v1 === v2;
};
}
```

==나 ===가 안전하다면 굳이 `Object.is(..)`는 사용하지 않는 편이 좋다(4장 강제변환 참고). 아무래도 기본 연산자가 좀 더 효율이 좋고 일반적이기 때문이다. `Object.is(..)`는 주로 특이한 동등 비교에 쓴다.

2.5 값 vs 레퍼런스

다른 언어에서 값은 사용하는 구문에 따라 값-복사value-copy 또는 레퍼런스-복사reference-copy의 형태로 할당/전달한다.

C++에서는 어떤 함수에 전달한 숫자 인자 값을 그 함수 내에서 수정하려면 `int& myNum` 형태로 함수 파라미터를 선언하고, 호출하는 쪽에서는 변수 `x`를 넘기면 `myNum`은 `x`를 참조한다. 레퍼런스는 포인터의 특수한 형태로 다른 변수의 포인터를 (꼭 별명alias처럼) 가진다. 레퍼런스 파라미터를 선언하지 않으면 전달한 값은 아무리 복잡한 객체일지라도 언제나 복사된다.

자바스크립트는 포인터라는 개념 자체가 없고 참조하는 방법도 조금 다르다. 우선 어떤 변수가 다른 변수를 참조할 수 없다. 그냥 안 된다.

자바스크립트에서 레퍼런스는 (공유된) 값을 가리키므로 서로 다른 10개의 레퍼런스가 있다면 이들은 저마다 항상 공유된 단일 값(서로에 대한 레퍼런스/포인터 따위는 없다)을 개별적으로 참조한다.

더구나 자바스크립트에는 값 또는 레퍼런스의 할당 및 전달을 제어하는 구문 암시syntactic hint가 전혀 없다. 대신, 값의 타입만으로 값-복사, 레퍼런스-복사 둘 중 한 쪽이 결정된다.

예를 들면,

```
var a = 2;
var b = a; // 'b'는 언제나 'a'에서 값을 복사한다.
b++;
a; // 2
b; // 3

var c = [1,2,3];
var d = c; // 'd'는 공유된 '[1,2,3]'값의 레퍼런스다.
d.push( 4 );
c; // [1,2,3,4]
d; // [1,2,3,4]
```

null, undefined, string, number, boolean, ES6 symbol 같은 단순 값(스칼라 원시 값scalar primitives)은 언제나 값-복사 방식으로 할당/전달된다.

객체(배열과 박싱된 객체 래퍼 전체 - 3장 네이티브 참고)나 함수 등 합성 값compound values은 할당/전달 시 반드시 레퍼런스 사본을 생성한다.

예제 코드에서 2는 스칼라 원시 값이므로 a엔 이 값의 초기 사본이 들어가고, b에는 또 다른 사본이 자리를 잡는다. 따라서 b를 바꿈으로써 a까지 동시에 값을 변경할 방법은 없다.

하지만, c와 d는 모두 합성 값이자 동일한 공유 값 [1,2,3]에 대한 개별 레퍼런스다. 여기서 기억해야 할 점은 c와 d가 [1,2,3]을 "소유"하는 것이 아니라 단지 이 값을 동등하게 참조만 한다는 사실이다. 따라서 레퍼런스로 실제 공유한 배열 값이 변경되면(.push(4)), 이 공유 값 한 군데에만 영향을 미치므로 두 레퍼런스는 갱신된 값 [1,2,3,4]를 동시에 바라보게 된다.

레퍼런스는 변수가 아닌 값 자체를 가리키므로 A 레퍼런스로 B 레퍼런스가 가리키는 대상을 변경할 수는 없다.

```
var a = [1,2,3];
var b = a;
a; // [1,2,3]
b; // [1,2,3]

// 그 후
b = [4,5,6];
a; // [1,2,3]
b; // [4,5,6]
```

b = [4,5,6]으로 할당해도 a가 참조하는 [1,2,3]은 영향을 받지 않는다. 그렇게 되려면 b가 배열을 가리키는 레퍼런스가 아닌 포인터가 되어야 하는데, 다시 말하지만 자바스크립트에 포인터란 없다!

함수 파라미터 역시 가장 자주 헷갈리는 부분이다.

```
function foo(x) {
x.push( 4 );
x; // [1,2,3,4]

// 그 후
x = [4,5,6];
x.push( 7 );
x; // [4,5,6,7]
}

var a = [1,2,3];

foo( a );

a; // [4,5,6,7]가 아닌 [1,2,3,4]
```

a를 인자로 넘기면 a의 레퍼런스 사본이 x에 할당된다. x와 a는 모두 동일한 [1,2,3] 값을 가리키는 별도의 레퍼런스다. 이제 함수 내부에서 이 레퍼런스를 이용하여 값 자체를 변경한다(push(4)). 하지만 그 후 x = [4,5,6]으로 새 값을 할당해도 초기 레퍼런스 a가 참조하고 있던 값에는 아무런 영향이 없다. 즉, a 레퍼런스는 여전히 (지금은 바뀐) [1,2,3,4] 값을 바라보고 있다.

레퍼런스 x로 a가 가리키고 있는 값을 바꿀 도리는 없다. 다만 a와 x 둘 다 가리키는 공유 값의 내용만 바꿀 수 있다.

배열을 새로 생성하여 할당하는 식으로는 a의 내용을 [4,5,6,7]로 바꿀 수 없다. 기존에 존재하는 배열 값만 변경해야 한다.

```
function foo(x) {
x.push( 4 );
x; // [1,2,3,4]

// 그 후
```

```
x.length = 0; // 기존 배열을 즉시 비운다
x.push( 4, 5, 6, 7 );
x; // [4,5,6,7]
}

var a = [1,2,3];

foo( a );

a; // [1,2,3,4]가 아닌 [4,5,6,7]
```

짐작했겠지만 `x.length = 0`, `x.push(4,5,6,7)`는 새 배열을 생성하는 코드가 아니라, 이미 두 변수가 공유한 배열을 변경하는 코드이므로 a는 새로운 값 [4,5,6,7]을 가리킨다.

값-복사냐 레퍼런스-복사냐를 여러분 마음대로 결정할 수 없음을 기억하자. 전적으로 값의 타입을 보고 엔진의 재량으로 결정된다.

(배열 같은) 합성 값을 값-복사에 의해 효과적으로 전달하려면 손수 값의 사본을 만들어 전달한 레퍼런스가 원본을 가리키지 않게 하면 된다. 예를 들어,

```
foo( a.slice() );
```

인자 없이 slice(..)를 호출하면 전혀 새로운 배열의 (얕은 복사shallow copy에 의한) 사본을 만든다. 이렇게 복사한 사본만을 가리키는 레퍼런스를 전달하니 foo(..)는 a의 내용을 건드릴 수 없다.

반대로 스칼라 원시 값을 레퍼런스처럼 바뀐 값이 바로바로 반영되도록 넘기려면 원시 값을 다른 합성 값(객체, 배열 등)으로 감싸야 한다.

```
function foo(wrapper) {
wrapper.a = 42;
}
```

```
var obj = {
a: 2
};

foo( obj );

obj.a; // 42
```

obj는 스칼라 원시 프로퍼티 a를 감싼 래퍼로 foo(..) 함수엔 obj 레퍼런스 사본이 전달되고 래퍼 파라미터의 값을 바꾼다. 이제 래퍼 레퍼런스로 공유된 객체에 접근하여 프로퍼티를 수정할 수 있다. 함수가 종료되면 obj.a는 수정된 값, 42다.

같은 원리로 2와 같은 스칼라 원시 값을 레퍼런스 형태로 넘기려면 Number 객체 래퍼로 원시 값을 박싱하면 된다(3장 네이티브 참고).

Number 객체의 레퍼런스 사본이 함수에 전달되는 것은 맞지만, 아쉽게도 여러분의 예상대로 공유된 객체를 가리키는 레퍼런스가 있다고 자동으로 공유된 원시 값을 변경할 권한이 주어지는 것이 아니다.

```
function foo(x) {
x = x + 1;
x; // 3
}

var a = 2;
var b = new Number( a ); // 'Object(a)'도 같은 표현이다.

foo( b );
console.log( b ); // 3이 아닌 2
```

문제는 내부의 스칼라 원시 값이 불변이란 점이다(문자열, 불리언도 마찬가지다). 스칼라 원시 값 2를 가진 Number 객체가 있다면, 이와 동일한 객체가 다른 원시 값을 가지도록 변경할 수 없다. 단지 다른 값을 넣은, 완전히 별개의 Number 객체를 생

성할 수는 있다.

표현식 x + 1에서 x가 사용될 때, 내부에 간직된 스칼라 원시 값 2는 Number 객체에서 자동 언박싱(추출)되므로 x = x + 1 문에서 x는 공유된 레퍼런스에서 Number 객체로 아주 교묘하게 뒤바뀌고 2 + 1 덧셈 결과인 스칼라 원시 값 3을 갖게 된다. 따라서 바깥의 b는 원시 값 2를 쓰운, 변경되지 않은/불변의 원본 Number 객체를 참조한다.

Number 객체에 (내부 원시 값 변경이 아니라) 프로퍼티를 추가하고, 간접적이나마 이 추가된 프로퍼티를 통하여 정보를 교환할 수는 있다.

그러나 별로 일반적이지도 않을뿐더러 많은 개발자가 좋은 습관이라고 생각하지 않는다.

이렇게 객체 래퍼 Number를 사용하기보단 차라리 처음부터 손수 객체 래퍼(obj)를 쓰는 편이 훨씬 낫다. 그렇다고 Number처럼 박싱된 객체 래퍼를 적절하게 잘 쓰는 것이 애당초 가능하지 않다는 말은 아니지만, 대부분의 경우 스칼라 원시 값을 사용하는 것이 좋다.

레퍼런스는 꽤 강력하지만 이따금 걸림돌이 되기도 하고 심지어 존재하지도 않는 레퍼런스를 찾아 정처 없이 헤매기도 한다. 값-복사냐 레퍼런스-복사냐를 결정하는 유일한 단서는 값의 타입뿐이므로 사용할 값 타입을 잘 정해서 간접적으로 할당/전달 로직에 반영해야 한다.

2.6 정리하기

자바스크립트 배열은 모든 타입의 값들을 숫자로 인덱싱한 집합이다. 문자열은 일종의 "유사 배열"이지만, 나름 특성이 있기 때문에 배열로 다루고자 할 때에는 조심하는 것이 좋다. 자바스크립트 숫자는 "정수"와 "부동 소수점 숫자" 모두 포함한다.

원시 타입에는 몇몇 특수 값이 있다.

null 타입은 null이란 값 하나뿐이고, 마찬가지로 undefined 타입도 값은 undefined뿐이다. undefined는 할당된 값이 없다면 모든 변수/프로퍼티의 디폴트 값이다. void 연산자는 어떤 값이라도 undefined로 만들어 버린다.

숫자에는 NaN(설명은 "숫자 아님"이지만, 사실 "유효하지 않은 숫자"라고 해야 더 정확함), +Infinity, -Infinity, -0 같은 특수 값이 있다.

단순 스칼라 원시 값(문자열, 숫자 등)은 값-복사에 의해, 합성 값(객체 등)은 레퍼런스-복사에 의해 값이 할당/전달된다. 자바스크립트에서의 레퍼런스는 다른 언어의 레퍼런스/포인터와는 전혀 다른 개념이며, 또 다른 변수/레퍼런스가 아닌, 오직 자신의 값만을 가리킨다.

chapter 3

네이티브

1, 2장에서 보통 "네이티브natives"[01]라고 하는 여러 가지 내장 타입을 몇 차례 넌지시 얘기했다. 이번 장에서 자세히 알아보자.

다음은 가장 많이 쓰는 네이티브들이다.

- `String( )`
- `Number( )`
- `Boolean( )`
- `Array( )`
- `Object( )`
- `Function( )`
- `RegExp( )`
- `Date( )`
- `Error( )`
- `Symbol( )` - ES6에서 추가됨!

짐작하겠지만 네이티브는 사실 내장 함수다.

자바를 아는 독자라면 자바스크립트의 `String( )`이 문자열 값을 생성하는

01 역자주_ 네이티브(native)란 말은 자바스크립트 계에서 여기저기 남용되는 경향이 있습니다. 영어 단어의 의미(원주민, 원래 있던 것)를 상기하여 기본적인 정의를 내린다면, 네이티브란 특정 환경(브라우저 등의 클라이언트 프로그램)에 종속되지 않은, ECMAScript 명세의 내장 객체를 말합니다. 예를 들어, `Object`, `Math`, `Function`, `Array`, `Window`, `Button` 중에 네이티브가 아닌 것은 `Window`, `Button` 두 가지입니다.

String(..) 생성자constructor와 비슷하게 보여서 다음 코드처럼 쓸 수 있음을 금세 눈치챘을 것이다.

```
var s = new String( "Hello World!" );

console.log( s.toString() ); // "Hello World!"
```

네이티브는 생성자처럼 사용할 수 있지만 실제로 생성되는 결과물은 여러분의 예상과 다를 수 있다.

```
var a = new String( "abc" );

typeof a; // "object" ... "String"이 아니다!

a instanceof String; // true

Object.prototype.toString.call( a ); // "[object String]"
```

(new String("abc")) 생성자의 결과는 원시 값 "abc"를 감싼 객체 래퍼다.

놀랍게도 typeof 연산자로 이 객체의 타입을 확인해보면 자신이 감싼 원시 값의 타입이 아닌 object의 하위 타입에 가깝다.

객체 래퍼가 어떻게 생겼는지 들여다보자.

```
console.log( a );
```

이 코드의 실행 결과는 브라우저마다 다르다. 개발자 콘솔 창에 어떻게 객체를 직렬화하여 보여주는 편이 좋을지는 브라우저 개발자가 임의로 결정했기 때문이다.

이 글을 쓰는 현재 크롬 최신 버전은 String {0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"}, 구 버전 크롬에선 String {0: "a", 1: "b", 2:"c"}으로 나온다. 파이어폭스 최신 버전에선 일단 String ["a","b","c"]으로 표시되고, 이탤릭체로 표기된 "abc"를 클릭하면 객체 안을 들여다볼 수 있다. 물론, 매일 같이 계속 업데이트가 되고 있는 중이니 여러분의 PC에선 다르게 보일 수 있다.

요는 new String("abc")은 "abc"를 감싸는 문자열 래퍼를 생성하며 원시 값 "abc"는 아니라는 점이다.

3.1 내부 [[Class]]

typeof가 "object"인 값(배열 등)에는 [[Class]]라는 내부 프로퍼티(전통적인 클래스 지향class-oriented 개념에서의 클래스라기보단 내부 분류법classification의 일부라고 보자)가 추가로 붙는다. 이 프로퍼티는 직접 접근할 수 없고 Object.prototype.toString(..)라는 메서드에 값을 넣어 호출함으로써 존재를 엿볼 수 있다. 예를 들면,

```
Object.prototype.toString.call( [1,2,3] );
// "[object Array]"

Object.prototype.toString.call( /regex-literal/i );
// "[object RegExp]"
```

내부 [[Class]] 값이, 배열은 "Array", 정규식은 "RegExp"임을 알 수 있다. 대부분 내부 [[Class]]는 해당 값과 관련된 내장 네이티브 생성자(다음에 설명한다)를 가리키지만, 그렇지 않을 때도 있다.

원시 값에도 내부 [[Class]]가 있을까? 먼저 null, undefined를 보자.

```
Object.prototype.toString.call( null );
// "[object Null]"

Object.prototype.toString.call( undefined );
// "[object Undefined]"
```

주지하다시피 `Null()`, `Undefined()` 같은 네이티브 생성자는 없지만 내부 `[[Class]]` 값을 확인해보니 "`Null`", "`Undefined`"이다.

하지만 그 밖의 문자열, 숫자, 불리언 같은 단순 원시 값은, 이른바 "박싱boxing" 과정을 거친다.

```
Object.prototype.toString.call( "abc" );
// "[object String]"

Object.prototype.toString.call( 42 );
// "[object Number]"

Object.prototype.toString.call( true );
// "[object Boolean]"
```

내부 `[[Class]]` 값이 각각 "`String`", "`Number`", "`Boolean`"으로 표시된 것으로 보아 단순 원시 값은 해당 객체 래퍼로 자동 박싱됨을 알 수 있다.

여기서 `toString()`과 `[[Class]]`는 ES5 → ES6 과정에서 작동 방식이 조금 바뀌었는데, 자세한 내용은 『You Don't Know JS: ES6 & Beyond』[02]를 참고하자.

3.2 래퍼 박싱하기

객체 래퍼는 아주 중요한 용도로 쓰인다. 원시 값엔 프로퍼티나 메서드가 없으므

02 http://goo.gl/GqxITA

로 .length, .toString( )으로 접근하려면 원시 값을 객체 래퍼로 감싸줘야 한다. 고맙게도 자바스크립트는 원시 값을 알아서 박싱(래핑)하므로 다음과 같은 코드가 가능하다.

```
var a = "abc";

a.length; // 3
a.toUpperCase(); // "ABC"
```

따라서 루프 조건 i < a.length처럼 빈번하게 문자열 값의 프로퍼티/메서드를 사용해야 한다면 자바스크립트 엔진이 암시적으로 객체를 생성할 필요가 없도록 처음부터 값을 객체로 갖고 있는 것이 이치에 맞는 것처럼 보인다.

하지만 좋은 생각이 아니다. 오래 전부터 브라우저는 이런 흔한 경우를 스스로 최적화하기 때문이다. 즉, 개발자가 직접 객체 형태로 (최적화되지 않은 방향으로) "선 최적화pre-optimize"하면 프로그램이 더 느려질 수 있다.

직접 객체 형태로 써야 할 이유는 거의 없다. 필요 시 엔진이 알아서 암시적으로 박싱하게 하는 것이 낫다. 즉, new String(" abc "), new Number(42)처럼 코딩하지 말고, 그냥 알기 쉽게 원시 값 " abc ", 42를 사용하자.

3.2.1 객체 래퍼의 함정

그래도 객체 래퍼를 사용해야 한다면 정말 조심해야 할 함정이 있으니 잘 보기 바란다.

예를 들어, Boolean으로 래핑한 값이 있다.

```
var a = new Boolean( false );

if (!a) {
console.log( "Oops" ); // 실행되지 않는다.
}
```

false를 객체 래퍼로 감쌌지만 문제는 객체가 " truthy "(4장 강제변환 참고)란 점이다. 그래서 예상과는 달리, 안에 들어있는 false 값과 반대의 결과다.

수동으로 원시 값을 박싱하려면 Object(..) 함수를 이용하자(앞에 new 키워드는 없다).

```
var a = "abc";
var b = new String( a );
var c = Object( a );

typeof a; // "string"
typeof b; // "object"
typeof c; // "object"

b instanceof String; // true
c instanceof String; // true

Object.prototype.toString.call( b ); // "[object String]"
Object.prototype.toString.call( c ); // "[object String]"
```

다시 한 번 말하지만 객체 래퍼로 직접 박싱하는 건 권하고 싶지 않다. 물론, 드물긴 하지만 그렇게 해야 할 경우가 전혀 없진 않다.

3.3 언박싱

객체 래퍼의 원시 값은 valueOf( ) 메서드로 추출한다.

```
var a = new String( "abc" );
var b = new Number( 42 );
var c = new Boolean( true );

a.valueOf(); // "abc"
b.valueOf(); // 42
c.valueOf(); // true
```

이때에도 암시적인 언박싱이 일어난다. 이 과정(강제변환)은 4장 강제변환에서 다시 자세히 다루지만, 간단히 다음 예제를 보자.

```
var a = new String( "abc" );
var b = a + ""; // 'b'에는 언박싱된 원시 값 "abc"이 대입된다.

typeof a; // "object"
typeof b; // "string"
```

3.4 네이티브, 나는 생성자다

배열, 객체, 함수, 정규식 값은 리터럴 형태로 생성하는 것이 일반적이지만, 리터럴은 생성자 형식으로 만든 것과 동일한 종류의 객체를 생성한다(즉, 래핑되지 않은 값은 없다).

앞에서 살펴본 다른 네이티브도 그랬듯이, 확실히 필요해서 쓰는 게 아니라면 생성자는 가급적 쓰지 않는 편이 좋다. 나중에 별별 오류와 함정에 빠져 고생하고 싶지 않으면 말이다.

3.4.1 Array(..)

```
var a = new Array( 1, 2, 3 );
a; // [1, 2, 3]

var b = [1, 2, 3];
b; // [1, 2, 3]
```

`Array(..)` 생성자 앞에 `new`를 붙이지 않아도 된다. 붙이지 않아도 붙힌 것처럼 작동한다. 즉, `Array(1,2,3)`와 `new Array(1,2,3)`는 결과적으로 같다.

Array 생성자에는 특별한 형식이 하나 있는데 인자로 숫자를 하나만 받으면 그 숫자를 원소로 하는 배열을 생성하는 게 아니라 "배열의 크기를 미리 정하는presize" 기능이다.

생각만 해도 아찔하다. 잊어버리기 쉬운 터라 제대로 알지 못하고 썼다간 큰 사고가 터질 것 같다.

하지만 그보다 배열의 크기를 미리 정한다는 것 자체가 말이 안 된다. 그렇게 하려면 빈 배열을 만들고 나중에 length 프로퍼티에 숫자 값을 할당하는 게 맞다.

실제로 슬롯에 값은 없지만 length만 봐서는 뭔가 값이 있을 것 같은, 그런 불길한 느낌의 배열은 정말 이상하고 혼란스러운, 자바스크립트만의 희한한 자료 구조다. 아직도 이런 배열을 생성할 수 있게 놔둔 건 순전히 이미 오래 전 비 권장된 역사적 유물(arguments 같은 "유사 배열 객체") 탓이다.

"빈 슬롯"을 한 군데 이상 가진 배열을 "구멍 난 배열sparse array"이라고 한다.

브라우저 개발자 콘솔 창마다 객체를 나타내는 방식이 제각각이라 혼란은 가중된다. 예를 들어,

```
var a = new Array( 3 );

a.length; // 3
a;
```

(이 책의 집필 시점에서) 크롬의 직렬화 결과는 [undefined x 3]이다. 슬롯 자체가 존재하지 않아(이른바 "빈 슬롯empty slot"일 때, - 이 이름 역시 탐탁지 않다), 배열 슬롯에 3개의 undefined 값을 밀어넣은 것이다.

다음 코드를 실행해보고 차이점을 확인하자.

```
var a = new Array( 3 );
var b = [ undefined, undefined, undefined ];
var c = [];
c.length = 3;

a;
b;
c;
```

예제에서 c를 잘 보면 알겠지만, 빈 슬롯은 배열 생성 직후 만들어진다. 실제로 정의된 슬롯 개수를 초과하여 length 값을 세팅하면 암시적으로 빈 슬롯이 생긴다. delete b[1] 코드를 실행해도 b 배열 가운데에 빈 슬롯이 만들어진다.

(현재 크롬에서) b는 [undefined, undefined, undefined]로 출력되는 반면, a와 c는 [undefined x 3]라고 표시된다. 헷갈린다고? 여러분만 그런 게 아니다.

설상가상으로 파이어폭스에서 실행하면 a와 c가 [, , ,]로 나온다. 왜 헷갈리는지 알겠는가? 눈을 크게 뜨고 유심히 보기 바란다. 콤마가 3개 있어서 마치 슬롯이 4개인 것처럼 보인다.

엥!? 파이어폭스는 직렬화한 문자열 끝에 콤마를 하나 더 붙이는데, 이는 ES5부터 리스트(배열 값, 프로퍼티 리스트 등)의 후미 콤마trailing comma를 허용(그러고 나서 떨구고 무시함)했기 때문이다. 그래서 여러분의 콘솔 창에 어떤 값이 [, , ,]과 같이 보인다면, 실제로는 [, ,](빈 슬롯이 3개인 배열)다. 비록 콘솔 창에서는 헷갈리지만 카피-앤-페이스트copy-n-paste 할 때는 정확히 작동한다는 점에서 그런대로 봐줄 만

하다.[03]

고개를 내젓고 눈이 휘둥그래지더라도 여러분만 그런 게 아니니 기운내시길!

파이어폭스 최신 버전에서는 `Array [ <3 empty slots> ]`로 콘솔 창에 찍히는데 `[ , , , ]`에 비하면 분명 엄청난 개선이다.

그런데 콘솔 결과가 헷갈리는 건 그렇다 치고 a와 b가 어떨 때는 같은 값처럼 보이다가도 그렇지 않을 때도 있다는 점이 더 나쁘다.

```
// var a = new Array( 3 );
// var b = [ undefined, undefined, undefined ];

a.join( "-" ); // "--"
b.join( "-" ); // "--"

a.map(function(v,i){ return i; }); // [ undefined x 3 ]
b.map(function(v,i){ return i; }); // [ 0, 1, 2 ]
```

오, 이런!

`a.map( .. )`는 a에 슬롯이 없기 때문에 `map( .. )` 함수가 순회할 원소가 없다. `join( .. )`은 다르다. 기본적으로 `join( .. )`의 구현 로직은 다음과 같다.

```
function fakeJoin(arr,connector) {
var str = "";
```

03 역자주_ ES5 §11.1.4, "배열 초기화(Array Initialiser)"에 다음과 같이 구문이 명시되어 있습니다.

```
ArrayLiteral :
[ Elisonopt ]
[ ElementList ]
[ ElementList , Elisonopt ]
```

ES3까지는 세 번째 줄이 빠져 있어서 개발자가 실수로 마지막 콤마(여분 콤마(Extra comma), 후위 콤마(trailing comma))를 붙이면 에러가 났지만, ES5부터는 규정이 관대해진 덕분에 에러 없이 조용히 무시합니다.

```
for (var i = 0; i < arr.length; i++) {
if (i > 0) {
str += connector;

}
if (arr[i] !== undefined) {
str += arr[i];
}
}
return str;
}

var a = new Array( 3 );
fakeJoin( a, "-" ); // "--"
```

보다시피 `join( .. )`은 슬롯이 있다는 가정하에 `length`만큼 루프를 반복한다. `map( .. )` 함수는 내부 로직이야 어떻든 (적어도 겉보기엔) 이런 가정을 하지 않는 까닭에 이상한 "빈 슬롯" 배열이 입력되면 예기치 않은 결과가 빚어지거나 실패의 원인이 된다.

("빈 슬롯" 말고) 진짜 `undefined` 값 원소로 채워진 배열은 (손으로 입력하지 않고) 어떻게 생성할까?

```
var a = Array.apply( null, { length: 3 } );
a; // [ undefined, undefined, undefined ]
```

이게 뭐냐고? 음, 간단히 설명하겠다. `apply( .. )`는 모든 함수에서 사용 가능한 유틸리티로, 특이한 방법으로 함수를 호출한다.

첫 번째 인자 `this`는 객체 바인딩object binding(「You Don't Know JS: this & Object Prototypes」[04] 참고)으로, 일단 여기선 그냥 `null`로 세팅하자. 두 번째 인자는 인자

04 http://goo.gl/oZEHe0

의 배열(또는 배열 비슷한 "유사 배열")로, 이 안에 포함된 원소들이 "펼쳐져spread" 함수의 인자로 전달된다.

따라서 `Array.apply( .. )`는 `Array( .. )` 함수를 호출하는 동시에 `{ length: 3 }` 객체 값을 펼쳐 인자로 넣는다.

`apply( .. )` 내부에서는 아마 `0`에서 `length`(여기서는 3) 직전까지 (앞서 보았던 `join( .. )`처럼) 루프를 순회할 것이다.

인덱스별로 객체에서 키를 가져온다. 만약 함수 내부에서 배열 객체 파라미터를 `arr`라고 명명한다면 프로퍼티는 `arr[0]`, `arr[1]`, `arr[2]`로 접근할 것이다. 물론 이 세 프로퍼티 모두 `{ length: 3 }` 객체에는 존재하지 않기 때문에, 모두 `undefined`를 반환할 것이다.

즉, `Array( .. )`를 호출하면 결국 `Array( undefined,undefined,undefined )`처럼 되어, (말도 안 되는) 빈 슬롯이 아닌, `undefined`로 채워진 배열이 탄생한다.

`undefined`로 채워진 배열을 만들고자 `Array.apply( null, { length: 3 } )` 같은 코드를 쓴다는 말이 어색하기도 하고 다소 장황해 보이지만, `Array( 3 )`의 멍청한 빈 슬롯을 갖고 씨름할 걸 생각하면 차라리 현명하고 미더운 선택이 아닐까 싶다.

여하튼, 절대로, 무슨 일이 있어도, 어떤 경우에서라도, 이런 이국적인 빈 슬롯 배열을 애써 만들어 멋 부리지 말자. 그냥 하지 마라, 바보같은 짓이다.

3.4.2 Object(..), Function(..), and RegExp(..)

일반적으로 `Object( .. )`/`Function( .. )`/`RegExp( .. )` 생성자도 선택 사항이다(어떤 분명한 의도가 아니라면 사용하지 않는 편이 좋다).

```
var c = new Object();
c.foo = "bar";
```

```
c; // { foo: "bar" }

var d = { foo: "bar" };
d; // { foo: "bar" }

var e = new Function( "a", "return a * 2;" );
var f = function(a) { return a * 2; }
function g(a) { return a * 2; }

var h = new RegExp( "^a*b+", "g" );
var i = /^a*b+/g;
```

new Object() 같은 생성자 폼은 사실상 사용할 일이 없다. 리터럴 형태로 한번에 여러 프로퍼티를 지정할 수도 있는데 굳이 한 번에 하나씩 일일이 프로퍼티를 지정해야 하는 방법으로 돌아갈 필요가 있을까?

Function 생성자는 함수의 파라미터나 내용을 동적으로 정의해야 하는, 매우 드문 경우에 한해 유용하다. Function(..)을 eval(..)의 대용품이라고 착각하지 말자. 이렇게 함수를 동적으로 정의할 일은 아마 거의 없을 것이다.

정규 표현식은 리터럴 형식(/^a*b+/g)으로 정의할 것을 적극 권장한다. 구문이 쉽고 무엇보다 성능상 이점(자바스크립트 엔진이 실행 전 정규 표현식을 미리 컴파일한 후 캐시한다)이 있다. 지금까지 살펴본 다른 생성자와는 달리, RegExp(..)는 정규 표현식 패턴을 동적으로 정의할 경우 의미있는 유틸리티다.

```
var name = "Kyle";
var namePattern = new RegExp( "\\b(?:" + name + ")+\\b", "ig" );

var matches = someText.match( namePattern );
```

이런 코드는 자바스크립트 코드에서 수시로 등장하는데 new RegExp("패턴","플래그") 형식으로 사용하자.

3.4.3 Date(..) and Error(..)

네이티브 생성자 Date(..)와 Error(..)는 리터럴 형식이 없으므로 다른 네이티브에 비해 유용하다.

date 객체 값은 new Date()로 생성한다. 이 생성자는 날짜/시각을 인자로 받는다(인자를 생략하면 현재 날짜/시각으로 대신한다).

date 객체는 Unix 타임스탬프 값(1970년 1월 1일부터 현재까지 흐른 시간을 초 단위로 환산)을 얻는 용도로 가장 많이 쓰일 것이다. data 객체의 인스턴스로부터 getTime()를 호출하면 된다.

하지만 ES5에 정의된 정적 도우미 함수helper function, Date.now()를 사용하는 게 더 쉽다. ES5 이전 브라우저에선 다음 폴리필을 쓰자.

```
if (!Date.now) {
Date.now = function(){
return (new Date()).getTime();
};
}
```

new 키워드 없이 Date()를 호출하면 현재 날짜/시각에 해당하는 문자열을 반환한다. 이 문자열의 정확한 형식은 로케일에 따라 달라지는데, 많은 브라우저들이 대략 "Fri Jul 18 2014 00:31:02 GMT-0500 (CDT)"과 같이 표시한다.

Error(..) 생성자는 (Array()와 마찬가지로) 앞에 new가 있든 없든 결과는 같다.

error 객체의 주 용도는 현재의 실행 스택 콘텍스트execution stack context를 포착하여 객체(자바스크립트 엔진 대부분이 읽기 전용 프로퍼티인 .stack으로 접근 가능하다)에 담는 것이다. 이 실행 스택 콘텍스트는 함수 호출 스택, error 객체가 만들어진 줄 번호 등 디버깅에 도움이 될 만한 정보들을 담고 있다.

error 객체는 보통 throw 연산자와 함께 사용한다.

```
function foo(x) {
if (!x) {
throw new Error( "x를 안 주셨어요!" );
}
// ..
}
```

Error 객체 인스턴스에는 적어도 message 프로퍼티는 들어 있고, type 등 다른 프로퍼티(읽기 전용)가 포함되어 있을 때도 있다. 그러나 사람이 읽기 편한 포맷으로 에러 메시지를 보려면 방금 전 언급한 stack 프로퍼티 대신, 그냥 error 객체의 (명시적 또는 암시적인 강제변환 과정을 거쳐 - 4장 강제변환 참고) toString( )을 호출하는 것이 가장 좋다.

굳이 따진다면, 일반적인 Error(..) 네이티브 이외에도 구체적인 에러 타입에 특화된 네이티브들이 있다. EvalError(..), RangeError(..), ReferenceError(..), SyntaxError(..), TypeError(..), URIError(..) 이 네이티브들은 코드에서 실제로 예외가 발생하면 자동으로 던져지므로 직접 사용할 일은 거의 없다(가령, 선언하지 않은 변수 참조 시 ReferenceError 에러를 던진다).

3.4.4 Symbol(..)

"심볼Symbol"은 ES6에서 처음 선보인, 새로운 원시 값 타입이다. 심볼은 충돌 염려 없이 객체 프로퍼티로 사용 가능한, 특별한 "유일 값"이다(절대적으로 유일함이 보장되지는 않는다!). 주로 ES6의 특수한 내장 로직에 쓰기 위해 고안되었지만 여러분도 얼마든지 심볼을 정의할 수 있다.

심볼은 프로퍼티명으로 사용할 수 있으나, 프로그램 코드나 개발자 콘솔 창에서 심볼의 실제 값을 보거나 접근하는 건 불가능하다. 심볼 값을 콘솔 창에 출력해보면 Symbol(Symbol.create)로 나온다.

ES6에는 심볼 몇 개가 미리 정의되어 있는데 Symbol.create, Symbol.

iterator 식으로 Symbol 함수 객체의 정적 프로퍼티로 접근한다. 사용법은 다음과 같다.

```
obj[Symbol.iterator] = function(){ /*..*/ };
```

심볼을 직접 정의하려면 Symbol(..) 네이티브를 사용한다. Symbol(..)은 앞에 new를 붙이면 에러가 나는, 유일한 네이티브 "생성자"다.

```
var mysym = Symbol( "my own symbol" );
mysym; // Symbol(my own symbol)
mysym.toString(); // "Symbol(my own symbol)"
typeof mysym; // "symbol"

var a = { };
a[mysym] = "foobar";

Object.getOwnPropertySymbols( a );
// [ Symbol(my own symbol) ]
```

심볼은 전용private 프로퍼티는 아니지만(Object.getOwnPropertySymbols (..)로 들여다보면 공용public 프로퍼티임을 알 수 있다), 본래의 사용 목적에 맞게 대부분 전용 혹은 특별한 프로퍼티로 사용한다. 지금까지 많은 개발자가 "잠깐, 이건 전용/특수/내부 프로퍼티입니다, 건드리지 마세요!"라고 하고 싶을 때 습관적으로 써 왔던, 언더스코어(_)가 앞에 붙은 프로퍼티명도 언젠가는 심볼에 의해 완전히 대체될 가능성이 높다.

심볼은 객체가 아니다. 단순한 스칼라 원시 값이다.

3.4.5 네이티브 프로토타입

내장 네이티브 생성자는 각자의 `.prototype` 객체를 가진다(예: `Array.prototype`, `String.prototype` 등).

- `prototype` 객체에는 해당 객체의 하위 타입별로 고유한 로직이 담겨 있다.

이를테면, 문자열 원시 값을 (박싱으로) 확장한 것까지 포함하여 모든 `String` 객체는 기본적으로 `String.prototype` 객체에 정의된 메서드에 접근할 수 있다.

문서화 관례에 따라 `String.prototype.XYZ`는 `String#XYZ`로 줄여 쓴다. 다른 `.prototype`도 마찬가지다.

String#indexOf(..)

문자열에서 특정 문자의 위치를 검색

String#charAt(..)

문자열에서 특정 위치의 문자를 반환

String#substr(..), String#substring(..), and String#slice(..)

문자열의 일부를 새로운 문자열로 추출

String#toUpperCase() and String#toLowerCase()

대문자/소문자로 변환된 새로운 문자열 생성

String#trim()

앞/뒤의 공란이 제거된 새로운 문자열 생성

이 중 문자열 값을 변경하는 메서드는 없다. 수정(대소문자 변환이나 트리밍)이 일어나면 늘 기존 값으로부터 새로운 값을 생성한다.

프로토타입 위임prototype delegation 덕분에(『You Don't Know JS: this & Object

Prototypes』[05] 참고) 모든 문자열이 이 메서드들을 같이 쓸 수 있다.

```
var a = " abc ";

a.indexOf( "c" ); // 3
a.toUpperCase(); // " ABC "
a.trim(); // "abc"
```

각 생성자 프로토타입마다 자신의 타입에 적합한 기능이 구현되어 있다. 가령, `Number#toFixed(..)`는 고정 소수점 이하 자릿수까지 끊긴 숫자를 문자열로 바꾸어 반환하고, `Array#concat(..)`는 배열을 병합한다. 모든 함수는 `Function.prototype`에 정의된 `apply(..)`, `call(..)`, `bind(..)` 메서드를 사용할 수 있다.

그러나 모든 네이티브 프로토타입이 다 그렇게 평범한 것만은 아니다.

```
typeof Function.prototype; // "function"
Function.prototype(); // 빈 함수다!

RegExp.prototype.toString(); // "/(?:)/" - 빈 regex
"abc".match( RegExp.prototype ); // [""]
```

이렇게 (여러분이 잘 알고 있는 것처럼 프로퍼티를 추가하지 않고) 네이티브 프로토타입을 변경하는 것도 가능하지만 결코 바람직하지 않은 발상이다.

```
Array.isArray( Array.prototype ); // true
Array.prototype.push( 1, 2, 3 ); // 3
Array.prototype; // [1,2,3]

// 이런 식으로 놔두면 이상하게 작동할 수 있다!
// 다음 코드는 'Array.prototype'을 비워버린다
Array.prototype.length = 0;
```

05 http://goo.gl/oZEHe0

그러고 보니 Function.prototype은 함수, RegExp.prototype은 정규 표현식, Array.prototype은 배열이다. 참 재미지고 쿨하지 않은가?

프로토타입은 디폴트다

변수에 적절한 타입의 값이 할당되지 않은 상태에서 Function.prototype → 빈 함수, RegExp.prototype → "빈(아무것도 매칭하지 않는)" 정규식, Array.prototype → 빈 배열은, 모두 멋진 "디폴트 값"이다. 예를 들면,

```
function isThisCool(vals,fn,rx) {
vals = vals || Array.prototype;
fn = fn || Function.prototype;
rx = rx || RegExp.prototype;

return rx.test(
vals.map( fn ).join( "" )
);
}

isThisCool(); // true

isThisCool(
["a","b","c"],
function(v){ return v.toUpperCase(); },
/D/
); // false
```

ES6부터는 'vals = vals || 디폴트 값' 식의 구문 트릭(4장 강제변환 참고)은 더 이상 필요 없다. 왜냐하면 함수 선언부에서 네이티브 구문을 통해 파라미터의 디폴트 값을 설정할 수 있기 때문이다(5장 문법 참고).

프로토타입으로 디폴트 값을 세팅하면 사소하지만 추가적인 이점이 있다. .prototypes는 이미 생성되어 내장된 상태이므로 단 한 번만 생성된다. 그러나 [], function(){}, /(?:)/를 디폴트 값으로 사용하면 (자바스크립트 엔진 구현에

따라 조금씩 다르지만) `isThisCool( .. )`를 호출할 때마다 디폴트 값을 다시 생성(그리고 나중에 가비지 콜렉팅)하므로 그만큼 메모리/CPU가 낭비된다.

그리고 이후에 변경될 디폴트 값으로 `Array.prototype`를 사용하는 일은 없도록 유의하기 바란다. 앞 예제에서 `vals`는 읽기 전용이지만 `vals` 변수 자체를 수정하면 결국 `Array.prototype` 자체도 수정되어 이미 앞에서 언급했던 함정에 빠지게 될 것이다!

이 책에서 네이티브 프로토타입과 그 유용성을 이야기했지만, 이것에 너무 의존하지는 말자. 특히, 어떤 식으로도 프로토타입을 변경하지 않도록 유의하기 바란다.

3.5 정리하기

자바스크립트는 원시 값을 감싸는 객체 래퍼, 즉 네이티브(`String`, `Number`, `Boolean` 등)를 제공한다. 객체 래퍼에는 타입별로 쓸 만한 기능이 구현되어 있어 편리하게 사용할 수 있다(예: `String#trim( )`, `Array#concat( .. )` 등).

"`abc`" 같은 단순 스칼라 원시 값이 있을 때, 이 값의 `length` 프로퍼티나 `String.prototype`에 정의된 메서드를 호출하면 자바스크립트는 자동으로 원시 값을 "박싱"(해당되는 객체 래퍼로 감싼다)하여 필요한 프로퍼티와 메서드를 쓸 수 있게 도와준다.

chapter 4

강제변환

지금까지 자바스크립트의 타입/값에 대해 충분히 살펴보았으니, 이제 말도 많고 탈도 많은, 강제변환[01] 이야기를 시작하겠다.

1장에서도 말했듯이, 강제변환이 유용한 기능인지, 언어 설계 상 결함인지는 (아니면 그 중간?) 처음부터 큰 논란거리였다. 시중에 출간된 다른 자바스크립트 책에도 '강제변환이 마법이다.', '사악하다.', '헷갈린다.', '그냥 나쁜 생각이다.' 등 안 좋은 말들이 많이 씌어있다.

이 시리즈를 내면서 다짐했다. 다른 사람들이 다 피하려 한다고, 기벽 때문에 며칠 밤을 지새워봤다고, 강제변환에서 도망치라고 하지 않겠노라고. 외려 여러분이 제대로 이해하지 못하고 있거나 그럴 시도조차 하지 않은 미지의 영역으로 직행할 것이다.

이 장의 목적은 여러분이 강제변환의 좋고 나쁨(좋은 점도 있다!)을 충분히 이해하고, 자신의 프로그램에 적절한지 스스로 현명하게 판단할 수 있는 역량을 갖추는 것이다.

01 역자주_ 정확히 말하면 "강제적 타입변환(coercive type conversion)" 또는 "타입 강제변환(type coercion)"이지만, 원문에서도 coercion이라고 간단히 표기하였기에 역서에서도 간단히 "강제변환"으로 옮깁니다. 참고로, ES5를 보면 "자동 타입 변환(automatic type conversion)"이라는 말이 나오는데 동일한 개념입니다. 표준 용어가 정해져 있지 않아 비슷한 용어가 혼용되는 현실은 어쩔 수 없지만 자바스크립트 엔진에 의해 타입이 자동 변환된다는 의미만 명확하게 이해하고 있으면 됩니다.

4.1 값 변환

어떤 값을 다른 타입의 값으로 바꾸는 과정이 명시적이면 "타입 캐스팅type casting", (값이 사용되는 규칙에 따라) 암시적이면 "강제변환coercion"이라고 한다.

항상 그렇지 않을 수도 있는데, 자바스크립트에서 강제변환을 하면 문자열, 숫자, 불리언 같은 스칼라 원시 값(2장 값 참고) 중 하나가 되며, 객체, 함수 등 합성 값 타입으로 변환될 일은 없다. 3장 네이티브에서 나온 "박싱"은 스칼라 원시 값을 해당 객체로 감싸는 건데, 정확히 말하면 박싱은 강제변환이 아니다.

두 용어를 이렇게 구분하는 사람들도 있다. "타입 캐스팅(또는 "타입 변환type conversion")은 정적 타입 언어에서 컴파일 시점에, "강제변환"은 동적 타입 언어에서 런타임 시점에 발생한다.

그러나 자바스크립트에서는 대부분 모든 유형의 타입 변환을 강제변환으로 뭉뚱그려 일컫는 경향이 있어서, 여기서는 "암시적 강제변환explicit coercion"과 "명시적 강제변환implicit coercion", 두 가지로 구별한다.

차이는 명확하다. "명시적 강제변환"은 코드만 봐도 의도적으로 타입 변환을 일으킨다는 사실이 명백한 반면, "암시적 강제변환"은 다른 작업 도중 불분명한 부수 효과side effect로부터 발생하는 타입 변환이다.

다음 코드를 보자.

```
var a = 42;

var b = a + ""; // 암시적 강제변환

var c = String( a ); // 명시적 강제변환
```

b에 암시적 강제변환이 발생한다. 공백 문자열(" ")과의 + 연산은 문자열 접합concatenation(두 문자열을 하나로 합친다) 처리를 의미하므로 (숨겨진) 부수 효과로서 숫자

42를 이와 동등한 문자열 " 42 " 로 강제변환한다.

이와는 대조적으로, String(..) 함수는 값을 인자로 받아 명백히 문자열 타입으로 강제변환한다.

두 가지 접근 방식 모두 42를 " 42 " 로 바꾸는 기능 자체는 같지만, 뜨거운 논란의 핵심은 바로 변환을 "어떻게" 할 것이냐, 하는 문제다.

엄밀히 말해 스타일의 차이뿐만 아니라 작동상 미묘한 차이도 있다.

"명시적explicit" : "암시적implicit" = "명백한obvious" : "숨겨진 부수 효과hidden side effect"

용어상으로는 이러한 대응 관계가 성립한다.

a + " " 의 의미를 정확히 파악하고 의도적으로 문자열 변환을 했다면, 그 자체로 충분히 "명시적"이지 않냐고 생각할 수 있다. 반대로, 한번도 문자열 강제변환 시 String(..) 함수를 사용한 적 없는 사람에겐 이것이야말로 "암시적"이지 않냐고 반문할 수 있다.

하지만 이 책에서 무엇이 "명시적"이고 무엇이 "암시적"이냐 하는 문제는 자바스크립트 마스터, 자바스크립트 명세에 몸바친 추종자들이 아닌, 자바스크립트를 어느 정도 알고 있는 개발자 사이에서 오간 여러 의견들에 근거하여 논할 생각이다. 자신의 의견과 잘 맞지 않을 수도 있지만, 일단 이 책을 읽는 여러분은 이 책의 관점에 맞춰주기 바란다.

한 가지만 기억하자. 여기에 작성한 코드를 나만 보게 될 일은 아주 드물다. 여러분이 자바스크립트를 속속들이 꿰고 있는 자바스크립트 마스터라 해도 경력이 일천한, 같은 팀 개발자가 여러분이 짠 코드를 보고 어떤 기분일지 상상해보자. "명시적", "암시적"... 이런 용어들을 여러분과 똑같은 의미로 받아들일까?

4.2 추상 연산

명시적/암시적 강제변환의 세계로 떠나기 전에 어떻게 값이 문자열, 숫자, 불리언 등의 타입이 되는지, 그 기본 규칙을 알아보자. ES5 §9를 보면 변환 규칙의 "추상 연산abstract operation"("내부 전용 연산internal-only operation"을 명세스럽게 다듬은 용어)이 정의되어 있다. ToString, ToNumber, ToBoolean을 집중적으로 알아보고 ToPrimitive는 대략만 훑어본다.

4.2.1 ToString

'문자열이 아닌 값 → 문자열' 변환 작업은 ES5 §9.8의 ToString 추상 연산 로직이 담당한다.

내장 원시 값은 본연의 문자열화 방법이 정해져 있다(예: null → "null", undefined → "undefined", true → "true"). 숫자는 예상대로 그냥 문자열로 바꾸고, 너무 작거나 큰 값은 지수 형태로 바꾼다(2장 값 참고).

```
// '1.07'에 '1000'을 7번 곱한다.
var a = 1.07 * 1000 * 1000 * 1000 * 1000 * 1000 * 1000 * 1000;

// 소수점 이하로 3 x 7 => 21자리까지 내려간다.
a.toString(); // "1.07e21"
```

일반 객체는 특별히 지정하지 않으면 기본적으로 (Object.prototype.toString() 에 있는) toString() 메서드가 내부 [[Class]]를 반환한다(예: "[object Object]").

자신의 toString() 메서드를 가진 객체는 문자열처럼 사용하면 자동으로 이 메서드가 기본 호출되어 toString()을 대체한다.

엄밀히는 '객체 → 문자열' 강제변환 시 ToPrimitive 추상 연산(ES5 §9.1) 과정을 거치지만, 이 장 ToNumber 절에서 다시 자세히 다루므로 자세한 의미는 일단 넘어가자.

배열은 기본적으로 재정의된 toString( )이 있다. 문자열 변환 시 모든 원소 값이 (각각 문자열로 바뀌어) 콤마(,)로 분리된 형태로 이어진다.

```
var a = [1,2,3];
a.toString(); // "1,2,3"
```

또한 toString( ) 메서드는 명시적으로도 호출 가능하며, 문자열 콘텍스트에서 문자열 아닌 값이 있을 경우에도 자동 호출된다.

JSON 문자열화

ToString은 JSON.stringify(..) 유틸리티를 사용하여 어떤 값을 JSON 문자열로 직렬화하는 문제와도 연관된다.

JSON 문자열화는 강제변환과 똑같지는 않지만, 방금 전 살펴본 ToString 규칙과 관련이 있으므로 JSON 문자열화에 대해 잠시 알아본다.

대부분 단순 값들은, 직렬화 결과가 반드시 문자열이라는 점을 제외하고는, JSON 문자열화나 toString( ) 변환이나 기본적으로 같은 로직이다.

```
JSON.stringify( 42 ); // "42"
JSON.stringify( "42" ); // ""42"" (따옴표가 붙은 문자열 인자를 문자열화한다)
JSON.stringify( null ); // "null"
JSON.stringify( true ); // "true"
```

JSON 안전 값JSON-safe value(JSON 표현형representation으로 확실히 나타낼 수 있는 값)은 모두 JSON.stringify(..)으로 문자열화할 수 있다.

JSON 안전 값이 아닌 것들을 반대로 떠올리면 이해가 빠를 것이다(예: undefined, 함수, (ES6부터) 심볼, 환형 참조circular references 객체(프로퍼티 참조가 무한 순환되는 구조의 객체)[02]). 이들은 모두 다른 언어로 이식하여 JSON 값으로 쓸 수 없는, 표준 JSON 규격을 벗어난 값이다.

JSON.stringify(..)는 인자가 undefined, 함수, 심볼 값이면 자동으로 누락시키며, 이런 값들이 만약 배열에 포함되어 있으면 (배열 인덱스 정보가 뒤바뀌지 않도록) null로 바꾼다. 객체 프로퍼티에 있으면 간단히 지워버린다.

```
JSON.stringify( undefined ); // undefined
JSON.stringify( function(){} ); // undefined

JSON.stringify(
[1,undefined,function(){},4]
); // "[1,null,null,4]"

JSON.stringify(
{ a:2, b:function(){} }
); // "{"a":2}"
```

혹시라도 JSON.stringify(..)에 환형 참조 객체를 넘기면 에러가 난다.

객체 자체에 toJSON( ) 메서드가 정의되어 있다면, 먼저 이 메서드를 호출하여 직렬화한 값을 반환한다.

부적절한 JSON 값이나 직렬화하기 곤란한 객체 값을 문자열화하려면 해당 객체의 JSON 안전 버전을 반환하는 기능의 toJSON( ) 메서드를 따로 정의해야 한다. 예를 들면,

02 역자주_ 마지막 객체가 첫 번째 객체를 참조하는 등 순환 참조가 발생하여 결국 메모리 누수(memory leak)를 유발하는 객체를 말합니다. HTML DOM 객체와 자바스크립트 간 연동 시 실수로 환형 참조를 하는 경우가 종종 있습니다.

```
var o = { };

var a = {
b: 42,
c: o,
d: function(){}
};

// 'a'를 환형 참조 객체로 만든다.
o.e = a;

// 환형 참조 객체는 JSON 문자열화 시 에러가 난다
// JSON.stringify( a );

// JSON 값으로 직렬화하는 함수를 따로 정의한다.
a.toJSON = function() {
// 직렬화에 프로퍼티 'b'만 포함시킨다.
return { b: this.b };
};

JSON.stringify( a ); // "{"b":42}"
```

toJSON( )이 JSON 문자열 표현형을 반환하리라 넘겨짚는 건 아주 흔한 오해다. 문자열을 문자열화할 의도가 아니라면 (대부분 그렇다!) 정확하지 않을 가능성이 높다. toJSON( )은 (어떤 타입이든) 적절히 평범한 실제 값을 반환하고, 문자열화 처리는 JSON.stringify(. .)이 담당한다.

다시 말해 toJSON( )의 역할은 "문자열화하기 적당한 JSON 안전 값으로 바꾸는 것"이지, "JSON 문자열로 바꾸는 것"이 아니다. 많은 개발자가 잘못 알고 있는 부분이다.

```
var a = {
val: [1,2,3],

// 맞다!
toJSON: function(){
```

```
return this.val.slice( 1 );
}
};

var b = {
val: [1,2,3],

// 틀리다!
toJSON: function(){
return "[" +
this.val.slice( 1 ).join() +
"]";
}
};

JSON.stringify( a ); // "[2,3]"

JSON.stringify( b ); // ""[2,3]""
```

두 번째 호출 코드는 배열 자체가 아니라 반환된 문자열을 다시 문자열화한다. 이건 프로그래머가 의도했던 바가 아닐 것이다.

`JSON.stringify( .. )` 이야기가 나왔으니, 잘 알려지지 않은 유용한 기능 하나를 귀띔하겠다.

배열 아니면 함수 형태의 대체자replacer를 `JSON.stringify( .. )`의 두 번째 선택 인자로 지정하여 객체를 재귀적으로 직렬화하면서 (포함할 프로퍼티와 제외할 프로퍼티를 결정하는) 필터링하는 방법이 있다. `toJSON( )`이 직렬화할 값을 준비하는 방식과 비슷하다.

대체자가 배열이면 전체 원소는 문자열이어야 하고, 각 원소는 객체 직렬화의 대상 프로퍼티명이다. 즉, 여기에 포함되지 않은 프로퍼티는 직렬화 과정에서 빠진다.

대체자가 함수면 처음 한 번은 객체 자신에 대해, 그 다음엔 각 객체 프로퍼터별로 한 번씩 실행하면서 매번 키, 값 두 인자를 전달한다. 직렬화 과정에서 해당 키를

건너뛰려면 undefined를, 그 외엔 해당 값을 반환한다.

```
var a = {
b: 42,
c: "42",
d: [1,2,3]
};

JSON.stringify( a, ["b","c"] ); // "{"b":42,"c":"42"}"

JSON.stringify( a, function(k,v){
if (k !== "c") return v;
} );
// "{"b":42,"d":[1,2,3]}"
```

함수인 대체자는 최초 호출 시 (객체 자신을 전달하므로) 키 인자 k는 undefined다. 대체자는 if 문에서 키가 " c " 인 프로퍼티를 솎아낸다. 문자열화는 재귀적으로 이루어지므로 배열 [1,2,3]의 각 원소(1,2,3)는 v로, 인덱스(0,1,2)는 k로 각각 대체자 함수에 전달된다.

JSON.stringify(..)은 세 번째 선택 인자는 스페이스space라고 하며 사람이 읽기 쉽도록 들여쓰기를 할 수 있다. 들여쓰기를 할 빈 공간의 개수를 숫자로 지정하거나 문자열(10자 이상이면 앞에서 10자까지만 잘라 사용한다)을 지정하여 각 들여쓰기 수준에 사용한다.

```
var a = {
b: 42,
c: "42",
d: [1,2,3]
};

JSON.stringify( a, null, 3 );
// "{
//   "b": 42,
//   "c": "42",
```

```
//   "d": [
//     1,
//     2,
//     3
//   ]
// }"

JSON.stringify( a, null, "-----" );
// "{
// -----"b": 42,
// -----"c": "42",
// -----"d": [
// ----------1,
// ----------2,
// ----------3
// -----]
// }"
```

`JSON.stringify( .. )`은 직접적인 강제변환의 형식은 아니지만 두 가지로 이유로 `ToString` 강제변환과 연관된다.

1. 문자열, 숫자, 불리언, `null` 값이 `JSON`으로 문자열화하는 방식은 `ToString` 추상 연산의 규칙에 따라 문자열 값으로 강제변환되는 방식과 동일하다.
2. `JSON.stringify( .. )`에 전달한 객체가 자체 `toJSON( )` 메서드를 갖고 있다면, 문자열화 전 `toJSON( )`가 자동 호출되어 `JSON` 안전 값으로 (말하자면) "강제변환"된다.

4.2.2 ToNumber

'숫자 아닌 값 → 수식 연산이 가능한 숫자' 변환 로직은 ES5 §9.3 `ToNumber` 추상 연산에 잘 정의되어 있다.

예를 들어 `true`는 `1`, `false`는 `0`이 된다. `undefined`는 `NaN`으로, (희한하게도) `null`은 `0`으로 바뀐다.

문자열 값에 ToNumber를 적용하면 대부분 숫자 리터럴(3장 네이티브 참고) 규칙/구문과 비슷하게 작동한다. 변환이 실패하면 결과는 (숫자 리터럴 구문 에러가 아닌) NaN이다. 한 가지 차이는 0이 앞에 붙은 8진수는 ToNumber에서 올바른 숫자 리터럴이라도 8진수로 처리하지 않는다는 점이다(대신 일반 10진수로 처리한다).

문자열 값에 대한 숫자 리터럴 문법과 ToNumber는 아주 세밀하고 미묘한 뉘앙스의 차이가 있어 더 이상 언급하지 않는다. 자세한 내용은 ES5 §9.3.1를 직접 참고하자.

객체(그리고 배열)는 일단 동등한 원시 값으로 변환 후, 그 결과값(아직 숫자가 아닌 원시 값)을 앞서 설명한 ToNumber 규칙에 의해 강제변환한다.

동등한 원시 값으로 바꾸기 위해 ToPrimitive 추상 연산(ES5 §9.1) 과정에서 해당 객체가 valueOf( ) 메서드를 구현했는지 (DefaultValue 연산을 내부적으로 이용하여 - ES5 §8.12.8 참고) 확인한다. valueOf( )를 쓸 수 있고 반환 값이 원시 값이면 그대로 강제변환하되, 그렇지 않을 경우 (toString( ) 메서드가 존재하면) toString( )을 이용하여 강제변환한다.

어찌해도 원시 값으로 바꿀 수 없을 땐 TypeError 오류를 던진다.

ES5부터는 [[Prototype]]가 null인 경우 대부분 Object.create(null)를 이용하여 강제변환이 불가능한 객체(valueOf( ), toString( ) 메서드가 없는 객체)를 생성할 수 있다. [[Prototype]]에 관한 자세한 내용은 『You Don't Know JS: this & Object Prototypes』[03]을 참고하자.

이 장 끝 부분에서 숫자로의 강제변환 방법을 자세히 설명한다. 지금은 다음 예제 코드에서 Number(. .)가 그 일을 대신 해준다고 생각하자.

03 http://goo.gl/OiWkXO

다음 코드를 참고하기 바란다.

```
var a = {
valueOf: function(){
return "42";
}
};

var b = {
toString: function(){
return "42";
}
};

var c = [4,2];
c.toString = function(){
return this.join( "" ); // "42"
};

Number( a ); // 42
Number( b ); // 42
Number( c ); // 42
Number( "" ); // 0
Number( [] ); // 0
Number( [ "abc" ] ); // NaN
```

4.2.3 ToBoolean

자바스크립트에서 불리언에 대해 간단히 살펴보자. 아직도 많은 혼동과 오해를 불러일으키는 주제니 바짝 긴장하기 바란다!

우선 자바스크립트에는 `true`와 `false`라는 키워드가 존재하며, 우리가 예상하는 그대로 잘 작동한다. 흔히들 `1`과 `0`이 각각 `true`, `false`에 해당한다고 생각하는데 다른 언어에서는 그럴지 몰라도 자바스크립트에서는 숫자는 숫자고, 불리언은 불리언으로 서로 별개다. `1`을 `true`로, `0`을 `false`로 (역방향도 마찬가지) 강제변환할

수는 있지만, 그렇다고 두 값이 똑같은 건 아니다.

Falsy 값

지금부터가 재미있다. true/false가 아닌 값을 불리언에 상당한 값으로 강제변환했을 때, 이 값들은 어떻게 작동할까?

자바스크립트의 모든 값은 다음 둘 중 하나다.

1. 불리언으로 강제변환하면 false가 되는 값
2. 1번을 제외한 나머지(즉, 명백히 true인 값)

여러분에게 짓궂은 장난을 치려는 게 아니다. 불리언으로 강제변환 시 false가 되는 몇 개 안 되는 특별한 값들이 자바스크립트 명세에 이미 정의되어 있다.

이 값들의 목록은 어떻게 알 수 있을까? ES5 §9.2 ToBoolean 추상 연산을 읽어보면 "불리언으로" 강제변환 시 모든 가능한 경우의 수가 나열되어 있다.[04]

명세가 정의한 " falsy " 값은 다음과 같다.

- undefined
- null
- false
- +0, -0, NaN
- " "

04 역자주_ ToBoolean 추상 연산은 다음 목록에 의해 인자로 전달받은 값을 불리언 타입의 값으로 변환합니다.

인자 타입	결과값
Undefined	false
Null	false
Boolean	인자 값과 동일(변환 안 함)
Number	인자가 +0, -0, NaN이면 false, 그 외에는 true
String	인자가 공백 문자열(length가 0)이면 false, 그 외에는 true
Object	true

이게 전부다. 이 목록에 있으면 " falsy " 값이며, 불리언으로 강제변환하면 false다.

반대로 이 목록에 없으면 다른 목록, 즉 " truthy " 값 목록에 있는 것이다. 그런데 정작 자바스크립트 명세에 " truthy " 값 목록 같은 건 없다. 모든 객체는 명백히 truthy하다는 식의 몇 가지 예시만 있을 뿐 " falsy " 값 목록에 없으면 응당 " truthy " 값이 되는 것이다.

Falsy 객체

잠깐, Falsy 객체라고? 절 제목 자체가 모순 아닌가? 방금 전 명세에 모든 객체는 truthy하다고 했는데, 그럼 "Falsy 객체" 같은 건 없는 것 아닐까?

왠 생뚱맞은 소리인지?

("", 0, false 같은) falsy 값을 둘러싼 객체 래퍼를 두고 하는 얘기가 아닐까 생각하기 쉽다. 하지만 함정에 빠지지 말자!

사람을 멍하게 만드는 명세의 농담이라고나 할까?

```
var a = new Boolean( false );
var b = new Number( 0 );
var c = new String( "" );
```

a, b, c는 명백히 falsy 값을 감싼 객체(3장 네이티브 참고)다. 세 변숫값은 true일까? false일까? 이건 쉽다.

```
var d = Boolean( a && b && c );

d; // true
```

d가 true인 것으로 봐서 세 변수 모두 true임을 알 수 있다.

Boolean(..)으로 a && b && c를 감싼 이유는 뭘까? 나중에 이 장 뒷 부분에서 다시 설명하니 살짝만 머릿속에 메모하자. 궁금함을 못 참겠다면 Boolean(..)을 빼고 d = a && b && c에서 d 값을 확인하기 바란다.

"falsy 객체"가 falsy 값을 감싼 객체가 아니라면 대체 뭐란 말인가?

여기부터가 어려운데, 사실 이 객체는 순수 자바스크립트의 일부가 아니다(실제로 여러분이 짠 자바스크립트 프로그램에도 이런 객체가 나올 수 있다).

뭐라고!?

일반적인 자바스크립트의 의미semantics뿐만 아니라 브라우저만의 특이한 작동 방식을 가진 값을 생성하는 경우가 있는데, 이것이 바로 "falsy 객체"의 정체다.

"falsy 객체"는 겉보기엔 (프로퍼티 등이) 평범한 객체처럼 작동할 것 같지만 불리언으로 강제변환하면 false다.

왜냐고?

가장 유명한 사례가 개발자들이 (자바스크립트 엔진이 아닌) DOM에서 사용했던 유사 배열(객체), document.all이다. document.all은 웹 페이지의 요소를 자바스크립트 프로그램에서 가져올 수 있게 해주었고, 그래서 실제로 " truthy " 한 일반 객체처럼 작동했다. 하지만 이젠 더 이상 아니다.

알 만한 사람들은 다 알겠지만 document.all은 "비표준"이며, 이미 오래 전에 비권장/폐기되었다.

"그냥 날려버리면 되지 않았나?" 하하, 마음은 굴뚝같았지만 그러기엔 document.all에 의존하는 레거시 코드베이스가 너무 많았다.

본론으로 다시 돌아와서, 그런데 왜 " falsy " 객체로 작동해야 할까? document.all을 불리언으로 강제변환한 결과값을 (if 문 등에서) 오래된, 비표준

IE 브라우저를 감지하는 수단으로써 줄곧 사용해왔기 때문이다. IE는 한참이 지난 후에서야 표준을 준수하기 시작했고, 지금은 오히려 많은 경우, 다른 브라우저 이상으로 표준을 밀어붙이고 있다.

하지만 아직도 `if (document.all) { /* it's IE */ }` 같은 낡은 코드가 사용 중이고, 아마 완전히 박멸될 것 같지는 않다. 이런 레거시 코드는 10년여 전 사용자들에게 불쾌한 경험을 선사했던, 구 IE 브라우저에 아직도 기대고 있다.

`document.all`을 완전히 걷어낼 순 없지만, IE 개발팀 입장에선 `if (document.all) { .. }` 같은 코드가 옛날처럼 잘 돌아가게 놔두면, 현대 IE에서는 새로운, 완전히 표준을 준수하는standards-compliant 코드 로직을 갖게 되므로 난감했다.

"신이여, 어쩌란 말입니까?"

"알았다! 자바스크립트 타입 체계를 살짝 바꿔서 `document.all`이 `falsy`인 것처럼 돌아가게 하는 거야!"

오, 이런. 대부분의 자바스크립트 개발자가 이해할 수 없는 이런 미친 꼼수를 떠올리다니! 하지만 승산 없는 이 문제를 그냥 방치하다간 일이 더 꼬이고 말테니...

그래서 결국... 브라우저 때문에 말도 안 되는 비표준("`falsy` 객체")이 자바스크립트에 더해진 것이다. 휴!

Truthy 값

`truthy` 값의 의미는 정확히 무엇일까? 다시 이야기하지만 `falsy` 값 목록에 없으면 무조건 `truthy` 값이다.

```
var a = "false";
var b = "0";
var c = "' '";
```

```
var d = Boolean( a && b && c );

d;
```

d엔 어떤 값이 들어갈까? true 아니면 false일 텐데...

정답은 true다. 이유는? 문자열 값을 보면 falsy처럼 보이지만 문자열 값 자체는 모두 truthy이기 때문이다(falsy 값 목록에 있는 유일한 문자열은 " "다).

다음 예제를 보자.

```
var a = []; // 빈 배열 - truthy일까, falsy일까?
var b = {}; // 빈 객체 - truthy일까, falsy일까?
var c = function(){}; // 빈 함수 - truthy일까, falsy일까?

var d = Boolean( a && b && c );

d;
```

그렇다, 알아맞혔겠지만 이번에도 d는 true다. 같은 이유에서다. 외양이야 어떻든 [ ], { }, function ( ){}는 falsy 값 목록에 없으므로 모두 truthy 값이다.

truthy 값 목록은 사실상 무한하여 작성이 불가하다. 그래서 몇 안 되는 falsy 값 목록을 만들고 참조하라는 것이다.

5분만 투자해서 포스트잇에 falsy 값 목록을 적어 모니터에 붙여놓든지 아니면 그냥 외워버리자. 어쨌든 이 목록을 암기하고 있어야 필요할 때마다 확인하여 가상의 truthy 값 목록을 머릿속에 떠올릴 수 있을 테니.

truthy/falsy 개념은 어떤 값을 불리언 타입으로 (명시적/암시적) 강제변환 시 그 값의 작동 방식을 이해한다는 점에서 중요하다. 이제 여러분은 두 목록 모두 알고 있으니 어떤 강제변환 문제가 나와도 당황하지 않고 달려들 준비가 되었다.

4.3 명시적 강제변환

명시적 강제변환explicit coercion은 분명하고 확실한 타입 변환이다. 개발자들이 흔히 사용하는 타입 변환은 대개 이 명시적 강제변환 범주 내에 속한다.

이 절의 목적은 훗날 다른 개발자가 본인이 작성한 코드로 인해 구렁텅이에 빠지지 않도록 값의 타입 변환 과정을 분명하고 명확하게 할 수 있는 패턴을 여러분 스스로 찾게 하는 것이다. 코드가 명확하면 명확할수록 다른 개발자가 내 코드를 보더라도 쓸데없이 내 의도를 추론할 필요 없이 단번에 이해할 수 있다.

명시적 강제변환은 정적 타입 언어에서 지극히 당연하다고 여겨지는 타입 변환의 관례를 충실히 따르고 있기에 별다른 논쟁거리는 없다. 따라서 (적어도 지금은) 명시적 강제변환이 나쁜 것도 아니고, 이슈가 많은 주제도 아니라고 당연하게 받아들이자. 이 부분은 나중에 다시 이야기할 것이다.

4.3.1 명시적 강제변환: 문자열 ↔ 숫자

우선, 가장 간단하면서도 가장 잦은 강제변환이라 할 수 있는 '문자열 ↔ 숫자' 강제변환이다.

'문자열 ↔ 숫자' 강제변환은 `String( .. )`과 `Number( .. )` 함수를 이용하는데, 앞에 `new` 키워드가 붙지 않기 때문에 객체 래퍼를 생성하는 것이 아니란 점을 눈여겨보자.

다음은 두 타입 간 명시적 강제변환 코드다.

```
var a = 42;
var b = String( a );

var c = "3.14";
var d = Number( c );

b; // "42"
d; // 3.14
```

앞서 설명했던 ToString 추상 연산 로직에 따라 String(..)은 값을 받아 원시 문자열로 강제변환한다. Number(..) 역시 마찬가지로 ToNumber 추상 연산 로직에 의해 어떤 값이든 원시 숫자 값으로 강제변환한다.

누가 봐도 결과적으로 타입 변환이 일어난다는 사실은 의심할 여지가 없으므로 '명시적 강제변환'이 맞다.

사실 이러한 사용법은 다른 정적 타입 언어와 비슷하다.

예컨대, C/C++ 언어에서 (int)x나 int(x)는 둘 다 x를 정수로 변환한다. 둘 다 문법상 맞지만 함수 호출 형태의 후자를 선호하는 개발자들이 더 많다. 자바스크립트의 Number(x)도 상당히 닮았다. 하지만 자바스크립트에서는 실제로 함수 호출인데 문제가 될까? 그렇지 않다.

String(..), Number(..) 이외에도 '문자열 ↔ 숫자'의 "명시적인" 타입 변환법은 또 있다.

```
var a = 42;
var b = a.toString();

var c = "3.14";
var d = +c;

b; // "42"
d; // 3.14
```

a.toString( ) 호출은 ("toString"만 봐도 "to a string(문자열로)"란 의미가 뚜렷하므로) 겉보기엔 명시적이지만, 몇 가지 암시적인 요소가 감춰져 있다. 원시 값 42에는 toString( ) 메서드가 없으므로 엔진은 toString( )를 사용할 수 있게 자동으로 42를 객체 래퍼로 "박싱"한다. 말하자면 "명시적으로 암시적인explicitly implicit" 작동이다.

+c의 +는 단항 연산자unary operator(피연산자가 하나뿐인 연산자)다. 수학책에 나오는 덧셈(아니면 문자열 붙이기)을 하는 게 아니라, 단항 연산자 +는 피연산자 c를 숫자로, 명시적 강제변환한다.

+c가 명시적 강제변환이라고? 개발자의 경험과 시야에 따라 다르다. 단항 연산자 +가 숫자로 강제변환하는 기능이란 사실을 분명히 알고 있다면 (여러분이 지금 막 알게 된 것처럼) 별 상관없다. 하지만 태어나서 단 한 번도 이 연산자를 본 적이 없는 사람에게는 그저 혼란스럽고 뭔가 부작용이 숨겨져 있을 것만 같은 불길한 느낌이 들 것이다.

오픈 소스 자바스크립트 커뮤니티에서는 + 단항 연산자를 명시적 강제변환 형식으로 대부분 인정하는 분위기다.

설사 +c처럼 쓰는 걸 좋아하는 사람일지라도 헷갈려 보이는 경우가 더러 있다. 다음 코드를 보자.

```
var c = "3.14";
var d = 5+ +c;
d; // 8.14
```

- 단항 연산자 역시 +처럼 강제변환을 하지만 숫자의 부호를 뒤바꿀 수도 있다. 그렇다고 바뀐 부호를 바로잡고자 --로 하면 증감 연산자가 되어버리니 안 될 노릇이다. 대신 - -" 3.14 " 처럼 두 연산자 사이에 공란을 하나 넣어주면 무사히 3.14로 강제변환된다.

이런 식으로 단항 연산자 뒤에 이진 연산자binary operator(예: 덧셈은 +)를 붙이면 무수히 많은, 오싹한 조합이 나올 수 있다. 다음은 그 중 하나다.

```
1 + - + + + - + 1; // 2
```

가급적 +/- 단항 연산자를 다른 연산자와 인접하여 사용하지 않기 바란다. 코드가 잘 돌아간다 해도 별로 좋지 않은 생각이다. d = +c(또는 d =+ c)는 d에 c를 더하는 d +=c와 완전히 다른 코드임에도 너무 헷갈리기 쉽다.

극단적인 예를 하나 더 들면 ++/-- 증감 연산자와 + 단항 연산자가 이웃하는 경우다(예: a +++b, a + ++b, a + + +b).

명시적으로 변환하여 문제를 악화시키지 말고 혼동을 줄이는 게 좋다는 걸 기억하자.

날짜 → 숫자

+ 단항 연산자는 'Date 객체 → 숫자' 강제변환 용도로도 쓰인다. 결과값이 날짜/시각 값을 Unix 타임스탬프 표현형(1 January 1970 00:00:00 UTC 이후로 경과한 시간을 밀리 초 단위로 표시)이기 때문이다.

```
var d = new Date( "Mon, 18 Aug 2014 08:53:06 CDT" );

+d; // 1408369986000
```

다음과 같이 현재 시각을 타임스탬프로 바꿀 때 관용적으로 사용하는 방법이다.

```
var timestamp = +new Date();
```

생성자 호출(new를 붙여 함수 호출) 시 전달 인자가 없다면 괄호 ( )는 생략 가능한, 기이한 자바스크립트의 구문 "트릭trick"이 있다는 걸 알고 있는 독자도 있을 것이다. 그래서 앞의 코드는 var timestamp = +new Date;처럼 써도 된다. 하지만 ( )를 생략하면 가독성이 좋아진다는 말을 인정하지 않는 개발자들도 있으며, 일반적인 fn( ) 호출 형식이 아닌, new fn( ) 호출 형식에만 적용되는, 흔치 않은 예외 구문이다.

그러나 강제변환을 하지 않아도 Date 객체로부터 타임스탬프를 얻는 방법이 있다. 오히려 강제변환을 하지 않는 쪽이 더 명시적이므로 권장할 만하다.

```
var timestamp = new Date().getTime();
// var timestamp = (new Date()).getTime();
// var timestamp = (new Date).getTime();
```

하지만 ES5에 추가된 정적 함수 Date.now( )를 쓰는 게 더 낫다.

```
var timestamp = Date.now();
```

구 버전 브라우저에서 Date.now( )를 사용하려면 다음 폴리필을 심자.

```
if (!Date.now) {
  Date.now = function() {
    return +new Date();
  };
}
```

날짜 타입에 관한 한 강제변환은 권하고 싶지 않다. 현재 타임스탬프는 Date.now( )로, 그 외 특정 날짜/시간의 타임스탬프는 new Date(..).getTime( )를 대신 쓰도록 하자.

이상한 나라의 틸드(~)

~(틸드)는 종종 사람들이 간과하는 자바스크립트의 강제변환 연산자이자, 가장 헷갈리는 연산자의 대명사다. 심지어 사용법을 터득한 개발자마저 사용을 꺼리는 연산자다. 그래도 이 책을 포함한 "You Don't Know JS" 시리즈를 읽는 독자 여러분의 모험심을 존중하여 ~에 대해 알아보자.

앞에서 자바스크립트 비트 연산자는 오직 32비트 연산만 가능하다고 했다. 즉, 비트 연산을 하면 피연산자는 32비트 값으로 강제로 맞춰지는데, ToInt32 추상 연산(ES5 §9.5)이 이 역할을 맡는다.

우선 ToInt32은 ToNumber 강제변환한다. " 123 " 이라면 ToInt32 규칙을 적

용하기 전 123으로 바꾼다.

엄밀히 말해 이 자체는 (타입이 바뀐 것은 아니므로!) 강제변환이 아니지만, 숫자 값에 |나 ~ 비트 연산자를 적용하면 전혀 다른 숫자 값을 생성하는 강제변환 효과가 있다.

예를 들어, 아무 연산도 하지 않는 (2장 값에서 봤던) `0 | x`의 "OR" 연산자(|)는 사실상 ToInt32 변환만 수행한다.

```
0 | -0; // 0
0 | NaN; // 0
0 | Infinity; // 0
0 | -Infinity; // 0
```

이러한 특수 숫자들은 (64비트 IEEE 754 표준에서 유래했기에 - 2장 값 참고) 32비트로 나타내는 것이 불가능하므로 ToInt32 연산 결과는 0이다.

`0 |` ——이 강제적인 ToInt32 연산의 명시적인 형태냐 아니면 더 암시적인 형태냐 하는 문제는 논란의 여지가 있다. 명세서 관점에서 보면, 두말 할 것 없이 명시적인 것이지만, 이 수준의 비트 연산을 이해하지 못한 사람들 눈에는 일종의 암시적인 마법으로 보일 수 있다. 어쨌거나, 이 장의 다른 절들과 내용을 맞추기 위해 여기서는 명시적이라고 하겠다.

다시 ~로 돌아와서, ~ 연산자는 먼저 32비트 숫자로 "강제변환"한 후 NOT 연산을 한다(각 비트를 거꾸로 뒤집는다).

!이 불리언 값으로 강제변환하는 것뿐만 아니라 비트를 거꾸로 뒤집는 것과 아주 비슷하다.

그런데... 가만!? 비트를 거꾸로 한다는 둥 이런 얘기는 왜 하는 걸까? 아주 전문적이고 미묘한 주제인 듯싶은데... 자바스크립트 개발자가 비트 하나하나를 따져야

할 경우가 있을까?

여러분 대학 시절에 컴퓨터 과학 또는 이산 수학discrete mathematics 강의를 들으면서 ~의 정의를 어떻게 배웠는지 기억을 되살려보자. ~는 2의 보수complement를 구한다. 맞다! 바로 이거다, 이제야 좀 알 것 같다!

자, 다시... ~x는 대략 -(x+1)와 같다. 이상한 것 같지만, 왜 그런지 금방 알 수 있다.

```
~42; // -(42+1) ==> -43
```

대관절 ~ 얘기가 왜 나왔을까, 강제변환과 무슨 상관인가 어리둥절할 것이다. 그럼 요점으로 직행하겠다.

-(x+1)를 보자. 이 연산의 결과를 0(정확히는 -0)으로 만드는 유일한 값은 무엇인가? 정답은 -1이다. 다시 말해, 일정 범위 내의 숫자 값에 ~ 연산을 할 경우, 입력 값이 -1이면 (false로 쉽게 강제변환할 수 있는) falsy한 0, 그 외엔 truthy한 숫자 값이 산출된다.

그래서 무슨 상관이란 말인가?

여기서 -1과 같은 성질의 값을 흔히 "경계 값sentinel value"이라고 일컫는데, 동일 타입(숫자)의 더 확장된 값의 집합 내에서 임의의 어떤 의미를 부여한 값이다. 예를 들어, C 언어의 함수는 대개 -1을 경계 값으로 사용하는데 return >= 0는 "성공", return -1은 "실패"라는 의미를 각각 부여한다.

자바스크립트 역시 선배의 전례를 따라 문자열 메서드 indexOf(..)로 특정 문자의 인덱스를 찾을 때 발견했을 경우 0 이상의 숫자 값(인덱스)을, 발견하지 못했을 경우 -1을 반환한다.

사실, indexOf(..)는 단순히 위치를 확인하는 기능보단 어떤 하위 문자열이 다른 문자열에 포함되어 있는지 조사하는 용도로 더 많이 쓰인다. 다음 코드를 보자.

```
var a = "Hello World";

if (a.indexOf( "lo" ) >= 0) { // true
  // found it!
}

if (a.indexOf( "lo" ) != -1) { // true
  // found it
}

if (a.indexOf( "ol" ) < 0) { // true
  // not found!
}

if (a.indexOf( "ol" ) == -1) { // true
  // not found!
}
```

그런데 내 눈에는 >= 0나 == -1 같은 코드가 좀 지저분해 보인다. 기본적으로 이런 부류의 코드는 "구멍 난 추상화leaky abstraction", 즉 (경계 값 -1을 "실패"로 정해버린) 내부 구현 방식을 내가 짠 코드에 심어놓은 꼴이다. 이런 부분은 감추는 게 낫다고 생각한다.

자, 이제 드디어 ~이 왜 쓸모 있는지 발표하겠다. indexOf( )에 ~를 붙이면 어떤 값을 "강제변환"(실제로는 단순히 변형transform)하여 적절히 불리언 값으로 만들 수 있다.

```
var a = "Hello World";

~a.indexOf( "lo" ); // -4 <-- truthy!

if (~a.indexOf( "lo" )) { // true
  // 찾았다!
}

~a.indexOf( "ol" ); // 0 <-- falsy!
!~a.indexOf( "ol" ); // true
```

```
if (!~a.indexOf( "ol" )) { // true
  // 못 찾았다!
}
```

~은 `indexOf(..)`로 검색 결과 "실패" 시 `-1`을 falsy한 `0`으로, 그 외에는 truthy한 값으로 바꾼다.[05]

`-(x+1)`은 `~`의 의사 알고리즘pseudo-algorithm으로, 내부적으로 `~-1`을 `-0`으로 만들지만, 수학 연산이 아닌 비트 연산이므로 결과값은 `0`이 된다.

굳이 따진다면 `if (~a.indexOf(..))` 문은 `a.indexOf(..)`의 결과값이 `0`이면 `false`, 그 외엔 `true`로 암시적인 강제변환을 하는 것이라 할 수도 있지만, 여기서 설명하고자 하는 의도를 여러분이 잘 따라오고 있다면 `~`이 오히려 명시적인 강제변환에 더 가깝지 않나 싶다.

이전의 `>= 0`나 `== -1` 같은 잡동사니들과 견주어 보면 코드가 훨씬 깔끔하지 않은가?

비트 잘라내기

`~` 용도를 하나 더 소개한다. 숫자의 소수점 이상 부분을 잘라내기truncate 위해 (즉, 완전수로 "강제변환"하려고) 더블 틸드double tilde `~~`를 사용하는 개발자들이 있다. 흔히들 (착각에 불과하지만) 이렇게 하면 `Math.floor(..)`와 같은 결과가 나온다고 생각한다.

`~~`가 하는 일은 이렇다. 먼저 맨 앞의 `~`는 `ToInt32` "강제변환"을 적용한 후 각 비트를 거꾸로 한다. 그리고 두 번째 `~`는 비트를 또 한 번 뒤집는데, 결과적으로 원

05 역자주_ 한눈에 들어오지 않을 독자들을 위해 부연하면 `a.indexOf( "lo" )`는 3("Hello World"의 4번째 위치에 lo가 시작되므로 인덱스는 4 - 1 = 3)이고, `~3`, 즉 3의 NOT 연산 결과는 -4입니다.

래 상태로 되돌린다. 결국 ToInt32 "강제변환"(잘라내기truncation)만 하는 셈이다.

~~의 비트 이중 뒤집기 묘기는 "명시적 강제변환: * → 불리언"의 패리티 이중 부정(!!)과 유사하다.

그러나 ~~ 사용 시 유의할 점이 있다. 우선 ~~ 연산은 32비트 값에 한하여 안전하다. 그런데 그보다도 음수에서는 `Math.floor( .. )`과 결과값이 다르다는 사실을 조심하자!

```
Math.floor( -49.6 ); // -50
~~49.6; // -49
```

`Math.floor( .. )`과의 다른 점은 차치하더라도 ~~x는 (32비트) 정수로 상위 비트를 잘라낸다. 하지만 같은 일을 하는 `x | 0`가 (조금이라도) 더 빠를 것 같다.

그럼 굳이 왜 `x | 0` 대신 ~~x를 써야할까? 바로 연산자 우선 순위 때문이다(5장 문법 참고).

```
~~1E20 / 10; // 166199296

1E20 | 0 / 10; // 1661992960
(1E20 | 0) / 10; // 166199296
```

이전에 제시한 다른 조언들과 마찬가지로 여러분의 코드를 읽을 주변 동료 개발자가 연산자의 작동 원리를 적절히 이해하고 있다는 전제하에 ~와 ~~를 명시적인 "강제변환" 및 값 변형 장치로 잘 활용하기 바란다.

4.3.2 명시적 강제변환: 숫자 형태의 문자열 파싱

문자열에 포함된 숫자를 파싱해도 '문자열 → 숫자' 강제변환과 결과는 비슷하지

만, 앞서 배운 타입 변환과는 분명한 차이가 있다.

```
var a = "42";
var b = "42px";

Number( a ); // 42
parseInt( a ); // 42

Number( b ); // NaN
parseInt( b ); // 42
```

문자열로부터 숫자 값의 파싱은 비 숫자형 문자non-numeric character를 허용한다. 즉, 좌 → 우 방향으로 파싱하다가 숫자 같지 않은 문자를 만나면 즉시 멈춘다. 반면, 강제변환은 비 숫자형 문자를 허용하지 않기 때문에 NaN를 내고 두 손 들어버린다.

파싱은 강제변환의 대안이 될 수 없다. 비슷해 보여도 목적 자체가 다르다. 우측에 비 숫자형 문자가 있을지 확실하지 않거나 별로 상관없다면 문자열을 숫자로 파싱한다. 반드시 숫자여야만 하고 " 42px " 같은 값은 되돌려야 한다면 문자열을 숫자로 강제변환한다.

`parseFloat( .. )`는 `parseInt( .. )`의 쌍둥이로 (이름 그대로) 문자열에서 부동 소수점 숫자를 추출한다.

`parseInt( .. )`는 문자열에 쓰는 함수임을 기억하자. 인자가 숫자라면 애당초 `parseInt( .. )`를 쓸 이유가 없다. 다른 타입(예: `true`, `function( ){..}`, `[1,2,3]`) 값들도 마찬가지다.

인자가 비 문자열non-string이면 제일 먼저 자동으로 문자열로 강제변환한다. 이는 일종의 감춰진 암시적 강제변환으로 프로그램에 이런 로직이 자꾸 들어가는 건 바람직하지 않다. 절대로 `parseInt( .. )`에 비 문자열 값을 넘기지 말자.

ES5 이전에는 무수한 자바스크립트 프로그램의 버그를 일으킨, `parseInt( .. )`의 다른 함정이 있었는데, 두 번째 인자로 기수radix(문자열을 숫자로 해석 시 사용되는 진법 종류)를 지정하지 않으면 문자열의 첫 번째 문자만 보고 마음대로 추정한다.

첫 번째 문자가 (관례상) `x`나 `X`면 16진수로, `0`이면 8진수로 문자열을 각각 임의로 해석하는 것이다.

16진수 형태의 문자열(앞에 `x`나 `X`가 붙은)은 그나마 별로 나타날 일이 드물다. 하지만 8진수는 흔하디 흔하다. 다음 예시를 보자.

```
var hour = parseInt( selectedHour.value );
var minute = parseInt( selectedMinute.value );

console.log(
  "The time you selected was: " + hour + ":" + minute
);
```

별 문제 없어 보인다. `hour`를 `08`, `minute`에 `09`로 하여 실행해보자. 결과는 `0:0`! 이유는 8, 9 모두 8진법에서 사용하는 숫자가 아니기 때문이다.

ES5 이전 시절의 디버깅 방법(항상 두 번째 인자로 `10`을 지정)은 간단하지만 잊어버리기 쉽다. 이렇게 해야 완전히 안전하다.

```
var hour = parseInt( selectedHour.value, 10 );
var minute = parseInt( selectedMiniute.value, 10 );
```

ES5 이후의 `parseInt( .. )`는 제멋대로 해석하지 않는다. 두 번째 인자가 없으면 무조건 10진수로 처리하는데, 왜 옛날부터 진작 이렇게 하지 않았는지 의문이다. 어쨌든 ES5 이전 환경에서는 항상 두 번째 인자, `10`을 전달해야 한다는 사실을 꼭 기억하자.

비 문자열 파싱

수년 전, 어떤 사람이 자바스크립트를 비웃기라도 하듯, parseInt(..)의 오작동 사례를 고발하여 주목을 받은 적이 있다.

```
parseInt( 1/0, 19 ); // 18
```

"무한대를 정수로 파싱하면 당연히 무한대 아닌가? 근데 18?" 당연한 생각인데 결과는 충격, 그 자체다. 이 코드만 놓고 보면 자바스크립트는 미친 것 같다.

누군가 용케 찾아내긴 했지만 그리 있음 직한 코드는 아니다. 그러나 잠시 이 문제를 진지하게 다뤄보면서 자바스크립트가 정말 그 정도로 미친 건지 음미해보자.

먼저, 비 문자열을 parseInt(..) 첫 번째 인자로 넘긴 것 자체가 잘못되었다. 있을 수 없는 일이고 당연히 말썽이 생긴다. 하지만 이런 상황이 닥쳐도 친절한 자바스크립트 엔진은 비 문자열을 문자열로 최대한 강제변환하려고 노력한다.

혹자는 이거야말로 말이 안 되는 로직 아니냐, 비 문자열을 받았다면 처리 자체를 거부해야지, 하고 따질 것이다. 예외를 던져야 한다? 으음, 자바 프로그래머다운 발상이다. 자바스크립트가 예외를 던지기 시작하면 그럼 거의 모든 줄을 try.. catch로 칭칭 감싸야 할 텐데... 생각만으로도 소름이 돋는다.

아니면 NaN을 반환해야 할까? 그럴 듯 하지만, 다음 코드를 보자.

```
parseInt( new String( "42" ) );
```

비 문자열 인자를 받았으니 실행되지 말아야 할까? String 객체 래퍼가 " 42 "로 언박싱되기를 바란다면, 42가 먼저 " 42 "가 된 다음, 다시 42로 파싱되어 반환되는 게 정말 이상한 일인가?

이렇게 종종 발생하는, 반은 명시적이고 반은 암시적인 강제변환이 꽤 유용하다고

생각한다. 예를 들면,

```
var a = {
  num: 21,
  toString: function() { return String( this.num * 2 ); }
};

parseInt( a ); // 42
```

인자 값을 강제로 문자열로 바꾼 다음 파싱을 시작하는 `parseInt( .. )`의 로직은 상당히 일리가 있다. 쓰레기를 집어넣고 쓰레기를 돌려받았다고 해서 쓰레기통을 비난하지 말지어다. 쓰레기통은 그저 자기가 할 일을 충실히 다했을 뿐!

만약 무한대(`1 / 0`의 결과) 같은 값을 넘긴다면 어떤 문자열로 변환하는 것이 최선이라 할 수 있을까? 가능한 후보는 "`Infinity`"와 "`∞`"이고, 그 중 자바스크립트는 전자를 택했다. 잘했다!

매 값마다 디폴트 문자열 표현형을 가지기 때문에, 자바스크립트가 디버깅이나 추정이 불가한, 신비한 블랙 박스가 아닌 점은 다행스러운 일이다.

그럼 두 번째 인자, 19로 넘어가자. 언뜻 봐도 부자연스럽고 일부러 꾸민 것 같다. 현존하는 어떤 자바스크립트 프로그램도 19진법을 쓰지 않는다. 그래도 이 우스꽝스러운 19진법의 세계를 잠시 엿보자. 19진법 체계의 유효한 숫자는 `0`부터 `9`, (대소문자 구별 없이) `a`부터 `i`까지다.

그럼, `parseInt( 1/0, 19 )`는 `parseInt( "Infinity", 19 )`인데, 어떻게 파싱되는 걸까? 음, 첫 번째 문자 "`I`"는 19진수 18에 해당한다. 두 번째 "`n`"은 `0-9`, `a-i` 범위 밖의 문자이므로 파싱은 여기서 멈춘다. "`42px`"가 "`p`"에서 멈춘 것과 같다.

그래서 결과는 18이다. 이제 보니 꽤 그럴싸하다. 에러도 아니고 `Infinity`도 아

니다. 18이란 결과에 이르기까지는 자바스크립트엔 쉽게 폐기할 수 없는 중요한 여정이었다.

놀랍지만 대단히 합리적인 parseInt(..)의 예제를 몇 가지 더 보자.

```
parseInt( 0.000008 ); // 0 ("0.000008" → "0")
parseInt( 0.0000008 ); // 8 ("8e-7" → "8")
parseInt( false, 16 ); // 250 ("false" → "fa")
parseInt( parseInt, 16 ); // 15 ("function.." → "f")

parseInt( "0x10" ); // 16
parseInt( "103", 2 ); // 2
```

parseInt(..)은 사실 예측 가능한 일관된 로직을 갖고 있다. 잘 사용하면 의미 있는 결과를 얻겠지만, 이상하게 사용해서 말도 안 되는 결과가 나왔다고 자바스크립트를 탓하지는 말자.

4.3.3 명시적 강제변환: * → 불리언

다음은 '비 불리언non-boolean → 불리언' 강제변환이다.

String(..), Number(..)도 그렇듯이 Boolean(..)은 (ToBoolean 추상 연산에 의한) 명시적인 강제변환 방법이다(앞에 new는 붙이지 않는다!).

```
var a = "0";
var b = [];
var c = {};

var d = "";
var e = 0;
var f = null;
var g;

Boolean( a ); // true
Boolean( b ); // true
Boolean( c ); // true
```

```
Boolean( d ); // false
Boolean( e ); // false
Boolean( f ); // false
Boolean( g ); // false
```

Boolean(..)은 분명히 명시적이지만, 그리 자주 쓰이지는 않는다.

+ 단항 연산자가 값을 숫자로 강제변환하는 것처럼 ! 부정negate 단항 연산자도 값을 불리언으로 명시적으로 강제변환한다. 문제는 그 과정에서 truthy, falsy까지 뒤바뀐다는 점이다. 그래서 일반적으로 자바스크립트 개발 시 불리언 값으로 명시적인 강제변환을 할 땐 !! 이중부정double-negate 연산자를 사용한다. 두 번째 ! 이 패리티를 다시 원상 복구하기 때문이다.[06]

```
var a = "0";
var b = [];
var c = {};

var d = "";
var e = 0;
var f = null;
var g;

!!a; // true
!!b; // true
!!c; // true

!!d; // false
!!e; // false
!!f; // false
!!g; // false
```

06 역자주_ ! 부정 단항 연산자는 표현식이 true이면 false, false이면 true로 결과를 반대로 뒤집기 때문에 불리언 타입으로 변환되는 것은 좋은데, 한 번 더 부정(negate)해야 원래 표현식의 true/false 값을 얻게 된다는 말입니다. 원문의 말이 다소 장황한 듯하여 간단히 부연합니다.

이 같은 ToBoolean 강제변환 모두 Boolean(..)이나 !!를 쓰지 않으면 if(..) 문 등의 불리언 콘텍스트에서 암시적인 강제변환이 일어난다. 여기서는 ToBoolean 강제변환의 원래 의도를 좀 더 명확히 구현하기 위해 값을 불리언으로 명시적인 강제변환을 했다.

자료 구조의 JSON 직렬화 시 true/false 값으로 강제변환하는 것도 명시적인 ToBoolean 강제변환의 일례다.

```
var a = [
  1,
  function(){ /*..*/ },
  2,
  function(){ /*..*/ }
];

JSON.stringify( a ); // "[1,null,2,null]"

JSON.stringify( a, function(key,val){
  if (typeof val == "function") {
    // 함수를 'ToBoolean' 강제변환한다.
    return !!val;
  }
  else {
    return val;
  }
} );
// "[1,true,2,true]"
```

자바 프로그래머라면 다음 구문에 익숙할 것이다.

```
var a = 42;

var b = a ? true : false;
```

삼항 연산자 ? : 는 표현식의 평가 결과에 따라 true 또는 false를 반환한다.

true/false 중 하나가 연산 결과로 도출된다는 점에서 명시적인 ToBoolean 강제변환의 모습과 닮았다.

그러나 여기엔 암시적 강제변환이 매복해 있다. a를 일단 불리언으로 강제변환해야 표현식 전체의 true/false 여부를 따져볼 수 있기 때문이다. 이전에도 이런 코드를 "명시적으로 암시적explicitly implicit"이라 했는데, 어쨌든 이런 코드는 무조건 쓰지 말자. 별로 얻을 것도 없고, 가면 뒤에 숨겨져 있는 불편한 진실이 있을 뿐이다.

Boolean(a)이나 !!a 같은 명시적 강제변환이 훨씬 좋다.

4.4 암시적 변환

암시적 강제변환은 부수 효과가 명확하지 않게 숨겨진 형태로 일어나는 타입 변환이다. 즉, 여러분들이 보기에 분명하지 않은 타입 변환은 모두 이 범주에 속한다.

명시적 강제변환의 목적(코드를 명확하게, 이해할 수 있게 작성함)은 분명하고 암시적 강제변환의 목적이 그 반대(코드를 더 이해하기 어렵게 만듦)라는 사실 또한 너무나 '분명'하다.

액면 그대로 받아들이다 보니 모든 오해와 노여움이 시작된 것 같다. "자바스크립트 강제변환"을 향한 대부분의 불평불만이 (제대로 이해하고 그러는진 모르겠지만) 바로 이 암시적 강제변환을 겨냥하고 있는 건 사실이다.

07 역자주_ 국내에는 『더글라스 크락포드의 자바스크립트 핵심 가이드(2008, 한빛미디어)』라는 제목으로 출간된 바 있습니다.
http://goo.gl/nY2kXY

『JavaScript : The Good Parts』[07]의 저자, 더글라스 크락포드Douglas Crockford는 많은 연설과 기고문에서 자바스크립트 강제변환은 반드시 피해야 한다고 역설한 바 있다. 그는 아마도 암시적 강제변환을 나쁜 것이라 생각한 모양이다. 하지만 그 자신이 직접 작성한 예제 코드에도 명시적/암시적 강제변환을 혼용했다! 사실 그가 주로 염려했던 것은 동등 연산(==)이 아니었나 싶은데, 이 장을 계속 읽다 보면 여러분도 알겠지만, 동등 연산은 강제변환 메커니즘의 일부에 지나지 않는다.

그럼, 암시적 강제변환은 정말 유해하고 위험한 것인가? 자바스크립트 언어 설계의 버그인가? 어떤 일이 있더라도 피해가는 게 상책인가?

"맞아요!" 독자 여러분 대다수가 열렬한 환호와 함께 대답하는 소리가 들리는 듯 싶다.

자, 이제 내 말을 귀담아 듣기 바란다.

암시적 강제변환이 무엇인지, 어떻게 활용할지, 무조건 "선량한 명시적 강제변환의 적"으로 몰아가지만 말고 조금 다른 시각에서 바라보자! 너무 편협하고 중요한 의미를 놓치고 있는 것 같아 안쓰럽다.

암시적 강제변환의 목적은, 중요한 내용으로부터 주의를 분산시켜 코드를 잡동사니로 가득 채워버리는, 장황함verbosity, 보일러플레이트boilerplate, 불필요한 상세 구현을 줄이는 것이다.

4.4.1 '암시적'이란?

자바스크립트 코드를 다루기 전에, 엄격한 타입 언어strongly typed language에서 이야기하는, 몇 가지 이론적인 의사 코드pseudocode부터 살펴보자.

```
SomeType x = SomeType( AnotherType( y ) )
```

임의의 타입, y 값을 SomeType 타입으로 변환하고자 한다. 이 언어에서는 y 값에

무관하게 곧바로 SomeType으로 변환할 수 없다. 어떤 중간 단계를 거쳐야 하는데, 이를테면 먼저 AnotherType으로 변환한 뒤 SomeType으로 최종 변환을 해야 한다.

자, 이 언어(또는 이 언어로 직접 작성한 정의)로 여러분이 원하는 바를 다음과 같이 직접 기술한다고 쳐보자.

```
SomeType x = SomeType( y )
```

코드를 이렇게 쓸 수 있다면 그 전에 보았던 코드의 중간 변환 단계(불필요한 "잡음")을 줄여 타입 변환을 단순화했다고 할 수 있지 않을까? 즉, y 값이 '원래 타입 → AnotherType 타입 → SomeType 타입'의 상세한 변환 과정을 거친다는 사실을 모든 사람들이 반드시 다 이해하고 고민해야 할까?

누군가는 이렇게 말할 것이다, 적어도 어떤 상황에선 그럴 수 있다고. 하지만 같은 논리로 나는, 단순화시킨 타입 변환 코드가, 언어 자체가 되었든 우리가 작성한 추상화 코드가 되었든 간에, 실제로 코드 가독성을 높이고 세세한 구현부는 추상화하거나 감추는 데 도움이 된다고 생각한다.

당연히 저 깊숙이 어딘가에선 중간 단계의 변환 과정이 일어나고 있을 것이다. 하지만 그 세부 내용이 적절히 숨겨져 있다면, 일반 연산을 하듯 y가 SomeType 타입으로 잘 변환되었겠거니 생각하고 지저분한 코드를 감출 수 있다.

완벽한 비유는 아니었겠지만, 어쨌든 이 장의 남은 부분에 걸쳐 이야기하려는 핵심은, 자바스크립트의 암시적 강제변환 역시 같은 이치로 우리가 작성하는 코딩에 도움이 될 수 있다는 것이다.

하지만 여기서 중요한 건, 내 말도 절대적이고 완벽한 건 아니라는 점이다. 분명히 어떤 상황에서는 코드 가독성 향상이라는 잠재적인 장점 이면에, 코드를 다치게

할 온갖 해로운 성분이 꿈틀대고 있을 수 있다. 따라서 버그로 도배한 코드를 짜려는 의도가 아니라면, 해로운 성분이 뭔지 인식하고 피하는 방법 또한 확실히 알아야 한다.

"어떤 장치가 A라는 유용한 기능이 있긴 하지만, 이를 남용/오용하면 Z라는 끔찍한 참사를 불러올 수 있다"라고 하면, 안전하게 가기 위해 전부 다 폐기하는 편이 낫다고 많은 개발자가 생각한다.

나는 여러분에게 "그런 식으로 안주하지 말라"고 당부하고 싶다. "아기를 목욕물과 함께 내다버릴 요량인가?"[08] 암시적 강제변환의 "나쁜 단면"을 보고 나서 전체가 다 나쁘다고 단정하진 말자. 만사가 "좋은 단면"도 있는 법이니, 부디 객관적으로 암시적 강제변환을 바라보고 수용하기 바란다.

4.4.2 암시적 강제변환: 문자열 ↔ 숫자

앞 부분에서 배웠던 '문자열 ↔ 숫자'의 명시적 강제변환을 이제부터 암시적 강제변환 버전으로 살펴보자. 그 전에 암시적 강제변환을 일으키는 몇몇 연산의 의미를 알아보자.

+ 연산자는 '숫자의 덧셈, 문자열 접합' 두 가지 목적으로 오버로드overload[09]된다. 자바스크립트 엔진은 연산자를 보고 어떤 연산을 해야 할지 어떻게 알까?

```
var a = "42";
var b = "0";

var c = 42;
var d = 0;
```

08 역자주_ 목욕물이 너무 많으면 아기가 사고로 익사할 수도 있겠지만, 그렇다고 아기랑 목욕물을 함께 내다버린다는 건 말도 안 된다는, 약간 코믹한 저자의 영어식 비유입니다.

09 역자주_ 연산자 오버로딩(operator overloading)은 `+`, `-`, `*`, `/` 등의 일반 연산을 위한 연산자가 다른 기능을 하도록 구현하는 것을 말합니다.

```
a + b; // "420"
c + d; // 42
```

결과값이 "420"으로 나올 때와 42로 나올 때는 어떤 차이가 있을까? 피연산자가 한쪽 또는 양쪽 모두 문자열인지 아닌지에 따라 + 연산자가 문자열 붙이기를 할지 결정한다고 보통 잘못 알고 있는 경우가 많다. 부분적으론 맞지만 실상은 그보다 더 복잡하다.

```
var a = [1,2];
var b = [3,4];

a + b; // "1,23,4"
```

피연산자 a, b 모두 문자열이 아니지만, 분명히 둘 다 문자열로 강제변환된 후 접합됐다. 무슨 일이 일어난 걸까?

다음 두 단락은 자세한 명세에 관한 내용이니 너무 머리가 아프면 건너뛰어도 좋다!

ES5 §11.6.1에 따르면 + 알고리즘(피연산자가 객체 값일 경우)은 한쪽 피연산자가 문자열이거나 다음 과정을 통해 문자열 표현형으로 나타낼 수 있으면 문자열 붙이기를 한다. 따라서 피연산자 중 하나가 객체(배열 포함)라면, 먼저 이 값에 ToPrimitive 추상 연산(9.1)을 수행하고, 다시 ToPrimitive는 number 콘텍스트 힌트를 넘겨 [[DefaultValue]] 알고리즘을 호출한다.

잘 뜯어보면 앞 단락의 작업이 ToNumber 추상 연산이 객체를 다루는 방법과 정확히 일치함을 알 수 있다. valueOf( )에 배열을 넘기면 단순 원시 값을 반환할 수 없으므로 바통은 toString( )으로 넘어간다. 그래서 두 배열은 각각 "1,2"와 "3,4"가 되고, +는 두 문자열을 붙여 최종 결과값은 "1,23,4"이 된다.

지저분한 상세는 나중에 다시 보고, 앞으로 돌아가 간단히 정리해보자. + 연산의 한쪽 피연산자가 문자열이면(또는 좀 전과 같이 어떤 과정을 거쳐 문자열이 되면) +는 문자열 붙이기 연산을 한다. 그 밖에는 언제나 숫자 덧셈을 한다.

잘 알려진 강제변환 함정이 있다. `[] + {}` 대 `{} + []`. 연산 결과는 각각 `"[object Object]"`와 0다.

그런데 이게 암시적 강제변환과 무슨 상관일까?

숫자는 공백 문자열 `""`와 "더하면" 간단히 문자열로 강제변환된다.

```
var a = 42;
var b = a + "";

b; // "42"
```

숫자 덧셈 +는 가환적commutative[10]이므로 `2 + 3`과 `3 + 2`은 결과가 같다. 문자열 연결 +는 대부분 가환적이지 않지만 `" "`는 특수한 경우라 가환적이다. 따라서 `a + " "`와 `" " + a`는 결과가 같다.

`a + " "`는 숫자를 문자열로 (암시적) 강제변환하는 아주 흔한 관용 코드다. 흥미로운 건, 그렇게 암시적 강제변환을 맹렬하게 비판하는 이들도 자신의 코드에선 (명시적인 강제변환 대신) 암시적 강제변환을 즐겨쓴다는 사실이다.

얼마나 욕을 먹든 간에, `a + " "` 같은 코드가 암시적 강제변환의 아주 멋진 사례라고 생각한다.

명시적 강제변환 `String(a)`에 비해 암시적 강제변환 `a + " "`에서는 한 가

10 역자주_ 연산의 순서를 바꾸어도 결과가 같는 것을 가환적(commutative)이라고 합니다. 실수 전체 집합에서는 덧셈과 곱셈 연산이 가환적입니다.

지 유의해야 할 기벽이 있다. ToPrimitive 연산 과정에서 a + " "는 a 값을 valueOf() 메서드에 전달하여 호출하고, 그 결과값은 ToString 추상 연산을 하여 최종적인 문자열로 변환된다. 그러나 String(a)는 toString()를 직접 호출할 뿐이다.

두 방법 모두 궁극적으로 변환된 문자열을 반환하지만, 평범한 원시 숫자 값이 아닌, 객체라면 결과값(문자열)이 달라질 수 있다!

```
var a = {
  valueOf: function() { return 42; },
  toString: function() { return 4; }
};

a + ""; // "42"

String( a ); // "4"
```

대부분의 경우, 여러분이 애써 어지러운 자료 구조와 복잡한 로직을 구사하려고 안간힘을 쓰지 않는다면 이런 함정 때문에 밤을 지샐 일은 없겠지만, valueOf(), toString() 메서드를 직접 구현한 객체가 있으면 강제변환 과정에서 결과값이 달라질 수 있으니 조심해야 한다.

방향을 바꾸어 '문자열 → 숫자' 암시적인 강제변환을 알아보자.

```
var a = "3.14";
var b = a - 0;

b; // 3.14
```

- 연산자는 숫자 뺄셈 기능이 전부이므로 a - 0은 a 값을 숫자로 강제변환한다. 자주 쓰이지는 않지만 a * 1이나 a / 1의 연산자 역시 숫자 연산만 하므로 마찬가지다.

객체 값에 - 연산을 하면? 이전의 +와 비슷하다.

```
var a = [3];
var b = [1];

a - b; // 2
```

두 배열은 우선 문자열로 강제변환된 뒤(toString( )로 직렬화) 숫자로 강제변환된다. 그리고 마지막엔 - 연산을 한다.

자, 지금까지 설명한, '문자열 ↔ 숫자'의 암시적인 강제변환이 여러분이 여태껏 감상한 호러 무비의 주인공만큼 흉측해 보이는가? 내 생각엔 그렇지 않다.

`b = String(a)`(명시적)과 `b = a + " "`(암시적)를 비교해보자. 둘 다 경우에 따라 유용하게 코드에 쓰일 수 있지만 자바스크립트 프로그램에선 후자를 훨씬 더 많이 쓴다. 이것만 보더라도 암시적 강제변환이 좋다, 나쁘다 하는 감정과는 무관하게 그 가치를 인정받았다는 걸 알 수 있다.

4.4.3 암시적 강제변환: 불리언 → 숫자

암시적 강제변환의 효용성은 복잡한 형태의 불리언 로직을 단순한 숫자 덧셈 형태로 단순화할 때 빛을 발한다. 범용적인 기법은 아니지만 특정 상황에선 기발한 해법이 될 수 있다.

```
function onlyOne(a,b,c) {
  return !!((a && !b && !c) ||
    (!a && b && !c) || (!a && !b && c));
}

var a = true;
var b = false;

onlyOne( a, b, b ); // true
```

```
onlyOne( b, a, b ); // true

onlyOne( a, b, a ); // false
```

onlyOne(. .)는 세 인자 중 정확히 하나만 true/truthy인지 아닌지를 확인하는 함수로 truthy 체크 시 암시적 강제변환을 하고 최종 반환 값을 포함한 다른 부분은 명시적 강제변환을 한다.

그런데 이런 식으로 4개, 5개, ... , 20개 인자를 처리해야 할 경우, 모든 비교 로직을 조합하여 코드를 구현한다는 게 상당히 어렵다는 사실을 금방 알 수 있다.

하지만 불리언 값을 숫자(명시적으로 0 또는 1)로 변환하면 의외로 문제가 쉽게 풀린다.

```
function onlyOne() {
  var sum = 0;
  for (var i=0; i < arguments.length; i++) {
    // falsy 값은 건너뛴다.
    // 0으로 취급하는 셈이다. 그러나 NaN은 피해야 한다.
    if (arguments[i]) {
      sum += arguments[i];
    }
  }
  return sum == 1;
}

var a = true;
var b = false;

onlyOne( b, a ); // true
onlyOne( b, a, b, b, b ); // true

onlyOne( b, b ); // false
onlyOne( b, a, b, b, b, a ); // false
```

onlyOne(..) 함수에서 for 루프 대신 간단히 ES5 reduce(..) 유틸리티를 써도 된다.

true/truthy를 숫자로 강제변환하면 1이므로 그냥 숫자를 모두 더한 것이 전부이고, sum += arguments[i]에서 암시적 강제변환이 일어난다. 인자 중 딱 하나만 true일 때 sum은 1이고, 그 외에는 1이 되지 않으므로 조건에 부합하는지 판단할 수 있다.

다음은 이 코드의 명시적 강제변환 버전이다.

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

먼저 !!arguments[i]로 인자 값을 true/false로 강제변환한다. 따라서 onlyOne("42", 0)처럼 비 불리언 값을 넘긴다 해도 문제없다(안 그랬다면 문자열을 붙이기를 했을 테고 로직은 틀어지게 됐을 것이다).

!!arguments[i]는 불리언 값이 확실하므로 Number(..)로 한 번 더 강제변환하여 0 또는 1로 바꾼다.

같은 기능이라면 명시적 강제변환이 더 "나은" 방법일까? 암시적 강제변환 코드에서 주석으로 달아놨던 NaN 함정은 잘 피해갔다. 하지만 어디까지나 판단은 여러분의 몫이다. 개인적인 생각으로는 암시적 강제변환 코드가 명시적 강제변환 코드보다 더 우아한 것 같다(undefined나 NaN을 인자로 넘기지만 않는다면). 후자는 조금 쓸데없이 장황한 느낌이 든다.

다시 한 번 말하지만, 여기서 논의한 모든 것들에 대한 판단은 여러분의 몫이다.

명시적이든 암시적이든, `onlyTwo(..)`, `onlyFive(..)`, ... 식으로 체크할 인자 개수를 다르게 하여 쉽게 변형할 수 있다. `&&`, `||` 표현식을 계속 추가하는 것에 비하면 엄청나게 편리하다. 이 예만 보면 강제변환은 대체로 매우 유용하다.

4.4.4 암시적 강제변환: * → 불리언

다음 주제는 불리언 값으로의 암시적 강제변환이다. 가장 평범하면서도 가장 골칫거리다.

암시적 강제변환은 어떤 값이 강제로 바뀌어야 하는 방향으로 사용할 때 발생한다는 것을 기억하자. 숫자, 문자열 연산에서 일어나는 강제변환은 쉽게 알아챌 수 있다.

다음은 불리언으로의 (암시적인) 강제변환이 일어나는 표현식을 열거한 것이다.

1. `if (..)` 문의 조건 표현식
2. `for ( .. ; .. ; .. )`에서 두 번째 조건 표현식
3. `while (..)` 및 `do..while(..)` 루프의 조건 표현식
4. `? :` 삼항 연산 시 첫 번째 조건 표현식
5. `||` (논리 OR) 및 `&&` (논리 AND)의 좌측 피연산자(테스트 표현식 역할을 한다 - 곧 설명한다!)

이런 콘텍스트에서 불리언 아닌 값이 사용되면, 이 장 앞 부분에서 언급했던 `ToBoolean` 추상 연산 규칙에 따라 일단 불리언 값으로 암시적 강제변환된다.

몇 가지 예시를 보자.

```
var a = 42;
var b = "abc";
```

```
var c;
var d = null;

if (a) {
  console.log( "넵" ); // 넵
}

while (c) {
  console.log( "절대 실행될 리 없지!" );
}

c = d ? a : b;
c; // "abc"

if ((a && d) || c) {
  console.log( "넵" ); // 넵
}
```

예제의 모든 콘텍스트에서 비 불리언 값은 조건 표현식을 평가하기 위해, 그와 동등한 불리언 값으로 강제변환된다.

4.4.5 &&와 || 연산자

||(논리 OR) 및 &&(논리 AND) 연산자는 여러분이 사용해온 대부분의 프로그래밍 언어에서 쉽게 찾아볼 수 있다. 그래서인지 자연스레 작동 방식도 별 차이가 없을 거라 생각한다.

그러나 여기에 아주 중요하면서도 미묘한 차이가 있다는 사실은 잘 모른다.

솔직히 이것들을 "논리 —— 연산자"라고 부르는 건 명칭만으론 이 연산자들의 기능을 나타내기에 한참 부족하기 때문에 부적절하다고 본다. 나더러 작명을 하라고 한다면 (더 어설퍼 보일 수도 있지만) "선택selector 연산자" 또는 더 완전하게는 "피연산자 선택 연산자" 정도로 할 것이다.

이유는 자바스크립트에서 이 두 연산자는 다른 언어와 달리 실제로 결과값이 논리

값(불리언)이 아니기 때문이다.

그럼 대체 결과값은? 두 피연산자 중 한쪽 (오직 한쪽의) 값이다. 즉, 두 피연산자의 값들 중 하나를 선택한다.

ES5 §11.11에 명세님이 가라사대,

> && 또는 || 연산자의 결과값이 반드시 불리언 타입이어야 하는 것은 아니며, 항상 두 피연산자 표현식 중 어느 한쪽 값으로 귀결된다.

예를 들어보자.

```
var a = 42;
var b = "abc";
var c = null;

a || b; // 42
a && b; // "abc"

c || b; // "abc"
c && b; // null
```

어랏, 뭔가 이상하다? 생각해보자. C, PHP 같은 언어라면 결과값은 당연히 true 아니면 false다. 그런데 자바스크립트에서는(그리고 파이썬, 루비도) 결과값이 피연산자 값이다.

||, && 연산자는 우선 첫 번째 피연산자(a, c)의 불리언 값을 평가한다. 피연산자가 비 불리언 타입이면(예제 코드처럼), 먼저 ToBoolean로 강제변환 후 평가를 계속한다.

|| 연산자는, 그 결과가 true면 첫 번째 피연산자(a, c) 값을, false면 두 번째 피연산자(b) 값을 반환한다.

이와 반대로 && 연산자는, true면 두 번째 피연산자(b)의 값을, false면 첫 번째

피연산자(a, c)의 값을 반환한다.

||, && 표현식의 결과값은 언제나 피연산자의 값 중 하나이고, (필요 시 강제변환된) 평가 결과가 아니다. c && b에서 c는 null이므로 falsy 값이다. 하지만 && 표현식은 평가 결과인 false(falsy를 강제변환)가 아니라 c 자신의 값, null로 귀결된다.

앞에서 이들을 "피연산자 선택 연산자"라고 부르려고 했던 이유를 이제 이해하겠는가?

어렵다면 다음과 같이 생각하자.

```
a || b;
// 대략 다음과 같다.
a ? a : b;

a && b;
// 대략 다음과 같다.
a ? b : a;
```

결과가 같아서 a || b가 a ? a : b와 "대략 같다"라고 표현했지만 의미상 차이는 있다. a ? a : b의 a가 만약 복잡한 표현식(이를테면, 함수 호출 등의 부수 효과를 가진 표현식)이라면, a 표현식은 (처음 평가 결과가 truthy라면) 두 번 평가될 가능성이 있다. 반면, a || b에서 a는 단 한 번만 평가하고 그 결과는 테스트 수행 시 강제변환과 최종 결과값(해당 시) 양쪽 모두 사용된다. a && b와 a ? b : a도 마찬가지다.

아직 확실히 이해되지 않는 독자를 위하여 이런 특성을 무지 잘 활용한 코드를 예시한다.

```
function foo(a,b) {
  a = a || "hello";
  b = b || "world";

  console.log( a + " " + b );
}
```

```
foo(); // "hello world"
foo( "오 마이", "갓!" ); // "오 마이 갓!"
```

a = a || "hello" 같은 패턴의 관용 코드(C# "null 접합 연산자coallescing operator"의 자바스크립트 버전이라고 하는 사람들도 있다)는 a 값이 없으면 (아니면 예기치 않게 a가 falsy 값이면) "hello!"을 a에 디폴트 값으로 할당한다.

하지만 조심하자!

```
foo( "바로 이거야!", "" ); // "바로 이거야! world" <-- 어이쿠!
```

뭐가 문제일까? 두 번째 인자 "" 은 falsy 값이므로 b = b || "world"에서 b에는 디폴트 값 "world"가 할당된다. 물론 이 프로그래머의 의도는 b를 "" 로 만들려는 것이었겠지만!

|| 연산자의 이러한 사용 패턴은 매우 흔하고 제법 쓸 만하지만 falsy 값은 무조건 건너뛸 경우에만 사용해야 한다. 그렇지 않으면 조건 평가식을 삼항 연산자(?:)로 더욱 명시적으로 지정해야 한다.

디폴트 값을 할당하는 관용 코드는 너무 일반적인(그리고 유용하다!) 나머지, 심지어 자바스크립트 강제변환을 매도하는 사람들조차도 사용할 정도다.

그렇다면 &&는?

개발자가 직접 손으로 코딩하기보다는 자바스크립트 압축기minifier에서 더 많이 쓰는, 또 다른 관용 코드가 있다. && 연산자는 첫 번째 피연산자의 평가 결과가 truthy일 때에만 두 번째 피연산자를 "선택"한다고 했는데, 이런 특성을 "가드 연산자guard operator"라고 하여, 첫 번째 표현식이 두 번째 표현식의 "가드" 역할을 한다.

```
function foo() {
  console.log( a );
```

```
}

var a = 42;

a && foo(); // 42
```

a 평가 결과가 truthy일 때에만 foo( )가 호출된다. 평가 결과가 falsy면 a && foo( ) 표현식은 그 자리에서 조용히 실행을 멈추고(그래서 "단락 평가short circuiting"라고 한다) foo( )는 호출되지 않는다.

대부분의 경우 이런 식의 코딩보다는 if (a) { foo( ); }처럼 작성하는 사람들이 더 많을 것이다. 하지만 JS 압축기는 코드를 최대한 쥐어짜야 하므로 a && foo( ) 같이 처리한다. 그래서 만약 이렇게 생긴 코드를 해독할 일이 있으면, 그 의미와 이유를 잘 알고 있어야 한다.

좋다, 암시적 강제변환이라는 재료를 기꺼이 반죽에 털어 넣을 준비가 됐다면 이제부터 ||, &&는 여러분에게 멋진 묘기를 보여줄 것이다.

a = b || "어쩌구", a && b( ) 같은 관용 코드는 단락 평가에 근거한다.

||, && 연산 결과가 실제로 true/false가 아니라는 사실이 여러분에게 불편할 수도 있다. 지금까지 여러분이 작성해온 a && (b || c) 같은 복합 논리 표현식이 포함된 if 문이나 for 루프는 도대체 어떻게 작동했던 것일까 궁금할 것이다.

걱정 마시라! 하늘이 무너질 일은 없다. 여러분의 코드는 (아마도) 계속 문제 없이 돌아갈 것이다. 다만 먼저 복합 표현식이 평가된 다음 불리언으로 암시적 강제변환이 일어난다는 사실을 예전엔 몰랐을 뿐이다.

```
var a = 42;
var b = null;
```

```
var c = "foo";

if (a && (b || c)) {
  console.log( "넵" );
}
```

이 코드는 여러분이 짐작한 그대로 작동하지만 절묘한 게 한 가지 있다. `a && (b || c)` 표현식의 실제 결과는 `true`가 아닌 "`foo`"다. `if` 문은 이 "`foo`"를 불리언 타입으로 강제변환하여 `true`로 만든다.

알겠는가? 당황할 필요가 없다. 여러분이 작성해온 코드는 안전하다. 무슨 일을 어떻게 하는지, 내면의 진실에 눈을 뜨게 되었을 뿐이다.

그리고 여러분은 이런 코드가 암시적 강제변환을 사용하고 있다는 것을 깨달았다. 여러분이 아직도 "(암시적) 강제변환 사용 금지 클럽"의 열성 회원이라면, 아마 다음과 같이 명시적인 평가 표현식을 작성해야 잘했다고 칭찬받을 것이다.

```
if (!!a && (!!b || !!c)) {
  console.log( "yep" );
}
```

신의 가호가 함께 하시길! 아, 놀려서 미안.

4.4.6 심볼의 강제변환

지금까지 다룬 명시적/암시적 강제변환은 코드 가독성이 문제일 뿐 가시적인 결과의 차이는 거의 없었다.

그러나 (간단히 살펴보겠지만) ES6부터 새로 등장한 심볼 탓에 강제변환 체계에 조심해야 할 함정이 하나 더 늘어났다. 지면상의 이유로 이 책에는 수록하지 못한 문제들 때문에 '심볼 → 문자열' 명시적 강제변환은 허용되나 암시적 강제변환은 금

지되며 시도만 해도 에러가 난다.

```
var s1 = Symbol( "좋아" );
String( s1 ); // "Symbol(좋아)"

var s2 = Symbol( "구려" );
s2 + ""; // TypeError
```

심볼 값은 절대 숫자로 변환되지 않지만(양방향 모두 에러가 난다), 희한하게도 불리언 값으로는 명시적/암시적 모두 강제변환(항상 true다)이 가능하다.

일관성이 있어야 배우기 쉽고 예외가 늘어날수록 다루기가 까다로워지는 법이지만, 새로운 ES6 심볼 값을 다루거나 강제변환할 때에는 유의해야 한다.

하지만 좋은 소식이 있다면, 심볼 값을 강제변환할 일은 아마도 정말 드물 거라는 사실! 원래 만들어진 의도대로만 사용한다면(3장 네이티브 참고) 강제변환할 필요는 거의 없을 것이다.

4.5 느슨한/엄격한 동등 비교

느슨한 동등 비교Loose Equals는 == 연산자를, 엄격한 동등 비교Strict Equals는 === 연산자를 각각 사용한다. 두 연산자 모두 두 값의 "동등함equality"을 비교하지만 "느슨함"과 "엄격함"이라는 아주 중요한 차이점 있고, 특히 "동등함"의 판단 기준이 다르다.

많은 이들이 종종 "==는 값의 동등함을, ===는 값과 타입 모두의 동등함을 비교한다"고 오해한다. 그럴 듯하지만 정확하진 않다. 엄청나게 많은 자바스크립트 책들에 그렇게 씌어있지만, 불행하게도 다 틀렸다.

정확한 정의를 내리겠다. "동등함의 비교 시 ==는 강제변환을 허용하지만 ===는

강제변환을 허용하지 않는다."

4.5.1 비교 성능

전자의 (부정확한) 정의와 후자의 (정확한) 정의를 잠시 음미해보자.

전자에 의하면 ===는 타입까지 체크하므로 ==에 비해 할 일이 많다. 후자에 따르면 오히려 타입이 다를 경우 강제변환을 해야 하므로 ==가 더 할 일이 많다.

하지만 마치 ==이 ===보다 눈에 띄게 처리가 더뎌서 어떤 식으로든 성능에 영향을 미치는 것처럼 생각하진 말자. 강제변환 시 처리 시간이 약간 더 소요되긴 하지만, 불과 몇 마이크로 초 단위의 차이일 뿐이다(그렇다, 1 마이크로 초는 백만 분의 1초다!).

타입이 같은 두 값의 동등 비교라면, ==와 ===의 알고리즘은 동일하다. 엔진의 내부 구현 방식은 조금씩 다를 수도 있지만, 기본적으로 하는 일은 같다.

타입이 다른 두 값의 동등 비교에서 성능은 중요한 포인트가 아니다. 여러분이 자문해봐야 할 사항은 비교 과정에서 강제변환의 개입 여부다.

강제변환이 필요하다면 느슨한 동등 연산자(==)를, 필요하지 않다면 엄격한 동등 연산자(===)를 사용하자.

어차피 ==든 ===든 피연산자의 타입을 체크하는 건 매한가지다. 다른 점은 타입이 다를 때 이후 처리 로직이다.

4.5.2 추상 동등 비교

== 연산자 로직은 ES5 §11.9.3 "추상적 동등 비교 알고리즘The Abstract Equality Comparison Algorithm"에 상술되어 있다. 잡다한 항목이 나열되어 있지만 알고리즘 자체는 간단하다. 모든 가능한 타입별 조합마다 (필요 시) 강제변환을 어떻게 수행하는지 그 방법이 적혀 있다.

(암시적) 강제변환을 (너무 복잡하고 결함투성이라) 곱지 않게 보는 사람들이 가장 일반적으로 비난하는 대상이 바로 이 "추상적 동등" 관련 규칙이다. 너무 난해하여 개발자들이 실제로 배우고 사용하기에 직관적이지 못할 뿐만 아니라, 자바스크립트 코드의 가독성을 향상시키지는 못할망정 버그만 양산하는 규칙들이라고 주장한다. 하지만 이런 부정적인 생각은 명세를 읽는 독자가 알고리즘(코드)만 전문으로 개발(물론 읽고 이해할 수 있어야 함!)하는 노련한 개발자여야 한다는 잘못된 가정에 근거한 것이다. 실제로 "추상적 동등" 규칙은 쉬운 용어로 평이하게 기술되어 있다. 의심 나면 내 말을 믿고 ES5 §11.9.3를 직접 한번 읽어보자. 의외로 읽을 만하다는 걸 알고 깜짝 놀랄 것이다.

일단 첫째 항(11.9.3.1)에는 이렇게 씌어있다. 비교할 두 값이 같은 타입이면, 누구나 예상하듯이 값을 식별하여 간단히, 자연스럽게 견주어본다. 예컨대, 42와 동등한 값은 42뿐이고, " abc "와 동등한 값은 " abc " 뿐이다.

다음 예외는 사소하나마 상식을 벗어나므로 주의해야 한다.

- `NaN`은 그 자신과도 결코 동등하지 않다(2장 값 참고).
- `+0`와 `-0`는 동등하지 않다(2장 값 참고).

§11.9.3.1의 마지막 항목에서는, 객체(함수와 배열 포함)의 느슨한 동등 비교에 대해, 두 객체가 정확히 똑같은 값에 대한 레퍼런스일 경우에만 동등하다고 기술되어 있다. 여기서 강제변환은 일어나지 않는다.

§11.9.3.6을 보면 엄격한 동등 비교 `===` 역시 두 객체 값 비교 방법에 대해 똑같이 기술되어 있다. 객체의 동등 비교에 있어서 `==`와 `===`의 로직이 똑같다는 사실은 거의 알려져 있지 않다.[11]

§11.9.3 알고리즘 나머지 부분에서는, 타입이 다른 두 값을 느슨한 동등 비교(`==`)시, 한쪽 또는 양쪽 피연산자에서 암시적 강제변환을 어떻게 해야 하는지 씌어 있

11 역자주_ ES5 명세 §11.9.3의 1번 f항과 §11.9.6.7의 7번의 원문을 찾아보면 정말 글자 단위로 정확히 같습니다. Return true if x and y refer to the same object. Otherwise, return false(x와 y가 같은 객체를 가리키고 있으면 `true`, 아니면 `false`를 반환한다).

다. 결과적으로 두 값의 타입을 일치시켜 간단히 값만 보고 비교하기 위함이다.

짐작하겠지만, 느슨한 비동등 연산자 !=의 결과값은 == 연산자의 동등 비교 수행 후 그 결과를 그대로 부정한negate 값이다. 엄격한 비동등 연산자 !== 역시 마찬가지다.

비교하기: 문자열 → 숫자

앞서 보았던 문자열/숫자 예제로 돌아가 == 강제변환을 살펴보자.

```
var a = 42;
var b = "42";

a === b; // false
a == b; // true
```

예상대로 a === b는 false다. 강제변환이 허용되지 않는 데다 42와 "42"는 그냥 봐도 다른 값이기 때문이다.

하지만 느슨한 동등 비교 a == b에서는 피연산자의 타입이 다르면, 비교 알고리즘에 의해 한쪽 또는 양쪽 피연산자 값이 알아서 암시적으로 강제변환된다.

그런데 정확히 어떻게 강제변환이 일어나는 걸까? 42가 문자열로 바뀌어 a가 되는 걸까 아니면 "42"가 숫자로 바뀌어 b가 되는 걸까?

ES5 §11.9.3.4-5 원문을 보자.

1. Type(x)가 Number고 Type(y)가 String이면, x == ToNumber(y) 비교 결과를 반환한다.
2. Type(x)가 String이고 Type(y)가 Number면, ToNumber(x) == y 비교 결과를 반환한다.

이 책에서 숫자number, 문자열string 등으로 표현한 원시 값이 명세에는 타입을 가리키는 Number, String 등의 정규 용어로 기술되어 있다. Number가 대문자로 시작한다고 Number ( ) 네이티브 생성자와 헷갈리지 말자. 사실 같은 의미이므로 타입명을 대문자로 시작하든 말든 상관없다.[12]

명세를 보니 비교 전 먼저 " 42 " 값이 숫자로 강제변환된다는 것을 분명히 알 수 있다. 강제변환은 이미 앞서 설명했던 ToNumber 추상 연산이 담당하고, 결과값은 42이므로 두 42 값은 명백히 동등하다.

비교하기: * → 불리언

어떤 값을 true/false와 직접 비교하려고 하면, 느슨한 동등 비교(==)의 숨겨진, 가장 끔찍한 강제변환 함정에 빠지게 될 것이다.

```
var a = "42";
var b = true;

a == b; // false
```

잠깐, 이게 뭐지? " 42 " 는 truthy 값이니(이 장 앞 부분 참고) == 비교하면 true 아닌가? 결과가 반대인 이유는 단순하면서도 꽤 까다롭다. 참으로 오해하기 쉬운 문제인데 많은 개발자가 관심을 갖고 제대로 이해하려고 하지 않는다.

ES5 §11.9.3.6-7를 인용한다.

1. Type (x)이 불리언이면 ToNumber (x) == y의 비교 결과를 반환한다.
2. Type (y)이 불리언이면 x == ToNumber (y)의 비교 결과를 반환한다.

자, 다음 코드를 보자.

12 역자주_ 원서에는 원시 값 타입을 number, string으로 표시했지만, 본 역서에서는 숫자, 문자열로 옮겼으므로 어디까지나 영어로 원서를 직접 읽는 사람들에게만 의미있는 내용입니다.

```
var x = true;
var y = "42";

x == y; // false
```

Type(x)은 불리언이므로 ToNumber(x) → 1로 강제변환된다. 따라서 1 == "42"이 되는데 타입이 상이하므로 (재귀적으로) 알고리즘을 수행한다. 결국 "42"는 42로 바뀌어 1 == 42 → false가 된다.

x, y 순서를 바꾸어도 결과는 같다.

```
var x = "42";
var y = false;

x == y; // false
```

이번에는 Type(y)가 불리언이므로 ToNumber(y)는 0이고, "42" == 0 → (재귀적으로) 42 == 0 → false다.

결론적으로 "42"는 == true도, == false도 아니다. 언뜻 보면 말도 안 되는 소리다. 어떻게 truthy도 falsy도 아닌 값이 있단 말인가?

하지만 바로 이게 문제다! 여러분은 지금 완전히 틀린 질문을 하고 있다. 여러분의 잘못은 아니고 여러분의 두뇌가 잠시 장난을 친 것이다.

"42"는 분명 truthy 값이지만 "42" == true는 여러분의 두뇌가 어떻게 반응하든 상관없이 불리언 평가나 강제변환을 전혀 하지 않는다. "42"가 불리언(true)으로 강제변환되는 것이 아니라, 도리어 true가 1로 강제변환되고, 그 후 "42"가 42로 강제변환된다.

이런 방식이 맘에 들지 않더라도 어쨌든 ToBoolean은 전혀 개입하지 않으며, "42" 값 자체의 truthy/falsy 여부는 == 연산과는 전혀 무관하다!

==의 비교 알고리즘이 각 타입 조합별로 어떻게 작동하는가가 중요하다. 아까도 보았지만 ==의 피연산자 한쪽이 불리언 값이면 예외 없이 그 값이 먼저 숫자로 강제변환된다.

나만 이상한 건가 싶다면, 절대 그렇지 않다! 어떠한 일이 있더라도, 절대로, 두 번 다시 == true, == false 같은 코드는 쓰지 말라고 개인적으로 강권하고 싶다.

하지만 여기서 쓰지 말라고 한 연산자는 ==이지, ===이 아니다. === true, === false는 강제변환을 허용하지 않기에 ToNumber 강제변환 따위는 신경쓰지 않아도 된다.

```
var a = "42";

// 나빠 (실패한다!):
if (a == true) {
  // ..
}

// 이것도 나빠 (실패한다!):
if (a === true) {
  // ..
}

// 그럴 듯 하군 (암시적으로 작동한다):
if (a) {
  // ..
}

// 훨씬 좋아 (명시적으로 작동한다):
if (!!a) {
  // ..
}

// 이것도 좋아 (명시적으로 작동한다):
if (Boolean( a )) {
  // ..
}
```

그냥 앞으로 == true, == false("직접 불리언 값과 느슨한 동등 비교하기") 같은 코드를 안 쓰면 여러분은 골치 아픈 truthy/falsy 문제에서 해방될 것이다.

비교하기: null → undefined

null과 undefined 간의 변환은 느슨한 동등 비교 ==이 암시적 강제변환을 하는 또 다른 예다. ES5 §11.9.3.2-3를 인용한다.

1. x가 null이고 y가 undefined면 true를 반환한다.
2. x가 undefined이고 y가 null면 true를 반환한다.

null과 undefined를 느슨한 동등 비교(==)하면 서로에게 타입을 맞춘다(강제변환한다). 언어 전체를 눈 씻고 찾아봐도 다른 값은 끼어들 여지가 없다.

즉, null과 undefined는 느슨한 동등 비교 시 상호 간의 암시적인 강제변환이 일어나므로 비교 관점에서 구분이 되지 않는 값으로 취급되는 것이다.

```
var a = null;
var b;

a == b; // true
a == null; // true
b == null; // true

a == false; // false
b == false; // false
a == ""; // false
b == ""; // false
a == 0; // false
b == 0; // false
```

'null ↔ undefined' 강제변환은 안전하고 예측 가능하며, 어떤 다른 값도 비교

결과 잘못된 긍정false positive을 할 가능성이 없다.[13] null과 undefined을 구분되지 않는 값들로, 결국 동일한 값으로 취급하는 강제변환은 권장하고 싶다.

예를 들면,

```
var a = doSomething();

if (a == null) {
  // ..
}
```

a == null의 평가 결과는 doSomething( )이 null이나 undefined를 반환할 경우에만 true, 이외의 값이 반환되면 (심지어 0, false, " " 등의 다른 falsy한 값이 넘어와도) false다.

강제변환이 내키지 않아 명시적으로 체크하겠다고 다음처럼 작성한 코드는 (내 주관으론) 쓸데없이 흉하기만 하다(그리고 사소하지만 성능도 약간 떨어진다!).

```
var a = doSomething();

if (a === undefined || a === null) {
  // ..
}
```

a == null 같은 코드는 가독성 좋고 안전하게 작동하는 암시적 강제변환의 일례다.

비교하기: 객체 → 비객체

객체/함수/배열과 단순 스칼라 원시 값(문자열, 숫자, 불리언)의 비교는 ES5 § 11.9.3.8-9에서 다룬다.

13 역자주_ 즉, null과 undefined 자신들끼리만 비교 결과가 true이므로, 이외의 값들과 비교했을 때 결과값이 true일 가능성은 없습니다.

1. Type(x)가 String 또는 Number고 Type(y)가 객체라면, x == ToPrimitive(y)의 비교 결과를 반환한다.
2. Type(x)가 Object이고 Type(y)가 String 또는 Number라면, ToPrimitive(x) == y의 비교 결과를 반환한다.

앞서 인용했던 §11.9.3.6-7에서 이미 Boolean 피연산자를 Number 타입으로 강제변환하는 문제를 다루었으므로 여기서는 String, Number만 언급되어 있고 Boolean은 빠져 있다.

```
var a = 42;
var b = [ 42 ];

a == b; // true
```

[42]는 ToPrimitive 추상 연산 결과, "42"가 된다. 그리고 "42" == 42 → 42 == 42이므로 a, b는 동등하다.

짐작하겠지만, 이 장 앞 부분(toString(), valueOf())에서 살펴보았던 ToPrimitive 추상 연산의 기벽은 여기에도 해당된다. 커스텀 valueOf() 메서드를 구현해야 할 정도로 복잡한 자료 구조라면 동등 비교 목적으로 단순 값을 제공하는 것이 꽤 유용할 수 있다.

3장 네이티브에서 배운 "언박싱"을 상기하자. 원시 값을 감싼 객체 래퍼(예: new String("abc"))를 한 꺼풀 벗겨 원시 값("abc")을 반환하는 과정이다. 언박싱은 == 알고리즘의 ToPrimitive 강제변환과 관련되어 있다.

```
var a = "abc";
var b = Object( a ); // 'new String( a )'와 같다.

a === b; // false
a == b; // true
```

b는 ToPrimitive 연산으로 " abc "라는 단순 스칼라 원시 값으로 강제변환되고("언박싱"으로 벗겨지고), 이 값은 a와 동일하므로 a == b는 true가 맞다.

하지만 항상 그런 것은 아니다. == 알고리즘에서 더 우선하는 규칙 때문에 그렇지 않은 경우들도 있다.

```
var a = null;
var b = Object( a ); // 'Object()'와 같다.
a == b; // false

var c = undefined;
var d = Object( c ); // 'Object()'와 같다.
c == d; // false

var e = NaN;
var f = Object( e ); // 'new Number( e )'와 같다.
e == f; // false
```

null과 undefined는 객체 래퍼가 따로 없으므로 박싱할 수 없다. 그래서 Object(null)는 Object( )로 해석되어 그냥 일반 객체가 만들어진다.

NaN은 해당 객체 래퍼인 Number로 박싱되지만, ==를 만나 언박싱되면 결국 조건식은 NaN == NaN이 되어 (NaN은 자기 자신과도 같지 않으므로) 결과는 false다(2장 값 참고).

4.5.3 희귀 사례

지금까지 느슨한 동등 비교 == 이면의 (말 되는 것 같다가도 사람을 놀래키는) 암시적 강제변환에 대해 아주 자세히 살펴보았다. 이 절에서는 여러분들이 강제변환 버그를 피해가도록 하기 위해서, 그 중에서도 가장 골치 아프고 쓰지 말아야 할, 희귀 사례corner cases를 골라 소개할 것이다.

먼저, 내장 네이티브 프로토타입을 변경하면 어떤 참사가 빚어지는지 살펴보자.

알 박힌 숫자 값

```
Number.prototype.valueOf = function() {
  return 3;
};

new Number( 2 ) == 3; // true
```

2 == 3 비교는 이 예와 무관하다. 2, 3이 둘 다 이미 원시 숫자 값이고 곧바로 비교가 가능하므로 Number.prototype.valueOf( ) 내장 메서드는 호출되지 않는다. 그러나 new Number(2)는 무조건 ToPrimitive 강제변환 후 valueOf( )를 호출한다.

정말 고약한 냄새가 나지 않는가? 감히 이런 코드는 흉내도 내지 말지어다. 이런 코드를 짤 수 있기 때문에 강제변환과 ==이 욕을 먹고 있지만, 그렇더라도 불만의 근거를 여기에서 찾는 건 잘못이다. 끔찍한 일을 저지를 수 있는 자바스크립트가 나쁜 것이 아니라, 그런 짓을 저지르는 개발자가 나쁜 것이다. "프로그래밍 언어는 개발자를 지켜줘야 해" 식의 빗나간 감상에 빠지지 말자.

다음 사례는 아까보다 훨씬 까다롭다.

```
if (a == 2 && a == 3) {
  // ..
}
```

a가 동시에 2가 되고 3이 된다는 게 말이 되나 싶겠지만, "동시에"란 전제부터가 틀렸다. 엄밀히 말해, 두 표현식 중 a == 2가 a == 3보다 먼저 평가된다.

a.valueOf( )에 부수 효과를 주면 어떨까? 이를테면 다음과 같이 처음 호출하면 2, 두 번째 호출하면 3을 반환하는 식으로 말이다.

```
var i = 2;
```

```
Number.prototype.valueOf = function() {
  return i++;
};

var a = new Number( 42 );

if (a == 2 && a == 3) {
  console.log( "이런, 정말 되는구만!" );
}
```

이런 코드는 그 자체로 공해니 생각조차 말고 강제변환을 비난하는 근거로 제시하지도 말자. 남용될 소지가 있다고 하여 비난받아 마땅한 정당한 근거가 되는 것은 아니다. 말도 안 되는 장난은 피하면 그만이고, 올바르게, 적절하게 강제변환을 이용하자.

Falsy 비교

==의 암시적 강제변환을 힐난하는 사람들은 `falsy` 값 비교에 관한 이상한 로직을 종종 거론한다.

`falsy` 값 비교에 관한 희귀 사례 목록을 보면서 정상과 비정상을 구별해보자.

```
"0" == null; // false
"0" == undefined; // false
"0" == false; // true — 어이쿠!
"0" == NaN; // false
"0" == 0; // true
"0" == ""; // false

false == null; // false
false == undefined; // false
false == NaN; // false
false == 0; // true — 어이쿠!
false == ""; // true — 어이쿠!
false == []; // true — 어이쿠!
false == {}; // false

"" == null; // false
```

```
"" == undefined; // false
"" == NaN; // false
"" == 0; // true — 어이쿠!
"" == []; // true — 어이쿠!
"" == {}; // false

0 == null; // false
0 == undefined; // false
0 == NaN; // false
0 == []; // true — 어이쿠!
0 == {}; // false
```

여기에 열거된 24개의 비교 중 17개는 이치에 맞고 예측 가능하다. 예컨대, `""`와 `NaN`은 전혀 동등할 만한 값들이 아니며 실제로도 느슨한 동등 비교 시 강제변환되지 않는다. 한편, `"0"`과 `0`은 그냥 봐도 같은 값이며 느슨한 동등 비교 시 강제변환된다.

하지만 여기서 "어이쿠!"라고 주석을 붙인 7개의 비교는 잘못된 긍정false positive이며, 개발자를 뜬눈으로 지새우게 할 소지가 다분하다. `""`와 `0`은 분명히 다른 값이며 같은 값으로 취급할 경우 또한 거의 없기 때문에 상호 강제변환은 문제가 있다. 참고로 7개 비교 중 잘못된 부정false negative은 하나도 없다.[14]

말도 안 되는...

여기서 끝나지 않는다. 더 심각한 강제변환 사례도 있다.

```
[] == ![]; // true
```

오오, 또 다른 차원의 경이로움이다!? 분명히 `truthy`와 `falsy`의 비교가 아닌가

14 역자주_ 7개 비교 표현식 모두 평가 결과가 `false`일 것 같지만 실제로는 `true`입니다. 이와 반대로 `true`일 것 같은데 실제 결과가 `false`인 것은 하나도 없습니다.

싶은데, 결과는 놀랍게도 true다. 어떤 값이 동시에 truthy도 되고 falsy도 될 수 있을까?

하지만 겉보기만 그럴 뿐 실제로 벌어지는 일은 다르다. 하나씩 파헤쳐 보자. ! 단항 연산자를 기억하는가? ToBoolean으로 불리언 값으로 명시적 강제변환을 하는 (그리고 패리티를 전부 뒤집는) 연산자다. 따라서 [] == ![] 이전에 이미 [] == false로 바뀐다. 24개 비교 목록 중 비슷한 코드(false == [])가 있으니 왜 결과가 이렇게 나왔는지 짐작이 간다.

다음 사례도 비슷하다.

```
2 == [2]; // true
"" == [null]; // true
```

ToNumber를 설명할 때 앞에서 이야기했지만, 우변의 [2], [null]은 ToPrimitive가 강제변환을 하여 좌변과 비교 가능한 원시 값(각각 2와 " ")으로 바꾼다. 배열의 valueOf( ) 메서드는 배열 자신을 반환하므로 강제변환 시 배열을 문자열화한다.

따라서 첫째 줄의 [2]는 " 2 "가 되고 다시 ToNumber 강제변환을 거쳐 2가 된다. [null]은 바로 " "이 된다.

결국 2 == 2와 " " == " "로 해석된다.

이런 결과가 체질적으로 마음에 들지 않는다면, 사실 여러분이 생각하는 것과는 달리 불만의 원인은 강제변환이 아니다. '배열 → 문자열' 강제변환 시 ToPrimitive이 수행하는 로직이 싫은 것이다. 즉, [2].toString( )이 " 2 "를 반환하고, [null].toString( )이 " "를 반환하는 형태 자체가 못마땅한 것이다.

그럼 과연 어떻게 문자열로 강제변환해야 맞는 걸까? 내 생각엔 [2] → " 2 " 만큼 적절한 변환은 아마 없을 것 같다. 아니면 " [2] " 이렇게? 하지만 콘텍스트가 달라지면 아주 이상하게 작동할지 모른다!

또 String(null) → " null " 이니 String([null]) 역시 " null " 이어야 맞지 않느냐고 이의를 제기할 수도 있다. 타당한 주장이지만, 바로 여기에 문제가 숨어 있다.

여기서 암시적 변환은 그 자체로 문제가 없다. [null]을 명시적 강제변환을 해도 결과는 " " 이다. 배열을 그 내용과 동등하게 문자열화하는 게 맞는 것인지, 그리고 정확히 어떤 방법으로 문자열화해야 하는지, 이 두 문제가 서로 앞뒤가 맞지 않는다. 따라서 이 모든 말도 안 되는 결과를 가져온 String([..]) 규칙들이 비난의 대상이 되어야 할 것이다. 아예 배열에서 문자열 강제변환을 없애버리는 건 어떨까? 하지만 그렇게 되면 자바스크립트 언어의 다른 쪽에서 많은 흠집이 생기게 된다.

다음 사례 역시 유명하다.

```
0 == "\n"; // true
```

공백문자 " ", " \n " (또는 " " 및 다른 공란 문자열 조합[15])이 ToNumber를 경유하여 0으로 강제변환된다는 사실은 이미 앞에서 언급했다. " " 를 숫자로 강제변환한 결과로 0 이외에 다른 마땅한 대안이 있을까? 명시적으로는 Number(" ")이 0으로 변환되는 것이 그토록 불편한 진실인가?

" " 또는 " " 를 숫자로 강제변환한 결과값으로 차선책을 찾는다면 NaN을 떠올려볼 수 있을 것이다. 그런데 과연 NaN이 0보다 더 나은 대안일까? " " ==

15 역자주_ " \n " , " \n " 등을 말합니다.

NaN 평가 결과는 당연히 false지만, 모든 걱정거리가 다 해결될 수 있을지는 확실치 않다.

0 == "\n"이라고 해서 실존하는 자바스크립트 프로그램에 오류가 발생할 가능성은 대단히 희박하고, 이런 희귀 사례는 에둘러 가기 쉽다.

희귀한 타입 변환 사례는 어느 언어에나 항상 있기 마련이고 강제변환에 국한된 문제는 아니다. 특정한 희귀 사례들을 한데 모아 예측하는 것이 문제인데(마땅히 그래야 하겠지만!?), 전체적인 강제변환 메커니즘에 있어서 가장 주된 논거라고 할 순 없다.

요점만 정리하자면, (앞서 보았던 의도적으로 어렵게 만든 valueOf()나 toString() 핵 말고) 여러분이 맞닥뜨릴 가능성이 조금이라도 있는, 평범한 값 사이에서 이상하게 작동하는 강제변환은 방금 전 열거한 7가지가 전부라고 보면 된다.

다음 코드를 24인의 강제변환 수배범 명단과 대조해보자.

```
42 == "43"; // false
"foo" == 42; // false
"true" == true; // false

42 == "42"; // true
"foo" == [ "foo" ]; // true
```

이렇게 falsy도 아니고 희귀 사례도 아닌 강제변환(죄다 취합하자면 리스트가 무한정 길어지겠지만) 결과는 전적으로 안전하고 합리적이며 설명할 수 있다.

근본부터 따져보자

암시적 강제변환을 깊숙이 파보니 정말 말도 안 되는 것들이 몇 가지 발견되었다. 이러니 개발자들 사이에서 강제변환은 악의 축이니 뭐니 하면서, 피하는 게 상책이란 말이 나돌게 된 것이다.

자, 잠시 숨을 가다듬고 근본부터 다시 따져보자.

예상을 벗어난 정도를 기준으로 하면, 강제변환이 심각한 문제가 될 경우는 7가지 발견되었다. 하지만 나머지 항목(실제로는 무수히 많겠지만 적어도 17개는)들의 강제변환은 이치에 맞고 설명이 가능하다.

"목욕물과 함께 아기를 내던져버리는" 전형적인 예시로 이것보다 적합한 예가 또 있을까? 겨우 7개에 불과한 함정 때문에 강제변환이란 시스템을 전부 폐기해야 할까? (안전하고 유용하게 쓸 수 있는 강제변환의 목록이 무한정 많은데도?)

조금 신중한 사람이라면 이렇게 반문할 것이다. "그 많은 강제변환의 좋은 부분을 활용하는 동시에 몇몇 나쁜 부분을 어떻게 요리조리 잘 피해갈 수 있을까요?"

나쁜 부분 7인방을 모아 보자.

```
"0" == false; // true — 어이쿠!
false == 0; // true — 어이쿠!
false == ""; // true — 어이쿠!
false == []; // true — 어이쿠!
"" == 0; // true — 어이쿠!
"" == []; // true — 어이쿠!
0 == []; // true — 어이쿠!
```

이들 중 처음 4개는 == false 비교와 연관되며, 다시 말하지만 이런 의미 없는 비교는 절대 하지 말자. 이 한 가지는 기억하기 어렵지 않다.

그 다음 3개를 보자.

```
"" == 0; // true — 어이쿠!
"" == []; // true — 어이쿠!
0 == []; // true — 어이쿠!
```

이런 강제변환 코드를 실제로 쓸 일이 있을까? 그렇다고 치면 과연 어떤 상황에서

가능성이 있을까?

개인적인 생각으로는 자신이 도통 뭘 하는지 모르고 코딩하는 개발자가 아닌 다음에야 실제 자바스크립트 프로그램에서 == [] 식으로 불리언 평가를 할 경우는 극히 드물다. 대신, 다음과 같이 == "" 나 == 0를 사용하지 않을까?

```
function doSomething(a) {
  if (a == "") {
    // ..
  }
}
```

우연히 doSomething(0)나 doSomething([])로 호출하면 어이쿠, 할 것이다. 다른 예제를 보자.

```
function doSomething(a,b) {
  if (a == b) {
    // ..
  }
}
```

역시 마찬가지로 doSomething("", 0)나 doSomething([], "")로 호출하면 문제가 된다.

결국, 강제변환 때문에 골탕 먹을 경우의 수가 있다는 사실은 부인하기 어렵고 함정에 빠지지 않으려면 주의할 필요는 있겠지만, 코드베이스 전체를 통틀어 그럴만한 코드가 나올 가능성은 매우 희박하다는 걸 알 수 있다.

암시적 강제변환의 안전한 사용법

자, 아주 중요한 조언을 한 마디 하겠다. 여러분이 짠 프로그램을 잘 살펴보고 == 연산자 좌우의 피연산자 값이 실제로 어떤 모습일지 머릿속에 그려 보자. 불행한

사고를 미연에 방지하는 차원에서 동등 비교 시 원칙을 몇 가지 제시하겠다.

- 피연산자 중 하나가 true/false일 가능성이 있으면 '절대로' == 연산자를 쓰지 말자.
- 피연산자 중 하나가 [], " ", 0이 될 가능성이 있으면 가급적 == 연산자는 쓰지 말자.

이런 상황이라면 == 대신 ===를 사용하여 여러분이 의도하지 않은 강제변환을 차단하는 게 훨씬 좋다. 이 두 가지 원칙만 준수해도 웬만한 강제변환의 함정들은 효과적으로 피해갈 수 있으리라 생각한다.

좀 더 분명하게, 코드를 길게 작성하는 것도 골치 아픈 디버깅의 늪에서 빠져 나오는 데 도움이 된다.

결국 ==냐 ===냐 하는 문제는 한 마디로 "동등 비교 시 강제변환을 허용할 거냐 말 거냐"와 본질적으로 같다.

비교 로직을 (이를테면 null과 undefined와 함께 사용하여) 간결하게 표현할 수 있는 강제변환이 유용한 경우가 많다.

전반적으로 암시적 강제변환이 정말로 위험한 경우는 그다지 흔치 않다. 하지만 이런 경우라고 해도 ===로 대체하여 안전하게 가면 그만이다.

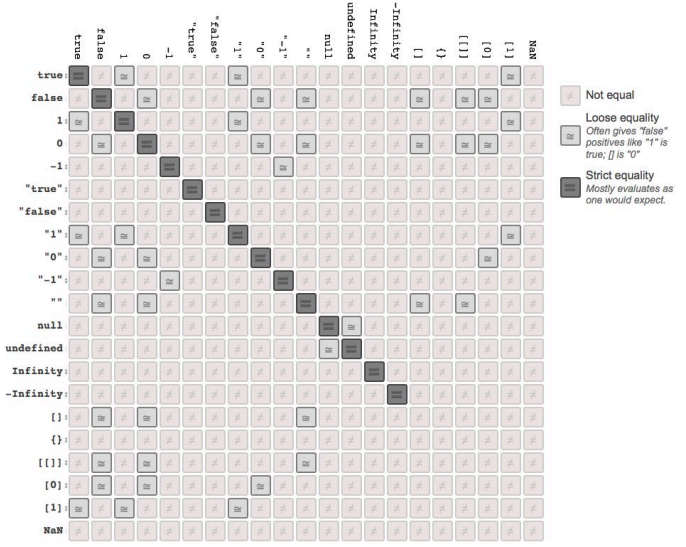
강제변환에 상처받지 않을 또 한 가지 비책은 바로 typeof 연산자다. typeof 연산은 항상 7가지 문자열 중 하나(공백 문자 " "는 없다)를 반환하므로(1장 타입 참고), 값의 타입을 체크한다고 암시적 강제변환과 문제가 될 일은 전혀 없다. typeof x == "function", typeof x === "function" 둘 다 100% 안전하다. 명세의 말마따나 이 둘의 알고리즘은 정확히 같다. 따라서 이 책을 읽는 여러분은 부디 코딩 툴이 경고를 날린다고 해서 또는 (이게 최악인데) 다른 자바스크립트 책에서 하지 말라고 씌어있다 하여 무턱대고 여기저기 ===를 남발하는 일은 없길 바란다. 코드의 품질은 결국 여러분이 어떻게 하느냐에 달려있다.

"암시적 강제변환, 그렇게 사악하고 위험한 녀석인가요?" - "네, 그럴 경우가 없잖지만, 99%는 아닙니다!"

사명감을 지닌 성숙한 개발자가 되자. 강제변환(명시적/암시적)의 강력한 파워를 효과적이고 안전하게 사용하는 방법을 잘 익혀서 주변 동료들 역시 잘 인도해주기 바란다.

[그림 4-1]은 동등 비교의 모든 조합을 나타낸 테이블로 깃허브 유저인 알렉스 도레이Alex Dorey (깃허브 @dorey)가 작성했다.

그림 4-1 자바스크립트의 동등 비교



4.6 추상 관계 비교

이 절에서 논의할 암시적 강제변환은 비교적 덜 관심 받는 분야지만, 그래도 a < b 비교 과정에서 어떤 일들이 벌어지는지 잘 알아둬야 한다(방금 전 a == b를 속속들이 알아본 것과 비슷하다).

ES5 §11.8.5 "추상적 관계 비교Abstract Relational Comparison" 알고리즘은 비교 시 피연산자 모두 문자열(후반부)일 때와 그 외의 경우(전반부), 두 가지로 나뉜다.

명세에는 a < b에 대해서만 정의되어 있다. a > b는 b < a로 처리된다.

이 알고리즘은 먼저 두 피연산자에 대해 ToPrimitive 강제변환을 실시하는 것으로 시작한다. 그 결과 어느 한쪽이라도 문자열이 아닐 경우, 양쪽 모두 ToNumber로 강제변환하여 숫자 값으로 만들어 비교한다.

예를 들면,

```
var a = [ 42 ];
var b = [ "43" ];

a < b; // true
b < a; // false
```

-0과 NaN 등을 조심해야 하는 건 앞서 설명한 == 알고리즘과 비슷하다.

그러나 < 비교 대상이 모두 문자열 값이면, 각 문자를 단순 어휘lexicographic(즉, 알파벳 순서로) 비교한다.

```
var a = [ "42" ];
var b = [ "043" ];

a < b; // false
```

두 배열을 ToPrimitive로 강제변환하면 문자열이기 때문에 a, b는 숫자로 강제변환하지 않는다. 따라서 "42"와 "043"를 문자 단위로(우선 "4"와 "0"을 비교하고, 그 다음 "2"와 "4"를 비교하는 식으로) 비교한다. "0"은 어휘상 "4"

보다 작은 값이므로 비교는 처음부터 실패한다.

다음 코드 역시 수행 로직은 동일하다.

```
var a = [ 4, 2 ];
var b = [ 0, 4, 3 ];

a < b; // false
```

a는 " 4,2 " 로, b는 " 0,4,3 " 으로 문자열화시킨 후 앞 예제와 마찬가지로 어휘 비교를 한다.

다른 예제를 보자.

```
var a = { b: 42 };
var b = { b: 43 };

a < b; // 결과는?
```

결과는 false다. a도 [object Object], b도 [object Object]로 변환되어 어휘적인 비교를 할 수 없기 때문이다.

하지만 이상한 건,

```
var a = { b: 42 };
var b = { b: 43 };

a < b; // false
a == b; // false
a > b; // false

a <= b; // true
a >= b; // true
```

`a == b`는 왜 `true`가 아닐까? 둘 다 동일한 문자열(`"[object Object]"`)이면 동등한 것 아닐까? 아니다. 이미 앞에서 `==`이 객체 레퍼런스에서 어떻게 작동하는지 설명했다.[16]

그건 그렇고, `a < b`, `a == b`, `a > b` 모두 `false`인데 `a <= b`와 `a >= b`는 어떻게 `true`인 걸까?

`a <= b`는 실제로 `b < a`의 평가 결과를 부정하도록 명세에 기술되어 있기 때문이다. 그래서 `b < a`가 `false`이므로 `a <= b`는 이를 부정한 `true`가 된다.

여러분은 지금까지 `<=`이 "같거나 더 작은"이라는 부등호의 의미일 거라 당연시해왔겠지만, 참 짓궂게도 정반대일 가능성도 있다. 실제로 자바스크립트 엔진은 `<=`를 "더 크지 않은"(`!(a > b)` → `!(b < a)`로 처리)의 의미로 해석한다. 더구나 `a >= b`는 먼저 `b <= a`로 재해석한 다음 동일한 추론을 적용한다.

불행히도 동등 비교에 관한 한 "엄격한 관계 비교strict relational comparison"는 없다. 다시 말해, 비교 전 a와 b 모두 명시적으로 동일한 타입임을 확실히 하는 방법 말고 `a < b` 같은 관계 비교 과정에서 암시적 강제변환을 원천 봉쇄할 수는 없다.

앞 절에서 `==` 대 `===` 문제를 근본적으로 따져본 것과 같은 원리다. `42 < "43"`처럼 강제변환이 유용하고 어느 정도 안전할 관계 비교라면 그냥 쓰자. 반면, 조심해서 관계 비교를 해야 할 것 같은 상황에서는 `<`(또는 `>`)를 사용하기 전, 비교할 값들을 명시적으로 강제변환해두는 편이 안전하다.

```
var a = [ 42 ];
var b = "043";

a < b; // false -- 문자열 비교!
```

16 역자주_ 두 객체가 정확히 똑같은 값에 대한 레퍼런스일 경우에만 동등합니다. a, b는 아예 값 자체도 다른, 별개의 객체입니다.

```
Number( a ) < Number( b ); // true -- 숫자 비교!
```

4.7 정리하기

이 장에서는 강제변환이라고 하는 자바스크립트의 타입 변환의 작동 원리를 명시적/암시적 두 가지 유형으로 나누어 알아보았다.

강제변환은 많은 욕을 얻어먹는 애물단지지만, 알고 보면 많은 경우 꽤 유용한 기능이다. 자바스크립트 개발을 천직으로 생각하는 사람이라면 시간을 내서 강제변환을 꼼꼼히 학습하고, 강제변환의 어떤 특성을 적극 활용하고 어떤 부분은 조심해서 우회해야 할지 잘 판단해서 결정해야 한다.

명시적 강제변환은 다른 타입의 값으로 변환하는 의도가 확실한 코드를 말하며 혼동의 여지를 줄이고 코드 가독성 및 유지 보수성을 높일 수 있는 장점이 있다.

암시적인 강제변환은 "숨겨진" 로직에 의한 부수 효과가 있으며 타입 변환이 처리되는 과정이 명확하지 않다. 그래서 암시적 강제변환이 명시적 강제변환의 정반대고 나쁜 것이라고들 하지만(주변에 이렇게 얘기하는 사람들이 의외로 많다!), 실은 암시적 강제변환이 오히려 코드 가독성을 향상시키는 장점도 있다.

암시적 강제변환은 변환 과정이 구체적으로 어떻게 일어나는지 명확하게 알고 사용해야 한다. 여러분 스스로 지금 내가 무슨 코드를 짜고 있고 어떻게 작동할 거란 점은 알고 있어야 한다. 더 나아가 다른 개발자들도 쉽게 배우고 이해할 수 있는 코드를 작성하도록 노력하기 바란다.

chapter 5

문법

마지막으로, 자바스크립트 언어에서 구문이 어떻게 작동하는지(문법) 알아보자. 자바스크립트 코딩 실력을 자신하는 사람들에게도, 자바스크립트 언어의 문법은 온갖 혼란과 오해의 늪에 빠지게 할, 별별 미묘한 것들로 가득하다. 자, 이런 것들을 확실히 끄집어내어 그간 여러분이 잘못 알아온 것들을 말끔히 정리하자.

독자들에게는 "문법grammar"보다 "구문syntax"이란 용어가 더 익숙할 것이다. 두 용어 모두 언어가 작동하는 규칙을 기술한다는 점에서 상당히 유사하다. 물론 미묘한 차이도 있지만, 이 장의 내용에 영향을 끼칠 만한 정도는 아니다. 자바스크립트 문법은 각종 구문(연산자, 키워드 등)들을 서로 잘 끼워 맞춰 규칙에 맞는, 유효한 프로그램을 만들기 위한, 구조적 방법이다. 즉, 문법을 배제한 채 구문을 논한다는 건 중요한 세부사항 역시 버리겠다는 말과 같다. 따라서 개발자들이 직접 마주할 대상은 다듬어지지 않은, 있는 그대로의 언어 구문이지만, 이 장에서는 최대한 정확한 문법을 기술하는 데 초점을 둘 것이다.

5.1 문과 표현식

문statement과 표현식expression을 대충 같은 의미라고 넘겨버리는 개발자들이 허다하다. 그러나 자바스크립트에서 두 용어는 아주 중요한 차이가 있으므로 명확하게 분별하자.

확실히 구분 짓기 위해 여러분이 좀 더 익숙한 영어 언어의 용어를 빌려보자.

"문장sentence"은 생각을 표현하는 단어들의 완전한 조형물이다. 문장은 하나

이상의 "어구phrase"로 구성되며, 각 어구는 구두점punctuation mark이나 접속사conjunction("그리고", "또는" 등)로 연결할 수 있고, 어구는 더 작은 어구로 나눌 수 있다. 어떤 어구는 불완전하여 그 자체로 완성된 문장을 형성할 수 없지만, 스스로의 힘만으로 완성되는 어구도 있다. 이러한 규칙들을 통틀어 영어 언어의 문법이라고 한다.

자바스크립트 문법도 마찬가지다. 문statement은 문장sentence, 표현식expression은 어구phrase, 연산자는 구두점/접속사에 해당된다. 자바스크립트에서 모든 표현식은 단일한, 특정한 결과값으로 계산된다. 예를 들면,

```
var a = 3 * 6;
var b = a;
b;
```

여기서 `3 * 6`은 (18이라는 값으로 평가되는) 표현식이다. 두 번째 줄 역시 표현식이며, 세 번째 줄 `b`도 표현식이다. `a`, `b` 표현식 모두 당시 변수들에 저장된 값으로 평가되므로 `b` 역시 18이 된다.

게다가 이 세 줄은 각각 표현식이 포함된 문이다. `var a = 3 * 6`, `var b = a` 두 문은 각각 변수를 선언(그리고 선택적으로 동시에 어떤 값을 할당)하므로 "선언문declaration statement"이라 한다. (앞에 `var`가 빠진) `a = 3 * 6`나 `b = a`는 "할당 표현식assignment expression"이라고 한다.

세 번째 줄은 `b`가 표현식의 전부지만, 이것만으로도 완전한 문이다(그다지 흥미를 유발하는 문은 아니지만!). 일반적으로 이런 문을 "표현식 문expression statement"이라고 일컫는다.

5.1.1 문의 완료 값

모든 문은 (그 값이 `undefined`라 해도) 완료 값completion value을 가진다는 사실을 의외

로 모르는 사람들이 많다.[01]

문의 완료 값은 들여다볼 방법은 없을까?

가장 확실한 방법은 브라우저 개발자 콘솔 창에서 문을 타이핑해보는 것이다. 콘솔 창은 가장 최근에 실행된 문의 완료 값을 기본적으로 출력하게 되어 있다.

`var b = a` 같은 문은 완료 값이 뭘까?

할당 표현식 `b = a`는 할당 이후의 값(여기서는 18)이 완료 값이지만, `var` 문 자체의 완료 값은 `undefined`다. 명세에 그렇게 적혀있기 때문이다.[02] 실제로 `var a = 42`를 콘솔 창에서 실행해보면 42 대신 `undefined`가 나온다.

엄밀히 말하여 실제로는 이것보다 좀 더 복잡하다. ES5 §12.2 "변수 문Variable Statement"의 `VariableDeclaration` 알고리즘은 실제로 어떤 값(선언된 변수명이 포함된 문자열 – 좀 이상하지 않은가?)을 반환하지만, 이 값은 (`for..in` 루프에서 사용할 경우를 제외하고) `VariableStatement` 알고리즘이 꿀꺽 삼켜버린 후 텅 빈(undefined) 완료 값을 내어놓는다.

콘솔 창(또는 자바스크립트 `REPLRead/Evaluate/Print/Loop` 툴)에서 여러 가지 코드를 테스트해본 사람이라면 문의 실행 결과가 `undefined`로 표시되는 걸 자주 목격했을 텐데, 대체 이게 뭘까 진지하게 고민해본 적은 한 번도 없었을 것이다. 한 마디로 콘솔은 실행한 문의 완료 값을 보고한 것이다.

그러나 콘솔 창이 내어준 완료 값은 개발자가 내부 프로그램에서 사용할 수 있는 값은 아니다. 완료 값을 순간 포착할 방법은 없을까?

방법은 있지만 꽤 복잡한 작업이다. 그런데 방법을 설명하기 앞서, 왜 꼭 그래야 하는지부터 생각하자.

---

01 역자주_ ES5.1, 12장. 문(statement) 도입부에 '문의 평가 결과는 항상 완료 값이다(The result of evaluating a Statement is always a Completion value).' 라고 기술되어 있습니다.

02 역자주_ http://www.ecma-international.org/ecma-262/5.1/#sec-12.2

다른 종류의 문 완료 값을 보자. 예를 들어, 보통의 { .. } 블록은 내부의 가장 마지막 문/표현식의 완료 값을 자신의 완료 값으로 반환한다.

```
var b;

if (true) {
b = 4 + 38;
}
```

콘솔 창에서 실행하면 42가 나온다. 블록 내의 마지막 문 b = 4 + 38의 완료 값이 42이므로 if 블록의 완료 값도 42를 반환한 것이다.

즉, 블록의 완료 값은 내부에 있는 마지막 문의 값을 암시적으로 반환한 값이다.

커피스크립트CoffeeScript 같은 언어 역시 함수 내 마지막 문의 값이 함수 자체의 암시적인 반환 값이라는 점에서는 개념적으로 유사하다.

하지만 다음과 같은 코드가 작동하지 않는 건 분명히 문제가 있다.

```
var a, b;

a = if (true) {
b = 4 + 38;
};
```

문의 완료 값을 포착하여 다른 변수에 할당한다는 건 쉬운 구문/문법으로는 불가능하다 .

뭔가 방법이 없을까?

어디까지나 예를 들기 위함이니 다음 코드는 실제로 사용하지 말자!

완료 값을 포착하려면 어쩔 수 없이 유해함의 대명사 `eval( .. )` 함수(종종 "evil"로도 발음한다)를 사용할 수밖에 없다.

```
var a, b;

a = eval( "if (true) { b = 4 + 38; }" );

a; // 42
```

으휴, 꼴 보기 싫은 코드지만 일단 잘 돌아간다! 콘솔 창 외에 자바스크립트 프로그램에서도 문 완료 값을 확인할 길이 있음을 알 수 있다.

ES7 명세에는 "do 표현식"이 제안된 상태다. 다음 코드를 보자.

```
var a, b;

a = do {
if (true) {
b = 4 + 38;
}
};

a; // 42
```

`do { .. }` 표현식은 (하나 이상의 문을 포함한) 블록 실행 후, 블록 내 마지막 문의 완료 값을 do 표현식 전체의 완료 값으로 반환하며, 결국 이 값이 변수 a에 할당된다.

인라인 함수 표현식 안에 감싸서 명시적으로 반환할 필요 없이 문을 (다른 문 안에 들어갈 수 있는) 표현식처럼 다루자는 게 기본적인 아이디어다.

아직까지는 문 완료 값을 대수롭지 않게 여기고 있지만, 자바스크립트 언어가 진화할수록 그 중요성은 점점 더 부각될 것 같다. 어서 `do { .. }` 표현식이 도입되어

eval(..) 같은 나쁜 것들을 사용하고픈 욕구를 영원히 잠재울 수 있길 고대한다.

노파심에 다시 경고하지만 eval(..)은 사용하지 말자. 진심이다. 자세한 내용은 『You Don't Know JS: 스코프와 클로저』[03]를 참고하자.

5.1.2 표현식의 부수 효과

대부분의 표현식에는 부수 효과가 없다. 예컨대,

```
var a = 2;
var b = a + 3;
```

표현식 a + 3 자체는 가령 a 값을 바꾸는 등의 부수 효과가 전혀 없다. 단지 b = a + 3 문에서 결과값 5가 b에 할당될 뿐이다.

다음의 함수 호출 표현식은 부수 효과를 가진(가졌을지 모를) 표현식의 전형적인 예다.

```
function foo() {
a = a + 1;
}

var a = 1;
foo(); // 결과값: 'undefined', 부수 효과: 'a'가 변경됨.
```

다른 부수 효과를 지닌 표현식을 보자.

```
var a = 42;
var b = a++;
```

03 http://goo.gl/YCqO7K

표현식 a++이 하는 일은 두 가지다. a의 현재 값 42를 반환(그리고 b에 할당하는 것까지)하고 a 값을 1만큼 증가시킨다.

```
var a = 42;
var b = a++;

a; // 43
b; // 42
```

여기서 b 값을 43으로 착각하는 개발자들이 생각보다 많다. 무엇보다 ++ 연산자에 부수 효과가 있다는 걸 잘 모르니까 더 헷갈리는 것이다.

단항 연산자인 증가 연산자(++)/감소 연산자(--)는 전위prefix("앞") 또는 후위postfix("뒤") 연산자로 사용된다.

```
var a = 42;

a++; // 42
a; // 43

++a; // 44
a; // 44
```

++를 전위 연산자로 사용하면 표현식으로부터 값이 반환되기 전에 (a 값을 1만큼 증가시키는) 부수 효과를 일으킨다. 반면, 후위 연산자로 사용하면 값을 반환한 이후에 부수 효과가 발생한다.

++a++은 문법에 맞는 구문일까? 실행하면 ReferenceError 에러가 난다. 부수 효과를 유발하는 연산자는 부수 효과를 일으킬 변수 레퍼런스가 꼭 필요하기 때문이다. ++a++에서는 a++ 부분이 (연산자 우선순위 규칙에 의해) 먼저 평가되어 증가되기 이전의 값을 돌려준다. 따라서 ++42 평가 시 ++ 연산자는 42 같은 원시 값에 직접 부수 효과를 일으킬 수는 없으므로 ReferenceError 에러를 던진다.

a++를 ( )로 감싸면 후위 부수 효과를 캡슐화할 수 있다고 착각하는 경우도 더러 있다.

```
var a = 42;
var b = (a++);

a; // 43
b; // 42
```

안타깝지만 ( )로 둘러싼다 해도 a++ 표현식에서 부수 효과 발생 이후 재평가된 새 표현식을 만들어내는 건 불가능하다. 설사 가능하다 하더라도 a++는 42를 먼저 반환하므로 ++ 부수 효과 이후 재평가가 가능한 연산자가 따로 있지 않은 한, 이 표현식에서 43을 반환받을 도리는 없다. 그래서 b에는 43이 할당되지 않는다.

하지만 방법이 아주 없는 건 아니다. 문을 나열하는statement-series 콤마 연산자 ,를 사용하면 다수의 개별 표현식을 하나의 문으로 연결할 수 있다.

```
var a = 42, b;
b = ( a++, a );

a; // 43
b; // 43
```

뒷 부분에서 다시 얘기하겠지만, 연산자 우선순위 규칙에 의해 a++, a 양쪽 괄호는 빠뜨리면 안 된다.

a++, a 표현식은 두 번째 a 표현식을 첫 번째 a++ 표현식에서 부수 효과가 발생한 이후에 평가한다. 그래서 b 값은 43이다.

delete 역시 부수 효과를 일으키는 연산자다. 2장 값에서 설명한 대로 delete는 객체의 프로퍼티를 없애거나 배열에서 슬롯을 제거할 때 쓴다. 하지만 단독 문

standalone statement으로 더 많이 쓴다.

```
var obj = {
a: 42
};

obj.a; // 42
delete obj.a; // true
obj.a; // undefined
```

delete 연산자의 결과값은 유효한/허용된 연산일 경우 true, 그 외에는 false다. 이 연산자의 부수 효과는 바로 프로퍼티(또는 배열 슬롯)를 제거하는 것이다.

'유효한/허용된'이란 무슨 의미일까? 존재하지 않는 프로퍼티, 또는 존재하면서 설정 가능한configurable한 프로퍼티일 경우(『You Don't Know JS: this & Object Prototypes』[04] 참고), delete 연산자는 true를 반환한다. 그 외의 경우는 false를 반환하거나 에러를 낸다.

마지막으로 예시할 부수 효과 유발 연산자는, 언뜻 보기에 분명한 것 같으면서도 분명하지 않은, = 할당 연산자다.

```
var a;

a = 42; // 42
a; // 42
```

a = 42에서 = 연산자는 아무리 봐도 부수 효과와는 무관해 보인다. 하지만 a = 42 문의 실행 결과는 이제 막 할당된 값(42)이므로 42를 a에 할당하는 자체가 본질적으로 부수 효과다.

04 http://goo.gl/oZEHe0

+=, -= 등의 복합 할당 연산자 역시 부수 효과를 지닌다. 예를 들어, a = b += 2 문에서 b += 2이 먼저 처리되고(b = b + 2), = 연산자가 그 결과값을 a에 할당한다.

이렇게 할당 표현식/문 실행 시 할당된 값이 완료 값이 되는 작동 원리는 다음과 같은 연쇄 할당문chained assignment에서 특히 유용하다.

```
var a, b, c;

a = b = c = 42;
```

c = 42 평가 결과는 (42를 c에 할당하는 부수 효과를 일으키며) 42가 되고, b = 42 평가 결과는 (42를 b에 할당하는 부수 효과를 일으키며) 42가 된다. 결국, a = 42로 (42를 a에 할당하는 부수 효과를 일으키며) 평가된다.

var a = b = 42처럼 연쇄 할당문을 잘못 쓰는 경우가 많다. 겉보기엔 똑같을 것 같지만 그렇지 않다. 변수 b를 (해당 스코프 어딘가에) 선언하지 않은 상태에서 실행하면 이 할당문은 b를 직접 선언하지 않는다. 대신, 엄격 모드 여부에 따라 에러가 나거나 원치 않는 전역 변수가 생성된다(『You Don't Know JS: this & Object Prototypes』[05] 참고).

또 다른 예를 보자.

```
function vowels(str) {
var matches;

if (str) {
// 모든 모음을 추출한다.
matches = str.match( /[aeiou]/g );

if (matches) {
return matches;
```

05 http://goo.gl/oZEHe0

```
}
}
}

vowels( "Hello World" ); // ["e","o","o"]
```

잘 작동하는 코드다. 많은 개발자가 이렇게 작성한다. 할당 연산자의 부수 효과를 잘 활용하면 다음과 같이 2개의 if 문을 하나로 간단히 합칠 수 있다.

```
function vowels(str) {
var matches;

// 모든 모음을 추출한다.
if (str && (matches = str.match( /[aeiou]/g ))) {
return matches;
}
}

vowels( "Hello World" ); // ["e","o","o"]
```

matches = str.match..를 감싸는 (..)를 빠뜨리면 안 된다.

두 조건이 서로 분명히 연관되어 있음을 잘 보여주기 때문에 필자는 후자를 더 선호하는 편이다. 물론 어떤 스타일을 선호할지는 순전히 개인 취향에 달려 있다.

5.1.3 콘텍스트 규칙

자바스크립트 문법 규칙 중에는 같은 구문이지만 어디에서 어떤 식으로 사용하느냐에 따라 서로 다른 의미를 가지는 경우가 있다. 하나씩 떨어뜨려 놓고 보면 상당히 헷갈릴 수 있다.

모든 사례를 일일이 나열할 순 없고, 그중 자주 나오는 몇 가지 사례를 살펴보자.

중괄호

자바스크립트에서 중괄호 { .. }이 나올 법한 곳은 (자바스크립트가 진화되면서 점점 더 늘어나겠지만!) 크게 두 군데다.

객체 리터럴

첫째, 객체 리터럴이다.

```
// 'bar()'함수는 앞에서 정의되었다.

var a = {
foo: bar()
};
```

{ .. }는 a에 할당될 값이므로 객체 리터럴이 맞다.

레퍼런스 a는 할당의 대상target이므로 "l 값l-value" (좌측lefthand 값), { .. }는 할당의 원본source 값이므로 "r 값r-value" (우측righthand 값)이라고 한다.

레이블

방금 전 코드에서 var a = 부분을 삭제하면 어떻게 될까?

```
// 'bar()' 함수는 앞에서 정의되었다.

{
foo: bar()
}
```

{ .. }는 어디에도 할당되지 않은, 그저 고립된 객체 리터럴처럼 보인다. 하지만 전혀 그렇지 않다.

여기서의 { .. }는 평범한 코드 블록이다. { .. } 혼자 떨어져 있는 모양새가 (다른 언어와 달리) 자바스크립트에서는 그리 자연스럽지 않지만, 문법적으로는 100% 옳은 코드다. 특히 let 블록 스코프 선언과 함께 쓰이면 아주 유용하다 (『You Don't Know JS: 스코프와 클로저』[06] 참고).

이 { .. } 코드 블록은 for/while 루프, if 조건 등에 붙어있는 코드 블록과 기능적으로 매우 유사하다.

그런데 그저 평범한 코드 블록에 불과하다면 foo: bar() 구문이 특이하게 보이는 이유는 무엇이고, 어떻게 이런 코드가 문법에 부합한단 말일까?

그건 자바스크립트에서 "레이블 문labeled statement"이라 부르는, 거의 잘 알려지지 않은 (그리고 솔직히 권장하고 싶지 않은), 기능 덕분이다. 즉, foo는 bar() 문의 레이블이다. 그런데 갑자기 레이블 문 얘기는 왜 꺼낸 걸까?

자바스크립트에 만약 goto 문이 있었다면 goto foo하여 코드 실행을 특정 위치로 점프하는 것이 이론적으로 가능할 것이다. 알다시피 goto는 코드를 아주 난해하게 만드는 (일명 "스파게티 코드spaghetti code"의) 주범이며, 자바스크립트가 goto 문을 지원하지 않기로 한 건 정말 잘 한 일이다.

하지만 제한적이기는 해도 자바스크립트에는 레이블 점프labeled jump라는 특별한 형태의 goto 장치가 대신 마련되어 있다. continue와 break 문은 선택적으로 어떤 레이블을 받아 goto처럼 프로그램의 실행 흐름을 "점프"시킨다.

```
// 'foo' 레이블 루프
foo: for (var i=0; i<4; i++) {
for (var j=0; j<4; j++) {
// 두 루프의 반복자가 같을 때마다 바깥쪽 루프를 continue 한다.
if (j == i) {
```

06 http://goo.gl/YCqO7K

```
// 다음 순회 시 'foo' 붙은 루프로 점프한다.
continue foo;
}

// 홀수 배수는 건너뛴다.
if ((j * i) % 2 == 1) {
// 평범한(레이블 없는), 안쪽 루프의 'continue'
continue;
}

console.log( i, j );
}
}

// 1 0
// 2 0
// 2 1
// 3 0
// 3 2
```

`continue foo`는 "foo라는 레이블 위치로 이동하여 계속 순회하라"는 의미가 아니라 "foo라는 레이블이 붙은 루프의 다음 순회를 계속하라"는 뜻이다. 따라서 사실 임의적인 `goto`와는 다르다.

코드를 보면 알 수 있듯이, 홀수 배수 3 1은 지나갔지만 레이블 루프 점프 역시 1 1 및 2 2 순회를 건너뛰었다.

아마도 바깥쪽 루프로 나가야 할 지점에 안쪽 루프에서 `break` ——처럼 사용해야 레이블 루프 점프를 좀 더 잘 활용하는 모습이 될 것 같다. 레이블 `break`를 쓰지 않고 작성하면 같은 로직이라도 다음 코드처럼 영 어색해진다.

```
// 'foo' 레이블 루프
foo: for (var i=0; i<4; i++) {
for (var j=0; j<4; j++) {
if ((i * j) >= 3) {
console.log( "그만!", i, j );
```

```
break foo;
}

console.log( i, j );
}
}

// 0 0
// 0 1
// 0 2
// 0 3
// 1 0
// 1 1
// 1 2
// 그만! 1 3
```

break foo는 "foo라는 레이블 위치로 이동하여 계속 순회하라"는 의미가 아니라, "foo라는 레이블이 붙은 바깥쪽 루프/블록 밖으로 나가 그 이후부터 계속하라"는 뜻이다. 역시 전통적인 goto와는 의미가 사뭇 다르다.

방금 전 예제를 레이블 없는nonlabeled break로 작성하려면 아무래도 함수 한두 개가 더 필요하고, 공유된 스코프 변수 접근 등을 신경 써야 하므로 코드가 더 복잡하고 혼란스러워 질 수 있다. 따라서 이런 경우라면 레이블 break 사용이 더 나은 선택이라 할 수 있다.

레이블은 비 루프nonloop 블록에 적용할 수 있는데, 단 이런 비 루프 레이블은 break만 참조할 수 있다. 레이블 break ——를 써서 레이블 블록 밖으로 나갈 수는 있지만, 비 루프 블록을 continue —— 한다든가, 레이블 없는 break로 블록을 빠져나가는 건 안 된다.

```
// 'bar' 레이블 블록
function foo() {
bar: {
console.log( "Hello" );
```

```
break bar;
  console.log( "절대 실행 안 되지!" );
}
console.log( "world" );
}

foo();
// Hello
// world
```

레이블 루프/블록은 사용 빈도가 극히 드물고 못마땅한 구석도 많아 가능한 한 피하는 게 상책이다. 이를테면, 루프 점프를 할 바에야 차라리 함수 호출이 더 낫다. 하지만 제한적이나마 도움이 되는 경우가 없지 않으므로, 만약 레이블 점프 기능을 사용할 의도라면 여러분이 뜻한 바를 상세한 주석으로 잘 문서화하기 바란다!

JSON이 자바스크립트 고유의 하위 집합이라고 생각하는 사람들을 아주 많이 봤다. 그래서인지 JSON 문자열(가령 { " a " :42} - JSON 규칙에 따라 프로퍼티 명을 따옴표로 감싼다)이 유효한 자바스크립트 프로그램이라고 착각을 한다. 어림없는 소리! 지금 { " a " :42}를 콘솔 창에 입력해보라, 에러가 날 테니.

이유는 간단하다. 자바스크립트 문의 레이블은 따옴표로 감싸면 안 되기 때문에 " a " 는 문법에 맞는 레이블이 아니며, 그래서 :이 그 뒤에 오면 안 된다. 따라서 JSON은 자바스크립트 구문의 하위 집합이라 할 수 있지만, 그 자체로 올바른 자바스크립트 문법은 아니다.

JSON에 관한 너무나 흔한 오해 중 하나는, 내용이 JSON 문자열로만 가득 채워진 파일을 <script src=..> 태그로 읽어 들여도 정상적인 자바스크립트 코드로 인식될 거란 섣부른 예상이다. 직접 해보면 알겠지만, 이런 파일은 프로그램에서 접근조차 할 수 없다. 그래서 보통 JSON-P(JSON 데이터를 foo({ " a " :42})와 같은 함수 호출로 감싸는 패턴) 방식으로 자바스크립트 함수 중 하나에 이 값을 인자로 실어 보내면 접근이 안 되는 문제를 해결할 수 있다고 한다.

하지만 사실은 그렇지 않다. { " a " :42}은 완전히 올바른 JSON 값이지만, 그 자체로는 레이블이 잘못된 문 블록으로 해석되어 에러가 난다. 하지만 foo({ " a " :42})는 함수 내부에서 { " a " :42}이 foo(. .)에 전달된 객체 리터럴이므로 유효한 자바스크립트 코드다. 따라서 제대로 표현하자면, JSON-P가 JSON을 문법에 맞는 자바스크립트 코드로 옷을 갈아입혀 주는 셈이다.

블록

곧잘 회자되는 (강제변환 관련 - 4장 강제변환 참고) 자바스크립트 함정 중엔 이런 것도 있다.

```
[] + {}; // "[object Object]"
{} + []; // 0
```

마치 + 연산자가 첫 번째 피연산자에 따라([]/{}) 다른 결과를 내놓는 것처럼 보인다. 하지만 실제로는 전혀 상관없다!

윗 줄에서 엔진은 + 연산자 표현식의 {}를 실제 값(빈 객체)으로 해석한다. 4장 강제변환에서 설명했듯이 []는 " "로 강제변환되고, {}도 문자열 " [object Object] "로 강제변환된다.

그러나 아랫 줄의 {}는 동떨어진 (아무 일도 하지 않는) 빈 블록으로 간주한다. 블록 끝을 꼭 세미콜론으로 끝내야 한다는 법은 없으므로 문제될 건 없다. 결국 + [] 표현식에서 명시적으로 []를 숫자 0으로 강제변환한다.

객체 분해

ES6부터는 "분해 할당destructuring assignments", 구체적으로는 객체 분해 시 { . . }를 사용한다(『You Don't Know JS: ES6 & Beyond』[07] 참고).

07 http://goo.gl/GqxITA

```
function getData() {
// ..
return {
a: 42,
b: "foo"
};
}

var { a, b } = getData();
console.log( a, b ); // 42 "foo"
```

var { a , b } = ..이 ES6 분해 할당의 형식이며, 그 의미는 대략 다음 코드와 같다.

```
var res = getData();
var a = res.a;
var b = res.b;
```

{a, b}는 { a: a, b: b }를 간결하게 쓴 형태로 작동에 문제도 없을 뿐더러 코드가 짧아져 더 좋다.

{ .. }를 이용한 객체 분해는 명명된 함수에도 활용할 수 있는데, 이를테면 암시적인 객체 프로퍼티 할당과 비슷한 간편 구문sugar syntax[08]이다.

```
function foo({ a, b, c }) {
// 다음 코드처럼 할 필요가 없다.
// var a = obj.a, b = obj.b, c = obj.c
console.log( a, b, c );
}
```

08 역자주_ 더 읽기 편하고 표현하기 쉬운 프로그래밍 언어의 구문을 말합니다. 마치 사람에게 설탕(sugar) 같은 단맛을 느끼게 해주는 독특한 언어의 스타일이라 생각하면 됩니다.

```
foo( {
c: [1,2,3],
a: 42,
b: "foo"
} ); // 42 "foo" [1, 2, 3]
```

{ .. }은 전적으로 사용 콘텍스트에 따라 의미가 결정되는데, 여기서 문법과 구문의 차이점을 엿볼 수 있다. 자바스크립트 엔진이 여러분이 예상치 못했던 방향으로 해석하는 것을 방지하는 차원에서라도 이러한 미묘한 차이를 이해하는 것이 중요하다.

else if와 선택적 블록

다음 코드가 잘 작동한다고 해서 자바스크립트에 `else if` 절이 존재한다고 믿는 건 미신을 신봉하는 것과 같다.

```
if (a) {
// ..
}
else if (b) {
// ..
}
else {
// ..
}
```

실은 `else if` 같은 건 없다. 자바스크립트 문법의 숨겨진 특성이다. `if`와 `else` 문은 실행문이 하나밖에 없는 경우, 블록을 감싸는 { }를 생략해도 된다. 여러분은 다음과 같은 코드를 무던히도 많이 봐왔을 것이다.

```
if (a) doSomething( a );
```

여러 스타일 가이드 문서에는 다음처럼 단일 문 블록도 { }로 감싸라고 조언한다.

```
if (a) { doSomething( a ); }
```

그러나 정확히 동일한 문법 규칙이 else 절에도 적용되어 좀 전에 봤던 코드는 실제로는 항상 이렇게 파싱된다.

```
if (a) {
// ..
}
else {
if (b) {
  // ..
}
else {
// ..
}
}
```

else 이후의 if (b) { .. } else { .. }는 단일 문이므로 { }로 감싸든 말든 상관없다. 즉, else if라고 쓰는 건 표준 스타일 가이드의 위반 사례가 되며, 단일 if 문과 같이 else를 정의한 셈이 된다.

물론 else if 는 이미 누구나 다 쓰는 관용 코드고, 한 단계 하위로 들여쓰기를 하는 효과가 있어 나름 매력은 있다. 어떤 식으로 코딩하든 여러분 자신의 스타일 가이드와 규칙을 명시적으로 준수하고 else if를 정확한 문법 규칙인 양 넘겨짚지는 말자.

5.2 연산자 우선 순위

4장에서 설명했지만 자바스크립트에서 &&, || 연산자는 단순히 true/false를

반환하는 게 아니라, 독특하게도 피연산자 중 하나를 선택하여 반환한다. 그나마 피연산자가 2개뿐이고 연산자가 하나밖에 없을 경우는 결과값을 쉽게 예상할 수 있다.

```
var a = 42;
var b = "foo";

a && b; // "foo"
a || b; // 42
```

하지만 연산자가 2개, 피연산자가 3개면?

```
var a = 42;
var b = "foo";
var c = [1,2,3];

a && b || c; // 결과는?
a || b && c; // 결과는?
```

두 표현식의 결과를 이해하려면 표현식에 연산자가 여러 개 있을 경우 어떤 규칙으로 처리되는지 알아야 한다.

바로 이 규칙을 "연산자 우선순위operator precedence"라고 한다.

여러분 상당수는 연산자 우선순위 정도는 자신이 잘 알고 있다고 자부할 것이다. 그러나 본 시리즈의 성격상 여러분의 지식이 얼마나 확실한지 요리조리 쿡쿡 찔러 볼 생각이고, 운 좋으면 그 과정에서 여러분은 몰랐던 새로운 진실에 눈을 뜨게 될 것이다.

앞에서 한번 봤던 예제다.

```
var a = 42, b;
b = ( a++, a );
```

```
a; // 43
b; // 43
```

( )를 없애면 결과는 달라질까?

```
var a = 42, b;
b = a++, a;

a; // 43
b; // 42
```

어랏, 갑자기 b 값은 왜 바뀌었을까?

이유는 , 연산자가 = 연산자보다 우선순위가 낮기 때문이다. 그러므로 b = a++, a를 엔진은 (b = a++), a로 해석한다. (이미 설명했듯이) a++는 ++ 연산자가 a 값을 변경하는 부수 효과를 일으키기 전 b에 42 값을 할당한다.

연산자 우선순위를 왜 알아야 하는지 좋은 참고가 되었으리라 본다. 다수의 문을 연결하는 연산자로 ,를 사용할 때에는 이 연산자의 우선순위가 최하위라는 사실 또한 반드시 알고 있어야 한다. 즉, 어떤 연산자라도 ,보다 먼저 묶인다.

다른 예제를 보자.

```
if (str && (matches = str.match( /[aeiou]/g ))) {
// ..
}
```

할당문 양쪽을 ( )로 감싸야 한다. &&가 =보다 우선순위가 높으므로 ( )로 묶어주지 않으면 표현식은 (str && matches) = str.match...로 처리된다. (str && matches)는 정상적인 변수가 아닌, 어떤 값(여기서는 undefined)으로 평가되는데 = 할당 연산자 좌측에 값이 나오는 건 어불성설이므로 에러가 난다.

좋다, 연산자 우선순위가 뭔지 대충 감을 잡았을 것이다.

이번에는 (이후 서너 절에 걸쳐 계속 연구할) 좀 더 복잡한 예제를 들어볼 텐데, 본인이 정말 확실히 알고 있는지 스스로 점검해보자.

```
var a = 42;
var b = "foo";
var c = false;

var d = a && b || c ? c || b ? a : c && b : a;

d; // 결과는?
```

정말 나쁜 코드 아닌가? 인정한다. 이렇게 오른쪽으로 길게 늘어진 표현식을 쓰고 싶을 사람이 있을까? 가능성은 낮지만, 이 코드를 보면서 다수의 연산자를 체이닝(아주 흔한 작업이다)할 경우의 이슈를 연구해보자.

결과부터 말하면 42다. 그러나 이 코드를 자바스크립트 엔진에 꽂아 정렬도 시켜보지 않고 답이 뭘까 궁리하는 건 별 의미가 없다.

자, 자세히 알아보자.

첫째 질문은 (이런 의문조차 들지 않은 독자들도 있겠지만) '제일 앞의 (a && b || c)가 (a && b) || c와 a && (b || c) 중 어느 쪽으로 해석될까'이다. 정답을 알겠는가? 적어도 이 둘이 완전히 다른 연산이란 정도는 마음 속으로 느끼고 있는가?

```
(false && true) || true; // true
false && (true || true); // false
```

그렇다, 분명히 다르다. 그럼 false && true || true의 결과는?

```
false && true || true; // true
(false && true) || true; // true
```

&& 연산자가 먼저 평가되고 || 연산자가 그 다음에 평가된다.

처리 방향이 좌측 → 우측이라서? 순서를 거꾸로 뒤집어보자.

```
true || false && false; // true
(true || false) && false; // false—아니네!
true || (false && false); // true—맞다, 이겼다!
```

여기까지 해서 &&는 언제나 ||보다 먼저 평가된다는 사실이 증명되었고, 예상했던 좌측 → 우측과 정반대의 방향으로 처리됐다.

무슨 근거로? 연산자 우선순위 규칙이다.

모든 프로그래밍 언어에는 연산자 우선순위 리스트가 있다. 자바스크립트는 연산자 우선순위 리스트를 자바스크립트 개발자가 자세히 읽어보는 일 자체가 드물다는 것이 역설적으로 당황스럽다.

&&가 ||보다 우선하여 처리된다는 걸 잘 알고 있는 사람이라면, 이 절의 예제들이 파놓은 함정에 빠질 일은 없다. 하지만 장담컨대 꽤 많은 독자들이 순간 골똘히 생각했을 것이다.

아쉽게도 자바스크립트 명세서에는 한눈에 알아보기 쉽게 잘 정리된 연산자 우선순위 리스트가 없으므로 스스로 분석해보고 문법 규칙을 이해해야 한다. 그래서 언젠가 자주 쓰는 유용한 항목들을 좀 더 간편하게 찾아볼 수 있도록 정리할 생각이다. 전체 연산자 우선순위 리스트는 MDN 웹사이트 "연산자 우선순위"[09]를 참고하자.

5.2.1 단락 평가

4장 강제변환에서 &&, || 연산자의 "단락 평가short circuiting" 특성을 잠깐 언급했다. 이 절에서 자세히 알아보자.

09 역자주_ https://goo.gl/V1aikc

&&, || 연산자는 좌측 피연산자의 평가 결과만으로 전체 결과가 이미 결정될 경우, 우측 피연산자의 평가를 건너뛴다. 그래서 "단락short circuited"이란 말이 유래된 것이다(가능한 한 빨리 지름길을 택한다).

예를 들어 `a && b`에서 a가 `falsy`면 b는 쳐다보지도 않는다. && 연산 결과가 이미 `false`로 굳어진 마당에 애써 b를 조사할 필요가 없다. 마찬가지로 `a || b`에서 a가 `truthy`면, 이미 전체 결과값은 `true`로 확정되므로 b는 관심을 둘 이유가 없다.

단락 평가는 아주 유용하고 자주 쓰인다.

```
function doSomething(opts) {
if (opts && opts.cool) {
// ..
}
}
```

`opts && opts.cool`에서 opts는 일종의 가드다. 만약 opts가 존재하지 않는다면(아니면 객체가 아니라면) 당연히 `opts.cool` 표현식은 에러일 수밖에 없다. 일단 opts를 먼저 단락 평가해보고, 그 결과가 실패면 `opts.cool`는 자동으로 건너뛰니 결과적으로 에러는 나지 않는다.

|| 단락 평가도 마찬가지다.

```
function doSomething(opts) {
if (opts.cache || primeCache()) {
// ..
}
}
```

`opts.cache`를 먼저 체크해서 OK면, 굳이 `primeCache ( )` 함수는 호출하지 않고 넘어갈 수 있다. 그래서 불필요한 작업이 줄어든다.

5.2.2 끈끈한 우정

다음은 앞에서 예시했던, 연산자 뭉치가 죽 연결된, 복잡한 문이다. 삼항 연산자 ? : 부분을 잘 보기 바란다. ? :는 &&와 ||보다 우선순위가 높을까 아니면 낮을까?

```
a && b || c ? c || b ? a : c && b : a
```

즉, 다음 둘 중에서 어느 쪽으로 처리될까?

```
a && b || (c ? c || (b ? a : c) && b : a)
(a && b || c) ? (c || b) ? a : (c && b) : a
```

정답은 아랫 줄이다. 왜 그럴까?

&&는 ||보다, ||는 ? :보다 각각 우선순위가 높기 때문이다.

따라서 표현식 (a && b || c)가 ? :보다 먼저 평가된다. 즉, &&와 ||는 ? :보다 "서로 더 끈끈한 우정"을 나눈 사이라고 할 수 있다. 만약 그 반대였다면 c ? c...가 먼저 묶여 (윗줄처럼) a && b || (c ? c..)로 해석되었을 것이다.

5.2.3 결합성

연산자 우선순위 규칙에 의거하여 &&와 ||는 ? :보다 먼저 묶인다. 그럼, 우선순위가 동일한 다수의 연산자라면 처리 순서가 어떻게 될까? 항상 좌측 → 우측으로? 아니면 우측 → 좌측으로?

일반적으로 연산자는 좌측부터 그룹핑grouping이 일어나는지 우측부터 그룹핑이 일어나는지에 따라 좌측 결합성left-associative 또는 우측 결합성right-associative을 가진다.

여기서 결합성associativity이란 문제는 처리 방향이 좌측 → 우측이냐, 우측 → 좌측이냐 하는 문제와는 전혀 다르다는 사실을 기억해야 한다.

처리 방향이 좌측 → 우측인지, 우측 → 좌측인지 하는 문제가 왜 중요할까? 다음 함수 호출과 같이 표현식이 부수 효과를 일으킬 수 있기 때문이다.

```
var a = foo() && bar();
```

foo ( ) 함수를 먼저 호출한 뒤, 그 결과값에 따라 bar ( ) 호출 여부를 결정한다. 만일 bar ( )가 foo ( ) 앞에 있다면 전혀 다른 식으로 프로그램이 흘러갈 것이다.

foo ( ) 함수가 먼저 호출되는 건 단순히 좌측 → 우측 순서로 처리(자바스크립트의 기본 처리 순서다)되니 그런 것이지 &&의 결합성과는 무관하다. && 연산자는 하나뿐이고 그룹핑은 없으므로 결합성이 끼어들 여지가 없다.

그러나 a && b && c 같은 표현식에서는 암시적인 그룹핑이 발생한다. 즉, a && b와 b && c 중 어느 한 편이 먼저 평가된다.

&&는 좌측부터 결합하므로(||도 마찬가지) a && b && c는 (a && b) && c와 같다. 설사 우측부터 결합하여 a && (b && c)라도 결과가 같다. 동일한 값에 대하여 동일한 표현식은 항상 동일한 순서로 평가된다.

만약 &&가 우측부터 결합하는 연산자라고 하면 a && (b && c)처럼 여러분이 직접 ( )를 삽입하여 그룹핑한 것처럼 처리될 것이다. 하지만 그렇다고 c가 b보다 먼저 처리되는 건 아니다. 우측부터 결합한다는 말은 우측 → 좌측 순서로 평가가 아닌, 그룹핑을 그렇게 한다는 뜻이다. 그룹핑이든 결합이든 평가의 순서는 엄격히 a → b → c (좌측 → 우측)를 따른다.

따라서 정확한 정의를 따져 보자는 얘기가 아니라면 &&, ||이 좌측 결합성 연산자라는 건 그다지 중요하지 않다.

하지만 모든 연산자가 다 그런 건 아니다. 결합 방향이 좌/우측 어느 쪽인지에 따라 완전히 다르게 작동하는 연산자도 있다.

우선 ? : ("삼항" 또는 "조건") 연산자가 그렇다.

```
a ? b : c ? d : e;
```

? :는 우측 결합성 연산자인데, 그럼 다음 둘 중 정답은?

```
a ? b : (c ? d : e)
(a ? b : c) ? d : e
```

윗줄 a ? b :(c ? d : e)다. &&, ||와는 달리, 우측부터 결합하므로 결과가 달라진다. 이를테면 (a ? b : c) ? d : e는 다섯 개 값들의 조합에 따라(전부 다는 아니고) 상이하게 작동한다.

다음 조합도 그렇다.

```
true ? false : true ? true : true; // false

true ? false : (true ? true : true); // false
(true ? false : true) ? true : true; // true
```

결과값은 같은데 좀 더 복잡 미묘한 차이가 숨겨진 조합도 있다.

```
true ? false : true ? true : false; // false

true ? false : (true ? true : false); // false
(true ? false : true) ? true : false; // false
```

어차피 최종 결과는 같으니 그룹핑은 그다지 문제될 게 없어 보인다. 그러나,

```
var a = true, b = false, c = true, d = true, e = false;

a ? b : (c ? d : e); // false, 'a'와 'b'만 평가한다.
(a ? b : c) ? d : e; // false, 'a', 'b', 'e'를 평가한다.
```

이 예제로, 우측 결합성을 가진 ? : 연산자는 연쇄적으로 맞물릴 때 조심하지 않으면 문제가 될 수 있음을 알 수 있다.

=도 우측 결합성(그룹핑) 연산자 중 하나다. 전에 봤던 연쇄 할당문 예제를 보자.

```
var a, b, c;

a = b = c = 42;
```

a = b = c = 42는 c = 42 → b = .. → a = .. 순서로 처리된다고 설명했다. 우측부터 결합하기 때문에, 실제로 엔진은 a = (b = (c = 42))처럼 해석한다.

이 장 앞에서 보았던, 복잡한 할당 표현식으로 다시 돌아가보자.

```
var a = 42;
var b = "foo";
var c = false;

var d = a && b || c ? c || b ? a : c && b : a;

d; // 42
```

이제 우선 순위와 결합성을 배웠으므로, 할당 표현식 d는 다음과 같이 해석된다는 사실을 알 수 있다.

```
((a && b) || c) ? ((c || b) ? a : (c && b)) : a
```

알아보기 쉽게 들여쓰기를 해보자.

```
(
(a && b)
||
```

```
c
)
?
(
(c || b)
?
a
:
(c && b)
)
:
a
```

한 단계씩 풀어보자.

1. (a && b)는 "foo"다.
2. "foo" || c는 "foo"다.
3. 첫 번째 삼항 연산: "foo"는 truthy? → 맞다.
4. (c || b)는 "foo"다.
5. 두 번째 삼항 연산: "foo"는 truthy? → 맞다.
6. a는 42다.

여기까지! 답은 42다. 앞에서 실행했던 결과와 일치한다. 그리 어렵지는 않다.

5.2.4 분명히 하자

이제 연산자 우선 순위과 결합성을 확실히 이해했으니, 여러분은 다수의 연산자가 연쇄적으로 복잡하게 얽혀 있는 코드를 봐도 훨씬 편안한 기분으로 머릿속에 그려 볼 수 있을 것이다.

여기서 궁금한 점이 있다. 이 모든 연산자 우선 순위/결합성을 코드를 볼 사람이

모조리 다 꿰고 있다는 전제하에 코딩을 해야 할까? 임의로 처리 순서나 바인딩을 다르게 가져가고자 할 때에만 ( )로 손수 그룹핑을 하는 게 맞을까?

한편, 이런 규칙들이 배우고 익힐 만한 가치가 있다는 건 잘 알지만, 우선 순위/결합성 규칙이 자동으로 적용된다는 사실을 알지 못하여 함정에 빠질 수도 있다는 것 또한 인지해야 하지 않을까? 만약 그렇다면 엔진이 제멋대로 행동하지 않게 언제나 일일이 ( )로 감싸줘야 하지 않을까?

이 문제는 매우 주관적인 데다 4장 강제변환 만큼 지루한 논란의 대상이기도 하다. 개발자들의 생각도 대체로 비슷하다. 규칙에 따라 코딩하는 사람도 있고, 규칙은 무시한 채 명시적인 코딩을 하는 이들도 있다.

4장 강제변환과 마찬가지로 필자 역시 뭐라고 분명히 여러분에게 왈가왈부하긴 어렵다. 단지 찬반 양쪽의 의견을 고루 제시할 테니, 다른 사람들의 말에 휘둘리지 않고 제대로 이해한 뒤 여러분이 직접 판단하기 바란다.

개인적으로는 중용이 중요하다고 생각한다. 즉, 연산자 우선 순위/결합성과 손으로 ( )를 감싸 주는 두 방법을 여러분의 프로그램에 적절히 배합하는 것이다(4장 강제변환에서 암시적 강제변환의 건전한/안전한 사용에 대해 언급할 때에도 필자는 암시적 강제변환만이 진리라고 떠받들진 않았다).

예를 들어, 필자에게 `(a && b && c)`는 그 자체로 최선이므로 굳이 결합성을 고려하여 순서를 명시하고자 `((a && b) && c)`처럼 쓰는 건 장황한 코드를 낳을 뿐이므로, 그냥 놔둘 것이다.

반면, 두 개의 `? :` 조건 연산자가 체이닝된 코드가 있다면, 주저 없이 ( )로 그룹핑하여 필자가 의도한 로직을 확실히 밝힐 것이다.

결론은 4장 강제변환과 대동소이하다. 연산자 우선 순위/결합성을 적절히 활용하여 짧고 깔끔한 코드를 작성하되 혼동을 줄이고 분명히 밝혀야 할 부분은 ( )로 예쁘게 감싸 주기 바란다.

5.3 세미콜론 자동 삽입

ASI(Automatic Semicolon Insertion, 자동 세미콜론 삽입)는 자바스크립트 프로그램의 세미콜론(;)이 누락된 곳에 엔진이 자동으로 ;을 삽입하는 것을 말한다.

애써 이런 수고를 대신하는 이유는, 단 하나의 ;이라도 누락되면 자바스크립트 프로그램은 돌아가지 않기 때문이다(생각보다 엔진이 그리 관대하지 않다). 자바스크립트 코딩 시 ;을 안 써도 될 것 같은 부분에 생략을 해도 프로그램이 실행되는 이유는 모두 ASI 덕분이다.

단, ASI는 새 줄(행 바꿈(line break))에만 적용되며, 어떠한 경우에도 줄 중간에 삽입되는 일은 없다.

기본적으로 자바스크립트 파서는 줄 단위로 파싱을 하다가 (;이 빠져) 에러가 나면 ;을 넣어보고 타당한 것 같으면(즉, 문의 끝 부분과 새 줄/행 바꿈 문자 사이에 공란이나 주석밖에 없으면) ;를 삽입한다.

```
var a = 42, b
c;
```

둘째 줄 c를 var 문의 연장으로 봐야 맞을까? a가 b, c 사이에 (차라리 다른 줄에라도) 있다면 그렇게 취급해야 맞겠지만, 실제 사정은 그렇지 않으므로 엔진은 b 뒤에 (새 줄에) 암시적으로 ;을 삽입한다. 따라서 c;는 독립적인 표현식 문이 된다.

다음 코드도 비슷하다.

```
var a = 42, b = "foo";

a
b // "foo"
```

표현식 문에도 ASI가 잘 적용되므로 에러 없는 유효한 프로그램이다.

다음 예제를 보면 ASI가 참 유용하다는 사실을 알 수 있다.

```
var a = 42;

do {
// ..
} while (a) // <— ;를 빼먹었군!
a;
```

while, for 루프와 달리 do..while 루프는 마지막에 ;을 넣어야 문법적으로 맞다. 하지만 모든 개발자가 다 똑똑하지는 않다. ASI는 조용히 끼어들어 친절하게 ;을 넣어준다.

앞서 얘기했듯이, 문 블록에서 ;는 필수가 아니므로 ASI 역시 필요 없다.

```
var a = 42;

while (a) {
// ..
} // <— ;는 굳이 필요 없군!
a;
```

ASI는 주로 break, continue, return, yield(ES6부터) 키워드가 있는 곳에서 활약한다.

```
function foo(a) {
if (!a) return
a *= 2;
// ..
}
```

return 문이 다음 줄 a *= 2 표현식으로 건너갈 일은 없으니 ASI는 return 문 끝에 ;을 추론하여 삽입한다. 물론, 다음과 같이 return 문을 여러 줄에 걸쳐 표

기할 수도 있는데, return 다음에 새 줄/행 바꿈 문자만 있는 경우는 제외된다.[10]

```
function foo(a) {
return (
a * 2 + 3 / 12
);
}
```

break, continue, yield에도 동일한 추론 로직이 적용된다.

5.3.1 에러 정정

(탭이냐 공란이냐 하는 문제 외에) 자바스크립트 커뮤니티에서 가장 뜨거운 (종교 전쟁을 방불케 하는) 논란거리는 ASI에 전적으로/완전히 의존해야 하는가이다.

대부분 세미콜론은 선택 사항이다(for(; ;)처럼 필수 세미콜론도 있다).

ASI 찬성 측 개발자들은 (몇 가지 안 되는) 세미콜론을 반드시 넣어야 하는 경우만 제외하고 생략하여 쓰면 ASI가 보다 간결한(그리고 더 "우아한") 코드를 작성하게 도와주는 아주 유용한 도구라고 주장한다. ASI를 잘 활용하면 상당수 ;는 선택적으로 쓸 수 있기 때문에 ; 없이 작성한 프로그램도 ;을 분명히 기재하여 작성한 프로그램이나 별반 다를 바 없다는 얘기다.

반면, 반대측에서는 마치 마법사가 마술을 부린 양 개발자가 의도하지 않은 ;들이 삽입되면서 그 의미가 달라질 수 있고, 그러다 보면 특히 경험이 부족한 비숙련 개발자들의 경우 실수할 가능성이 높다고 말한다. 세미콜론을 누락했다면 이는 순전

10 역자주_ 즉, 다음과 같이 ( )를 생략하면 return;으로 해석하여 undefined가 반환됩니다.

```
function foo(a) {
return
a * 2 + 3 / 12
;
}
```

히 개발자의 실수에서 비롯된 것이니 자바스크립트 엔진이 은밀하게 실수를 바로잡기 전에 관련 툴(린터linters 등)로 분명히 정정해야 한다고 얘기하는 사람들도 있다.

필자의 생각은 이렇다. 명세에는 분명히 ASI가 "에러 정정error correction" 루틴이라고 씌어 있다. 여기서 에러란 구체적으로는 파서 에러parser error다. 다른 말로 풀이하면 ASI가 파서를 너그럽게 하여 에러를 줄이는 것이다.

파서 에러는 프로그램을 부정확하게/잘못 코딩했기 때문에 나는 것일 뿐, 그 외의 경우는 없다. 따라서 ASI가 꼼꼼히 파서 에러를 정정했음에도 발생한 파서 에러는, 프로그램 작성자가 정말 (문법에 따라 반드시 넣어야 할 세미콜론이 누락된 것과 같은) 잘못 짠 코드가 있다는 증거다.

그러므로 단도직입적으로 말해서, 누군가 "세미콜론은 써도 되고 안 써도 되니 나는 그냥 생략하련다"고 말하는 건 꼭 내 귀엔 "프로그램이 잘 돌아가기만 하면 그만이니 나는 최대한 파서를 깨뜨리는 프로그램을 작성하겠다"는 소리로 들린다.

타이핑을 아끼면서 "아름다운 코드"를 작성하겠다는 발상 자체가 터무니없다.

더구나 이 문제는 (순전히 코드의 외양에 관련된) 탭이냐 공란이냐의 문제와는 분명히 다르며, '코드를 문법 요건에 맞게 작성할 것인가' 아니면 '가까스로 비껴 지나갈 수 있을 정도로 문법 예외 사항에 의존할 것인가'에 관한 근본적인 질문이다.

ASI의 기능에 의존하는 것은 본질적으로 새 줄을 "유효 공백문자significant whitespace"로 바라보는 것과 같다는 의견도 있다. 파이썬Python 같은 언어에는 진짜 유효 공백문자가 있긴 하다. 그러나 현재의 자바스크립트에 유효 공백문자 같은 게 있을까?

어쨌건 나의 결론은 이렇다. "필요하다고" 생각되는 곳이라면 어디든지 세미콜론을 사용하고, ASI가 어떻게든 뭔가 해줄 거라는 가정은 최소화하자.

하지만 그렇다고 내 말을 무조건 수용하라는 건 아니다. 2012년, 자바스크립트의

창시자, 브렌단 아이크Brendan Eich는 다음과 같이 밝힌 바 있다.

ASI는 (공식적으로는) 구문 오류를 정정하는 프로시저다. 만약 여러분이 ASI를 보편적인, 유효 개행문자significant-newline 규칙쯤으로 여기고 코드를 작성한다면 곤경에 처하게 될 것이다... 내가 1995년 5월의 열흘 간으로 되돌아갈 수만 있다면 강력한 유효 개행문자를 만들 것이다... ASI가 마치 유효 개행문자를 넣어주는 것처럼 착각하지 않기 바란다.

5.4 에러

자바스크립트에는 하위 에러 타입(`TypeError`, `ReferenceError`, `SyntaxError` 등)뿐만 아니라, (에러는 대부분 런타임 시점에 발생하지만) 일부 에러는 컴파일 시점에 발생하도록 문법적으로 정의되어 있다.

특히 자바스크립트에는 오랜 전부터 (컴파일 도중) "조기 에러early error"로 붙잡아 던지게 되어 있는, 여러 가지 특정 조건들이 있었다. 한눈에 봐도 알 수 있는 구문 에러(예: `a = ,`)는 물론이고, 자바스크립트 문법에는 구문상 오류는 아니지만 허용되지 않는 것들도 정의되어 있다.

코드가 실행도 되기 전에 발생하므로 이런 에러는 `try..catch`로 잡을 수 없으며, 그냥 프로그램 파싱/컴파일이 실패한다.

명세는 브라우저(그리고 개발자 툴)가 에러 리포팅을 하는 방법에 대해 딱히 구체적으로 정해놓은 건 없다. 따라서 지금부터 볼 예제는 브라우저에서 실행하면 하위 에러 타입이나 메시지가 조금씩 다를 수 있다.

다음 정규 표현식 리터럴 내부의 구문이 그런 예다. 자바스크립트 구문상 아무 문제가 없지만, 올바르지 않은 정규 표현식은 조기 에러를 던진다.

```
var a = /+foo/; // 에러!
```

할당 대상은 반드시 식별자(또는 하나 이상의 식별자를 가져오는 ES6 분해 표현식)여야 하므로 다음 예제에서 42는 잘못된 위치에 있기 때문에 곧바로 에러가 난다.

```
var a;
42 = a; // 에러!
```

ES5 엄격 모드는 조기 에러가 더 많다. 예컨대, 엄격 모드에서 함수 파라미터명은 중복될 수 없다.

```
function foo(a,b,a) { } // 정상 실행

function bar(a,b,a) { "use strict"; } // 에러!
```

동일한 이름의 프로퍼티가 여러 개 있는 객체 리터럴도 사정은 비슷하다.

```
(function(){
"use strict";

var a = {
b: 42,
b: 43
}; // 에러!
})();
```

의미만 놓고 보자면 이런 에러는 엄밀히 구문 에러가 아니라 문법 에러에 가깝다. 실제로 구문 자체는 문제가 없는 코드들이다. 하지만 그렇다고 `GrammarError`라는 에러 타입이 따로 있지는 않으므로 브라우저는 대신 `SyntaxError`를 빌려 쓴다.

5.4.1 너무 이른 변수 사용

ES6는 (솔직히 이름이 좀 헷갈리지만) TDZ("임시 데드 존Temporal Dead Zone")라는 새로운 개념을 도입했다.

TDZ는 아직 초기화를 하지 않아 변수를 참조할 수 없는 코드 영역이다.

ES6 let 블록 스코핑이 대표적인 예다.

```
{
a = 2; // ReferenceError!
let a;
}
```

let a 선언에 의해 초기화되기 전 a = 2 할당문이 ({ .. } 블록 스코프 안에 있는) 변수 a에 접근하려고 한다. 하지만 a는 아직 TDZ 내부에 있으므로 에러가 난다.[11]

재밌는 사실은, 원래 typeof 연산자는 선언되지 않은 변수 앞에 붙여도 오류는 나지 않는데 TDZ 참조 시에는 이러한 안전 장치가 없다.[12]

```
{
typeof a; // undefined
typeof b; // ReferenceError! (TDZ)
let b;
}
```

5.5 함수 인자

TDZ 관련 에러는 ES6 디폴트 파라미터 값에서도 찾아볼 수 있다(『You Don't Know JS: ES6 & Beyond』[13] 참고).

11 역자주_ let a;를 실행 후에 비로소 TDZ가 끝나고 a에 undefined가 할당됩니다.

12 역자주_ TDZ 내에 위치하여 접근할 수 없으니 typeof 적용하는 것조차 허용되지 않는 것입니다.

13 http://goo.gl/GqxITA

```
var b = 3;

function foo( a = 42, b = a + b + 5 ) {
// ..
}
```

두 번째 할당문에서 좌변 b는 아직 TDZ에 남아 있는 b를 참조하려고 하기 때문에 (더 바깥쪽에서 끌어오지 못하고) 에러를 던진다. 그러나 이 시점에서 파라미터 a는 TDZ를 밟고 간 이후여서 문제가 없다.[14]

ES6 디폴트 파라미터 값은 함수에 인자를 넘기지 않거나 undefined를 전달했을 때 적용된다.

```
function foo( a = 42, b = a + 1 ) {
console.log( a, b );
}

foo(); // 42 43
foo( undefined ); // 42 43
foo( 5 ); // 5 6
foo( void 0, 7 ); // 42 7
foo( null ); // null 1
```

a + 1 표현식에서 null은 0으로 강제변환된다(4장 강제변환 참고).

ES6 디폴트 파라미터 입장에서 보면 인자 값이 없거나 undefined 값을 받거나 그게 그거다. 하지만 차이점을 엿볼 수 있는 경우도 있다.

14 역자주_ 함수 파라미터의 디폴트 값은 마치 하나씩 좌측 → 우측 순서로 let 선언을 한 것과 동일하게 처리됩니다. 그래서 일단 무조건 TDZ에 속하게 됩니다.

```
function foo( a = 42, b = a + 1 ) {
console.log(
arguments.length, a, b,
arguments[0], arguments[1]
);
}

foo(); // 0 42 43 undefined undefined
foo( 10 ); // 1 10 11 10 undefined
foo( 10, undefined ); // 2 10 11 10 undefined
foo( 10, null ); // 2 10 null 10 null
```

아무런 인자도 넘기지 않았을 때 디폴트 파라미터 값이 a, b에 각각 적용되었지만 arguments 배열에는 원소가 하나도 없다.

그런데 undefined 인자를 명시적으로 넘기면 arguments 배열에도 값이 undefined 인 원소가 추가되는데, 여기에 해당하는 디폴트 파라미터 값과 다르다.

ES6 디폴트 파라미터 값이 arguments 배열 슬롯과 이에 대응하는 파라미터 값 간의 불일치를 초래하는 것은 사실이지만, 이전 버전 ES5에서도 똑같은 불일치는 교묘하게 발생한다.

```
function foo(a) {
a = 42;
console.log( arguments[0] );
}

foo( 2 ); // 42 (연결된다)
foo(); // undefined (연결되지 않는다)
```

인자를 넘기면 arguments의 슬롯과 파라미터가 연결되면서 항상 같은 값이 할당되지만, 인자 없이 호출하면 전혀 연결되지 않는다.

더욱이 엄격 모드에서는 어떻게 해도 연결되지 않는다.

```
function foo(a) {
"use strict";
a = 42;
console.log( arguments[0] );
}

foo( 2 ); // 2 (연결되지 않는다)
foo(); // undefined (연결되지 않는다)
```

이렇게 확실하지 않은 연결에 의존하여 코딩하는 것은 바람직하지 않은 것 같다. 사실 연결 그 자체만 봐도 잘 설계된 기능이라기보다 엔진 내부의 상세한 구현체를 노출시킨, 구멍 난 추상화leaky abstraction다.

arguments 배열은 이제 비 권장 요소지만(특히 ES6부터는 ... 레스트rest 파라미터를 권장한다 - 『You Don't Know JS: ES6 & Beyond』[15] 참고), 그렇다고 완전히 나쁜 것은 아니다.

ES6 이전까지 arguments는 함수에 전달된 인자들을 배열 형태로 추출할 수 있는 유일한 방법이었고 제법 유용하게 잘 써왔다. "파라미터와 이 파라미터에 해당하는 arguments 슬롯을 동시에 참조하지 마라"는 간단한 규칙만 준수한다면, arguments 배열과 파라미터를 섞어 사용해도 안전하다. 나쁜 습관만 피한다면 구멍 난 추상화 역시 모습을 드러낼 리 없다.

```
function foo(a) {
console.log( a + arguments[1] ); // 안전하다!
}

foo( 10, 32 ); // 42
```

15 http://goo.gl/GqxITA

5.6 try..finally

try..catch 블록의 사용 방법은 다들 익숙할 것이다. 그런데 이 블록에 finally 절을 같이 사용하면 어떻게 작동할지 한 번쯤 생각해본 적이 있는지? catch, finally 둘 다 같이 써도 상관은 없지만, 원래 try 이후에는 catch, finally 중 하나만 필수라는 얘기는 금시초문 아닌지?

finally 절의 코드는 (어떤 일이 있어도) 반드시 실행되고, 다른 코드로 넘어가기 전에 try 이후부터 (catch가 있으면 catch 다음부터) 항상 실행된다. 어떤 의미에서 finally 절은 다른 블록 코드에 상관없이 필히 실행되어야 할 콜백 함수와 같다고 봐야 맞다.

그런데 만약 try 절에 return 문이 있으면? 당연히 어떤 값을 반환할 텐데, 그 값을 반환 받은 호출부 코드는 finally 이전에 실행될까? 아니면 이후에 실행될까?

```
function foo() {
try {
return 42;
}
finally {
console.log( "Hello" );
}

console.log( "실행될 리 없지!" );
}

console.log( foo() );
// Hello
// 42
```

return 42에서 foo( ) 함수의 완료 값은 42로 세팅되고, try 절의 실행이 종료되면서 곧바로 finally 절로 넘어간다. 그 후 foo( ) 함수 전체의 실행이 끝나고 완료 값은 호출부 console.log(..) 문에 반환된다.

try 안에 throw가 있어도 비슷하다.

```
function foo() {
try {
throw 42;
}
finally {
console.log( "Hello" );
}

console.log( "실행될 리 없지!" );
}

console.log( foo() );
// Hello
// Uncaught Exception: 42
```

만약 finally 절에서 (사고든 고의든) 예외가 던져지면, 이전의 실행 결과는 모두 무시한다. 즉, 이전에 try 블록에서 생성한 완료 값이 있어도 완전히 사장된다.

```
function foo() {
try {
return 42;
}
finally {
throw "어이쿠!";
}

console.log( "실행될 리 없지!" );
}

console.log( foo() );
// Uncaught Exception: 어이쿠!
```

continue나 break 같은 비선형nonlinear 제어문 역시 return과 throw와 비슷하게 작동한다.

```
for (var i=0; i<10; i++) {
try {
continue;
}
finally {
console.log( i );
}
}
// 0 1 2 3 4 5 6 7 8 9
```

continue 문 때문에 console.log(i) 문은 루프 순회 끝 부분에서 실행된다. 그러나 i++로 인덱스가 수정되기 직전까지 꾸준히 실행되므로 1..10이 아닌 0..9이 콘솔 창에 표시된다.[16]

ES6부터 도입된 yield 문은 어떤 점에서는 꼭 중간intermediate return 문 같은 발생자generator다(『You Don't Know JS: Async & Performance』[17] 참고). 하지만 return과는 달리 yield는 발생자가 재개되기 전까지, 즉 try { .. yield .. } 실행이 끝나기 전까지 완료되지 않는다. 그래서 뒤에 finally 절을 추가해도 return처럼 yield 직후에 실행되지 않는다.

finally 절의 return은 그 이전에 실행된 try나 catch 절의 return을 덮어쓰는 특출한 능력을 갖고 있는데, 단 반드시 명시적으로 return 문을 써야 한다.

```
function foo() {
try {
return 42;
}
finally {
// 여기 'return ..'이 없군, 그럼 이전 return을 존중하자!
}
}
```

16 역자주_ 무슨 말인지 정확히 이해가 되지 않는다면, 예제 코드의 continue; 앞에 console.log 함수로 i 값을 출력해보시기 바랍니다.

17 http://goo.gl/nSOxEo

```
function bar() {
try {
return 42;
}
finally {
// 'return 42'를 무시한다.
return;
}
}

function baz() {
try {
return 42;
}
finally {
// 'return 42'를 무시한다.
return "Hello";
}
}

foo(); // 42
bar(); // undefined
baz(); // Hello
```

보통 함수에서는 return을 생략해도 return; 또는 return undefined;한 것으로 치지만, finally 안에서 return을 빼면 이전의 return을 무시하지 않고 존중한다.

레이블 break와 finally가 만나면 그야말로 장관이다.

```
function foo() {
bar: {
try {
return 42;
}
finally {
// 'bar' 레이블 블록으로 나간다.
break bar;
```

```
}
}

console.log( "미쳤군!" );

return "Hello";
}

console.log( foo() );
// 미쳤군!
// Hello
```

부디 이런 코딩은 피하자. 진심이다. 사실상 return을 취소해버리는 'finally + 레이블 break' 코드는 그냥 골치 아픈 코드를 양산할 뿐이다. 장담컨대, 주석을 한 트럭 달아놓아도 이런 코드는 마냥 어렵다.

5.7 switch

마지막으로 if..else if..else.. 문의 사슬을 짧게 해주는 switch 문을 간략히 살펴보자.

```
switch (a) {
case 2:
// 뭔가 할테고
break;
case 42:
// 다른 일을 할테고
break;
default:
// 아무것도 안 걸리면 여기로!
}
```

보다시피 switch 표현식의 평가 결과를 각 case 표현식의 값들(여기서는 그냥 단

순 값이다)과 매치한다. 죽 내려가다 매치된 case 절의 코드를 실행하기 시작하여 break 문을 만나기 전이나 switch 블록의 끝까지 계속 나아간다.

여기까진 지나치게 평범하지만 switch에는 여러분이 예전에는 몰랐을 기벽이 있다.

첫째, switch 표현식과 case 표현식 간의 매치 과정은 `===` 알고리즘(4장 강제변환 참고)과 똑같다. 예제처럼 case 문에 확실한 값이 명시된 경우라면 엄격한 매치가 적절하다.

그러나 강제변환이 일어나는 동등 비교(==, 4장 강제변환 참고)를 이용하고 싶다면 switch 문에 꼼수를 좀 부려야 한다.

```
var a = "42";

switch (true) {

case a == 10:
console.log( "10 또는 '10'" );
break;
case a == 42:
console.log( "42 또는 '42'" );
break;
default:
// 여기 올 일은 없지!
}
// 42 또는 '42'
```

어쨌든 case 절에 (단순 값은 아니지만) 표현식이 있으니 실행상 문제는 없다. 즉, switch 표현식의 결과(true)와 case 표현식의 결과를 엄격하게 매치한다. a == 42는 true이므로 여기서 매치된다.

==를 써도 switch 문은 엄격하게 매치한다. 그래서 case 표현식 평가 결과가 truthy이지만 엄격히 true는 아닐 경우(4장 강제변환 참고) 매치는 실패한다. 이를테면 표현식에 &&나 || 같은 "논리 연산자"를 사용할 때 문제가 된다.

```
var a = "Hello world";
var b = 10;

switch (true) {
case (a || b == 10):
// 여기 올 일은 없지!
break;
default:
console.log( "어이쿠" );
}
// 어이쿠
```

(a || b == 10) 평가 결과는 true가 아닌 "Hello world"이므로 매치가 되지 않는다. 이때는 분명히 표현식의 결과가 true/false로 떨어지게 case !!(a || b == 10): (4장 강제변환 참고)와 같이 작성해야 한다.

끝으로 default 절은 선택 사항이며 (오래된 관습이긴 하지만) 꼭 끝 부분에 쓸 필요는 없다. 그런데 default에서도 break를 안 써주면 그 이후로 코드가 계속 실행된다.

```
var a = 10;

switch (a) {
case 1:
case 2:
// never gets here
default:
console.log( "default" );
case 3:
console.log( "3" );
break;
case 4:
console.log( "4" );
}
// default
// 3
```

앞서 레이블 break에서도 설명했듯이 case 절에서도 레이블 break를 쓸 수 있다.

case 절들을 훑다가 매치되는 게 없으니 결국 default 절 실행을 시작하는데, break가 없으니 이미 이전에 한 번 지나친 case 3: 블록을 다시 실행하게 된다.

이런 식으로 에두르는 로직이 자바스크립트에서 가능하긴 하지만, 그런 로직으로 짠 코드가 합리적이고 이해하기 쉬울 리 만무하다. 만약 그런 환형 로직을 피할 수 없는 상황이라면 왜 그래야 하는지 진지한 의문을 가져보고, 그래도 정말 불가피하다면 자신의 의도가 명백히 드러나도록 주석을 충분히 달아놓기 바란다!

5.8 정리하기

자바스크립트 문법에는 우리 같은 개발자들이 평소 하던 것보다 더 많은 시간을 들여 관심 있게 봐야 할 오묘한 것들이 꽤 많다. 약간의 노력만 기울이면 여러분도 좀 더 깊이 있는 자바스크립트 지식을 다지는 데 큰 도움이 될 것이다.

문과 표현식은 영어 언어의 문장, 어구와 각각 유사하다. 표현식은 순수하고 독립적이지만 부수 효과를 일으킬 수 있다.

자바스크립트 문법에는 순수 구문 외에 의미론적인 사용 규칙(콘텍스트)이 내재되어 있다. 예를 들어, 프로그램에서 자주 등장하는 { } 쌍은 문 블록, 객체 리터럴이 될 수도 있고, 해체 할당(ES6)이나 명명된 함수 파라미터(ES6)로 쓸 수도 있다.

자바스크립트 연산자는 그 우선 순위(어떤 것이 다른 것보다 먼저 묶여지는지)와 결합성(여러 연산자 표현식이 암시적으로 그룹핑되는 방법)이 분명히 정해져 있다. 일단 우선 순위/결합성 규칙을 잘 알아두기 바라며, 두 규칙이 무턱대고 사용하기엔 너무 암시적이라 사용을 자제할지 아니면 짧고 깔끔한 코딩에 도움이 되니 적극 활용할지 판단은 알아서 하자.

ASI(자동 세미콜론 삽입)는 자바스크립트 엔진에 내장된 "파서 에러 감지 시스템"으로 필요한 ;이 코드에서 누락된 경우 파서 에러가 나면 자동으로 삽입해보고 코드 실행에 문제가 없도록 도와준다. 이런 장치가 준비되어 있어서 대부분의 ;을 선택적으로 쓰는 편이 좋을지(깔끔한 코드를 위해서 쓰지 말자!) 원래 넣어야 ;을 개발자가 실수로 빠뜨린 것을 엔진이 대신 뒤처리를 해주는 것에 불과한지 의견이 분분하다.

자바스크립트 에러는 몇 가지 유형이 있지만 크게 "조기 에러"(컴파일러가 던진 잡을 수 없는 에러)와 "런타임 에러"(try..catch로 잡을 수 있는 에러)로 분류된다는 사실은 별로 잘 알려져 있지 않다. 모든 구문 에러는 프로그램을 실행 전 중단시키는 조기 에러가 분명하지만, 다른 유형의 에러들도 있다.

함수 arguments와 명명된 파라미터의 관계는 흥미롭다. arguments 배열은 조심하지 않으면 구멍 난 추상화에서 비롯된 갖가지 함정에 빠질 수 있다. 가급적 arguments 사용을 자제하되 꼭 사용해야 할 경우 arguments의 원소와 이에 대응하는 명명된 파라미터를 동시에 사용하지 말자.

try(또는 try..catch)에 붙는 finally 절에는 실행 처리 순서 면에서 별난 기벽이 있다. 때로는 이런 기벽이 도움이 되기도 하지만, 레이블 블록과 함께 사용하면 많은 혼란을 가중시킬 수 있다. 교묘하지만 헷갈리는 코드를 작성하지 말고 finally로 좋은 코드, 깔끔한 코드를 작성하기 바란다.

switch는 장황한 if..else if.. 문을 대체하는 훌륭한 수단이지만, 단순하게만 생각했다간 예기치 않은 결과에 당황할 수 있다. 조심하지 않으면 여러분을 고생시킬 기벽들이 곳곳에 숨어있지만, 꽤 그럴듯한 트릭도 함께 지니고 있다.

부록
다양한 환경의 자바스크립트

실제 서비스 환경에서 실행되는 자바스크립트 코드는 지금까지 우리가 자세히 살펴봤던 언어의 핵심 메커니즘을 벗어나 여러분의 예상과 다르게 작동할 수 있다. 오로지 자바스크립트 엔진 내부에서만 실행된다면야 명세의 내용에 근거하여 결과를 정확히 예측할 수 있겠지만, 자바스크립트 프로그램은 거의 항상 호스팅 환경의 콘텍스트에서 실행되므로 예측하기 어려운 부분이 어느 정도 존재한다.

이를테면, 작성한 코드를 다른 소스 코드와 함께 실행하거나 (브라우저 이외의) 다른 유형의 자바스크립트 엔진에서 실행하면 예상과 다르게 작동할 수 있다.

부록에서는 이 문제를 간략히 다루고자 한다.

부록.1 부록 B(ECMAScript)

자바스크립트의 공식적인 언어 명칭이 ECMAScript(ECMA는 표준 관리 주체 기관임)라는 사실은 그리 잘 알려져 있지 않다. 그럼 "자바스크립트"는? 자바스크립트는 ECMAScript 언어의 상품명으로 통용되며, 더 정확하게는 명세의 브라우저 구현체다.

공식 ECMAScript 명세 "부록 B"[01]는 브라우저의 자바스크립트와의 호환성을 염두에 두고 공식 명세와의 특정한 차이점deviation을 기술한다.

01 http://goo.gl/O7uQkY

이러한 차이점은 브라우저에서 코드를 실행할 경우에만 존재/해당되며, 여러분이 코드를 항상 브라우저에서만 실행한다면 눈에 띄는 차이점을 찾기는 어려울 것이다. 그렇지 않은 경우(예: 노드JS, 리노Rhino 등에서 실행) 또는 여러분 스스로 확실치 않다면 신중을 기하기 바란다.

다음은 주요 호환성 차이compatibility difference를 열거한 것이다.

- `0123`(10진수 83)와 같은 8진수 리터럴은 느슨한 모드에서 사용할 수 있다.
- `window.escape(..)`/`window.unescape(..)`를 이용하여 '문자열 ↔ % 기호로 구분된 16진수 이스케이프 시퀀스escape sequence' 변환escape/역변환unescape이 가능하다(예: `window.escape( "?foo=97%&bar=3%" )` → "%3Ffoo%3D97%25%26bar%3D3%25").
- `String.prototype.substr`은 `String.prototype.substring`과 유사하나, 두 번째 파라미터는 종료 인덱스(포함되지 않음)가 아닌, 길이(문자열의 문자 개수)를 의미한다.

부록. 1.1 웹 ECMAScript

웹 ECMAScript 명세[02]는 공식 ECMAScript 명세와 현재 브라우저 자바스크립트 구현체와의 차이점을 기술한 문서다.

즉, 이 문서에 기술된 항목들은 (다른 브라우저와의 호환을 위한) 브라우저의 "필수 요건"이며, 공식 명세 "부록 B"에는 (이 책을 집필하는 현재) 없는 것들이다.

- `<!--` and `-->`는 한 줄짜리 유효한 주석 구분자delimiter다.
- `String.prototype` 추가(HTML 형식의 문자열 반환 기능) : `anchor(..)`, `big(..)`, `blink(..)`, `bold(..)`, `fixed(..)`, `fontcolor(..)`, `fontsize(..)`, `italics(..)`, `link(..)`, `small(..)`, `strike(..)`, `sub(..)`

이들 메서드를 직접 쓸 일은 매우 드물며, 내장 `DOM API` 또는 사용자 정의 유틸리티로 대체하고 가급적 사용하지 말자.

02 https://javascript.spec.whatwg.org/

- RegExp 확장: RegExp.$1 .. RegExp.$9(그룹 매치), RegExp.lastMatch/RegExp[" $& "](가장 최근 매치)
- Function.prototype 추가 : Function.prototype.arguments(arguments 객체의 별칭alias), Function.caller(arguments.caller의 별칭)

arguments/arguments.caller는 이제 권장하지 않으니 가능한 한 사용을 피하자. 별칭이라고 다르지 않으니 사용을 자제하라!

그 밖의 사소하거나 거의 쓸 일이 없는 것들은 제외했다. 전체 내용은 "부록 B"와 "웹 ECMAScript" 문서를 찾아보자.

결론적으로, 이 절에서 소개한 내용은 이제는 거의 쓰지 않는 것들이므로 크게 신경쓰지 말자. 혹시라도 여러분이 작성한 코드가 연관될 경우에 한하여 참고해두자.

부록. 2 호스트 객체

잘 만들어진 자바스크립트 변수 관련 규칙도 (코드 호스팅 환경이 제공하거나 생성하는) 자동 정의auto-defined 변수, 이른바 "호스트 객체host objects"(내장 객체/함수 포함)에 관해서라면 예외다.

예를 들어,

```
var a = document.createElement( "div" );

typeof a; // "object" — 예상대로군!
Object.prototype.toString.call( a ); // "[object HTMLDivElement]"

a.tagName; // "DIV"
```

a는 일반 객체가 아니라 DOM 요소를 가리키는 특별한 호스트 객체다. 따라서 내부

[[Class]] 값(“HTMLDivElement”)이 다르고, 미리 정의된 (종종 변경이 불가능한) 프로퍼티를 가진다.

“Falsy 객체”의 기벽을 기억하는가? 명백히 실존하는 객체라서 불리언 값으로 강제변환하면 true가 나올 것 같은데, (황당하게도) false가 되어버리는 녀석들이다.

- 호스트 객체를 다룰 때 조심해야 할 유의 사항을 정리한다.
- toString() 같은 일반 객체의 내장 메서드에 접근할 수 없다.
- 덮어쓸 수 없다.
- 미리 정의된, 읽기 전용 프로퍼티를 가진다.
- 다른 객체로 this를 재정의할 수 없는 메서드가 있다.
- 기타 등등.

자바스크립트 코드가 주변 환경과 잘 어울리고 제대로 실행되려면 호스트 객체가 꼭 필요하다. 그러나 대부분 일반적인 자바스크립트 객체의 기준을 따르지 않으므로 호스트 객체를 사용하여 뭔가 하려고 할 때에는 이들이 어떻게 작동하는지 세심하게 살펴야 한다. 꼭 기억하기 바란다.

가장 자주 쓰는 호스트 객체로는 console 객체와 부속 함수들(log(..), error(..) 등)이 있다. 호스팅 환경이 독자적으로 제공하는 console 객체는 개발 과정의 다양한 결과 출력 용도로 쓰인다.

브라우저에서는 console이 개발자 툴의 콘솔 창과 연결되어 있지만, 노드JS 및 기타 서버-사이드 자바스크립트 환경에서는 대개 자바스크립트 환경 시스템 프로세스의 표준 출력(stdout), 표준 에러(stderr) 스트림과 연결된다.

부록. 3 전역 DOM 변수

전역 스코프에서 (var를 붙이거나 var 없이) 변수를 선언하면 전역 변수 하나만 달랑 생성되는 게 아니라 전역 객체(브라우저는 window)에도 동일한 이름의 프로퍼티가

마치 거울처럼 만들어진다는 건 아마 잘 알고 있을 것이다.

하지만 (구식 브라우저 특성 탓에) id 속성으로 DOM 요소를 생성해도 같은 이름을 가진 전역 변수가 생성된다는 사실은 모르는 사람들이 많다. 예를 들면,

```
<div id="foo"></div>
```

그리고 자바스크립트 코드에서,

```
if (typeof foo == "undefined") {
foo = 42; // 절대 실행되지 않겠지...
}

console.log( foo ); // HTML 요소
```

전역 변수는 자바스크립트 코드를 통해서만 생성된다는 섣부른 전제하에 (typeof 또는 window 객체로) 전역 변수 여부를 체크하면 그만이라 생각하겠지만, 예제로부터 알 수 있듯이 HTML 페이지 역시 전역 변수를 생성할 수 있으니 조심하지 않으면 기존의 체크 로직은 휴지 조각이 되어버릴 수 있다.

따라서 이런 이유 때문에라도 가급적 전역 변수 사용은 자제하는 편이 좋고, 사용할 수밖에 없는 상황이라면 충돌하지 않게 유일한 이름으로 변수를 명명하자.[03] 그리고 HTML 콘텐츠를 비롯한 다른 코드와 충돌하지 않도록 항상 확실히 점검해야 한다.

03 역자주_ 간단하게는 변수명 뒤에 8자리 정도의 랜덤 숫자를 붙일 수도 있고, http://jsfiddle.net/EFq7c/10와 같이 다른 사람이 작성한 생성기를 활용할 수도 있습니다.

부록. 4 네이티브 프로토타입

어떠한 일이 있어도 네이티브 프로토타입을 확장하지 말지어다. 자바스크립트 고수들이 강변하는, 가장 널리 알려진 모범 사례이자 오래된 지침이다.

`Array.prototype`의 메서드나 프로퍼티 중 본인이 찾는 것이 없어 추가 구현해야 할 듯싶은 경우가 생긴다면, (그리고 여러분이 추가하고자 고려 중인 메서드/프로퍼티가 유용할 뿐만 아니라 잘 설계되고 이름도 잘 지어졌다면) 명세에 이미 포함되어 있을 가능성이 매우 높다. 그럼에도 불구하고 여러분이 직접 확장을 한다면 충돌은 피할 수 없다.

실제로 필자가 그렇게 해서 고생했던 경험을 공유하고자 한다.

당시 다른 웹사이트에서 임베드하여 사용 가능한 위젯을 작성 중이었고 jQuery 기반이었다. (대부분의 프레임워크가 이런 함정 때문에 고생 꽤나 했을 것이다) 거의 대부분 웹사이트에서 문제없이 작동했지만, 유독 한 곳에서 완전히 깨져버리는 현상이 목격됐다.

일주일 넘게 분석/디버깅을 계속한 끝에 문제의 사이트 내부에 다음 레거시 파일이 깊숙이 박혀있는 것을 찾아낼 수 있었다.

```
// 넷스케이프 4는 Array.push 메서드가 없다.
Array.prototype.push = function(item) {
this[this.length] = item;
};
```

말도 안 되는 주석(대체 누가 요즘 넷스케이프 4를 쓴단 말인가!?)은 차치하고, 자체만 봐서는 분명 잘못된 코드는 아니다.

문제는, 이 코드의 `Array.prototype.push`가 넷스케이프 4 이후에 명세에 추가되었지만, 그렇게 추가한 부분이 이 코드와 호환되지 않고 충돌한다는 점이다. 표

준 push(. .) 메서드는 한꺼번에 여러 원소를 추가할 수 있지만, 예제에서 쓰인 꼼수는 두 번째 이후 원소들은 그냥 무시해버린다.

거의 모든 자바스크립트 프레임워크에 push(. .) 메서드로 여러 원소를 한 번에 추가하는 코드가 구현되어 있다. 내 경우엔 CSS 셀렉터 엔진 근처에서 코드가 뻗어 버렸는데, 그 밖에도 의심 가는 부분이 한두 군데가 아니었다.

그 옛날 push(. .) 꼼수를 작성해 넣은 개발자는 동물적인 감각으로 push라 명명했지만, 원소를 다수 추가하는 기능까지는 생각하지 못했던 것 같다. 물론 이 코드는 당시엔 의도한 대로 잘 작동했겠지만, 10년이 흘러 내가 우연히 발견하기까지 제거되지 않은 지뢰로 남아 있었던 것이다.

자, 다음 몇 가지 교훈을 마음 속에 꼭 새기자.

첫째, 여러분이 짠 코드가 특정 환경에서만 실행될 것이라는, 절대적 확신이 없는 한 네이티브를 확장하지 말자. 100% 확신한다고 자신 있게 말할 수 없다면, 어떤 식으로든 직접 확장한 네이티브는 위험하다. 여러분 스스로 리스크를 감수해야 한다.

둘째, (실수로 네이티브를 덮어쓸 지 모르니) 무차별적인 확장은 피하자. 다음 코드가 그런 예다.

```
if (!Array.prototype.push) {
// 넷스케이프 4는 Array.push 메서드가 없다.
Array.prototype.push = function(item) {
this[this.length] = item;
};
}
```

여기서 if 문은 Array.prototype.push가 코드 실행 환경에 없을 경우에 한해 push( ) 꼼수를 정의하는 가드 역할을 한다. 내 경우엔 이렇게만 해도 OK다. 하

지만 그렇다고 리스크가 완전 사라지는 건 아니다.

1. 웹사이트 코드가 (어떤 말도 안 되는 이유에서!) 한 번에 여러 원소를 추가하는 기능이 누락된 `push(..)`에 의존적이라면, 벌써 오래 전 표준 `push(..)` 메서드가 탄생한 이후로 제대로 작동한 적이 한 번도 없었을 것이다.

2. 사용 중인 다른 라이브러리가 `if` 가드 앞에서 그들만의 `push(..)` 꼼수를 사용했다면, 역시 호환되지 않는 방향으로 흘러갔을 테고, 이미 그때부터 웹사이트는 조용하게 작동하지 않았을 것이다.

솔직히 자바스크립트 개발자들이 관심을 갖는 문제는 아니지만, 여러분에게 한 가지 재미있는 질문을 던져 보련다. "여러분이 작성한 코드 말고 다른 코드 역시 함께 실행되어야 하는 환경이라면, 전적으로 내장 네이티브 기능에만 의존해야 할까?"

엄밀하게는 "아니오"가 정답이지만, 사실 그럴 가능성 또한 대단히 희박하다. 여러분이 작성한 코드가 의존하는 모든 (건드릴 수 없는untouchable) 전용private 내장 기능은 재정의할 수 없기 때문이다.

그럼, 여러분의 예상대로 내장 기능이 작동하는지 여부를 피처feature/컴플라이언스compliance 테스트를 해야 할까? 그렇게 했는데 만약 테스트 결과가 실패하면, 여러분이 작성한 코드는 실행을 거부해야 할까?

```
// Array.prototype.push를 믿지 마라.
(function(){
if (Array.prototype.push) {
var a = [];
a.push(1,2);
if (a[0] === 1 && a[1] === 2) {
// 테스트가 성공했으니 안심하고 사용하자!
return;
}
}

throw Error(
```

```
"Array#push()는 없거나 깨졌어!"
);
})();
```

이론적으론 그럴 듯 하지만, 그 많은 내장 메서드를 일일이 테스트한다는 발상 자체가 정말 비현실적이다.

그럼 어쩌란 말인가? 일단 신뢰하되 모든 것을 검증(피처/컴플라이언스 테스트)해야 하나? 아니면 존재 자체를 컴플라이언스 기준을 충족했다고 보고, (다른 원인에 의해) 깨진 코드가 수면 위로 떠오르게 놔두어야 맞을까?

명쾌한 답은 없다. 하지만 유일하게 확신을 갖고 말할 수 있는 건, 네이티브 프로토타입의 확장을 시도하는 그 이후부터 가시밭길이 펼쳐질 거란 점이다.

여러분이 네이티브 프로토타입을 확장하지 않고, 여러분의 애플리케이션에서 다른 사람들도 그렇게 하지만 않으면, 안전하다. 그렇지 않다면, 깨질지 모를 코드를 예상하여 여러분은 최소한의 회의적/비관적인 생각을 가져야 한다.

알려진 모든 환경에서 실행되는 코드에 대해 철저히 단위unit/회귀regression 테스트를 마친 상태라면 이 절에서 논한 이슈들을 조기에 감지할 수 있겠지만, 그렇다고 실제로 충돌이 나지 않게 지켜주는 것은 아니다.

부록. 4.1 심/폴리필

구 버전(표준 명세를 준수하지 않는) 환경은 바뀔 가능성이 거의 없기 때문에 네이티브를 확장시켜도 안전한, 유일한 장소로 꼽힌다(새 명세의 기능을 탑재한 새 브라우저는 구 버전 기능을 수정amend하는 게 아니라 교체replace한다).

만약 여러분이 신통 방통한 예지력을 가진 덕분에 미래에 어떤 표준(이를테면 `Array.prototype.foobar`)이 등장할지 미리 다 알고 있다면, 여러분만의 호환 버전을 만들어 사용해도 안전할 것이다.

```
if (!Array.prototype.foobar) {
//정말 어리석군!
Array.prototype.foobar = function() {
this.push( "foo", "bar" );
};
}
```

이미 `Array.prototype.foobar` 명세가 정해졌고 그 기능이 예제와 같은 로직이라면, 안심하고 이같이 정의해서 사용할 수 있다. 이런 코드를 보통 "폴리필 polyfill"(또는 "심shim")이라고 한다.

최신 명세가 미처 반영되지 못한 옛날 브라우저 환경을 "패치patch"할 용도로 현재 코드베이스에 포함시킬 수 있으니 아주 유용하다. 폴리필은, 지원할 모든 환경별로 예측 가능한 코드를 생성할, 아주 훌륭한 수단이다.

ES5-심은 ES5 명세 기반의 프로젝트를 위한 심/폴리필의 종합 세트다. 마찬가지로 ES6-심은 ES6 이후로 추가된 새로운 API의 심을 제공한다. 다만 API는 폴리필/심으로 가능하지만, 새로운 구문은 이런 식으로는 안 된다. 구문 간극까지 메우려면 트라세Traceur[04] 같은 ES6-to-ES5 교환 컴파일러transpiler를 써야 한다.

어떤 표준이 출시가 임박했고 이미 명칭과 작동 방식에 대해 구체적인 논의가 진행되어 많은 사람들이 수긍한 상태에서, 앞으로의 표준에 맞추기 위해 사전 제작된 폴리필을 "프롤리필prollyfill(probably fill)"이라고 한다.

다만, 새로운 일부 표준 기능은 (완전히) 폴리필/프롤리필로 대체할 수 없다는 걸 조심해야 한다.

많이 나오는 기능들은 부분적인 폴리필만으로도 (적용 불가능한 부분은 문서화하는 식으로) 충분한지 아니면 100% 명세의 내용을 반영한 폴리필이 아니면 사용을 하지 말아야 할지? 이 문제는 아직도 커뮤니티에서 논란거리다.

04 https://github.com/google/traceur-compiler

아직 반영되지 않은 부분은 애당초 사용할 의도 또한 없었던 것으로 보고, 적어도 자주 쓰는 부분 폴리필(예: `Object.create(..)`)은 사용해도 괜찮다고 보는 개발자들이 많다.

일부 개발자들은 폴리필/심을 감싼 `if` 가드에 어떤 형태로든 적합성 여부를 테스트한 뒤 만약 기존 메서드 자체가 존재하지 않거나 적합성 테스트 실패 시에만 기존 메서드를 교체해야 한다고 이야기한다. 이러한 추가적인 적합성 테스트는 (컴플라이언스 테스트를 마친) "심"과 (존재한다는 사실이 확인된) "폴리필"을 각각 구분하는 용도로 사용한다.

여기서 절대적으로 옳은 정답이란 없다는 사실을 기억하기 바란다. 네이티브 확장은, 설령 구 버전 환경에서 "안전하게" 이루어진다 해도 100% 안전을 장담할 수는 없다. 다른 코드가 함께 실행되는 환경에서 (아마도 확장된) 네이티브에 전적으로 의존하는 것도 마찬가지다.

어떤 결정을 내리든지 조심조심, 방어적인 코드를 작성하고, 위험 요소에 대해 가급적 문서를 잘 작성하여 여러분의 의도를 분명히 밝히기 바란다.

부록. 5 〈script〉들

브라우저로 접속하는 대다수의 웹사이트나 애플리케이션은 다수의 자바스크립트 코드 파일로 구성되고, 이 파일들은 대개 `<script src=..></script>` 태그로 하나씩 페이지에 읽어 들인다. 또 인라인 형태로 `<script> .. </script>` 사이에 코드를 넣기도 한다.

그런데 이렇게 따로따로 떨어진 파일/코드 조각들은 개별적인 프로그램일까 아니면 집합적인 하나의 자바스크립트 프로그램일까?

실상은 (놀랍게도) 대부분의 경우(모든 경우 다는 아니다) 저마다 각자의 자바스크립트

프로그램으로 작동한다.

프로그램들이 서로 공유하는 건 단일 전역 객체(브라우저는 window)가 유일하다. 여러 파일들이 공유된 명칭공간에 자신의 코드를 덧붙여서 상호 작용을 할 수 있는 것도 이 때문이다.

그러므로 전역 함수 foo ( )가 정의된 첫 번째 스크립트 실행 후, 두 번째 스크립트에서는 마치 자신의 함수인 양 foo ( )에 접근/호출이 가능하다.

그러나 전역 변수 스코프는 파일을 넘나들면서 호이스팅(『You Don't Know JS: 스코프와 클로저』[05] 참고)하지는 않기 때문에 다음 코드는 <script> .. </script> 식의 인라인이든 <script src=..></script>로 외부에서 읽어 들이든 상관없이 (foo ( ) 선언부가 선언되지 않았으므로) 작동하지 않는다.

```
<script>foo();</script>
<script>
function foo() { .. }
</script>
```

반면 다음 두 예제는 문제없이 작동한다.

```
<script>
foo();
function foo() { .. }
</script>

<script>
function foo() { .. }
</script>

<script>foo();</script>
```

05 http://goo.gl/YCqO7K

또한, 한 스크립트에서 에러가 나면(인라인/외부 로딩 둘 다), 독립적인 개발 자바스크립트로 해당 프로그램만 실패 후 중단되며, 다른 후속 스크립트는 아무런 영향을 받지 않고 (여전히 전역 객체를 공유한 상태로) 실행된다.

스크립트 요소를 코드에서 동적으로 생성한 후 페이지 DOM에 주입하는 기법도 있다. 이렇게 포함된 코드는 떨어져 있던 파일에서 읽어들인 것과 동일하다.

```
var greeting = "Hello world";

var el = document.createElement( "script" );

el.text = "function foo(){ alert( greeting );\
} setTimeout( foo, 1000 );";
document.body.appendChild( el );
```

여기서 el.text에 코드를 할당하는 대신 el.src에 자바스크립트 파일의 URL을 가리키게 해도 마치 외부에서 동적으로 <script src=..></script> 요소를 읽어들인 것처럼 작동한다.

인라인 코드 블록 안에 코드가 있는 경우와 똑같은 코드가 외부 파일에 위치한 경우는 서로 한 가지 차이점이 있다. 전자의 경우, 문자 시퀀스 </script>는 (사용된 위치에 상관없이) 코드 블록의 끝으로 해석될 수 있으므로 함께 나오면 안 된다. 그래서 다음과 같은 코드를 조심해야 한다.

```
<script>
var code = "<script>alert( 'Hello world' )</script>";
</script>
```

그냥 봐서는 아무렇지 않을 것 같지만, 문자열 리터럴의 일부인 </script>은 스크립트 블록을 비정상적으로 종료시켜 에러가 발생한다. 그래서 보통 다음 꼼수로 해결한다.

```
"</sc" + "ript>";
```

또한, 외부 파일에 포함된 코드는 기본 서비스(또는 디폴트) 캐릭터 셋(UTF-8, ISO-8859-8 등)으로 엔진이 해석하지만, 똑같은 코드라도 인라인 스크립트로 사용하면 해당 HTML 페이지의 캐릭터 셋(또는 그 페이지의 디폴트 캐릭터 셋)으로 해석한다는 점을 유의해야 한다.

charset 속성은 인라인 스크립트 요소에는 적용되지 않는다.

인라인 코드 주변에 HTML 스타일이나 X(HT)ML 스타일의 주석을 쓰는 코딩 습관도 이제는 권장하지 않는다.

```
<script>
<!--
alert( "Hello" );
//-->
</script>
<script>
<!--//--><![CDATA[//><!--
alert( "world" );
//--><!]]>
</script>
```

아직도 이런 불필요한 습관을 갖고 있다면 당장 그만두자!

(HTML 스타일 주석인) <!--와 -->는 실제로는 자바스크립트에서 유효한 한 줄짜리 주석 구분자(예: `var x = 2; <!-- 유효한 주석 -->`)다. 이제는 낡은 수법이니 사용하지 말자.

부록. 6 예약어

ES5 §7.6.1에는 개별 변수명으로 사용해선 안 되는 "예약어reserved words" 목록이 나열되어 있다. 구체적으로는 "키워드keywords", "앞으로 예약된 단어future reserved words", `null` 리터럴, `true/false` 불리언 리터럴까지 4개의 카테고리가 있다.

`function`, `switch` 같은 키워드는 그냥 봐도 확실하다. 앞으로 예약된 단어에는 `enum` 등이 있지만, 이제는 이것들도 상당수가(`class`, `extends` 등) ES6 이후 실제로 사용된다. `Interface`처럼 엄격 모드 전용 예약어도 있다.

스택오버플로StackOverflow의 "art4theSould"란 ID를 가진 사용자가 예약어를 응용한 영시 한 편을 읊었는데, 재미있게 감상하기 바란다.[06]

Let this long package float,

Goto private class if short.

While protected with debugger case,

Continue volatile interface.

Instanceof super synchronized throw,

Extends final export throws.

Try import double enum?

- False, boolean, abstract function,

06 역자주_ 우리말로 옮기면 아무런 의미가 없으므로 영시 그대로 둡니다. 잘 읽어보시면 '정말 기발한 생각을 하는 사람들이 외국엔 많구나' 하는 생각이 들 겁니다.

Implements typeof transient break!

Void static, default do,

Switch int native new.

Else, delete null public var

In return for const, true, char

…Finally catch byte.

byte, long 등은 ES3에서는 예약어지만 ES5 이후부터는 아니다.

ES5 이전 시절엔 예약어는 프로퍼티명이나 객체 리터럴의 키 이름으로 절대 사용할 수 없었지만, 이제 더 이상 그런 제약은 없다.

즉, 다음 코드는 허용되지 않지만,

```
var import = "42";
```

다음 코드라면 OK다.

```
var obj = { import: "42" };
console.log( obj.import );
```

아직도 과거 구버전 브라우저(주로 옛날 IE)는 이러한 규칙들이 한결같이 지켜지지 않아 객체 프로퍼티명으로 예약어를 쓰면 문제가 될 수 있으니 조심하자. 모든 지원 대상 브라우저별로 꼼꼼히 테스트하기 바란다.

부록. 7 구현 한계

함수 파라미터 개수나 문자열 리터럴 길이 등의 제약은 명세에 따로 정의된 바 없지만, 엔진마다 상세한 내부 구현에 따라 실제로는 한계가 있다.

예를 들어,

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

var nums = [];
for (var i=1; i < 100000; i++) {
nums.push(i);
}

addAll( 2, 4, 6 ); // 12
addAll.apply( null, nums ); // 4999500000이 되어야 한다.
```

정확히 실행 결과가 4999500000으로 나와야 할 텐데, (사파리 6.x 같은) 일부 자바스크립트 엔진에서는 "RangeError: Maximum call stack size exceeded." 에러가 난다.

다음은 이런 유형의 알려진 한계들이다.

- (문자열 값이 아닌) 문자열 리터럴의 최대 문자 개수
- 함수 호출 시 인자로 보낼 수 있는 데이터 사이즈(바이트) (스택 크기)
- 함수 선언 시 파라미터 개수
- 최적화되지 않은 (재귀) 호출 스택의 최대 깊이: 한 함수가 다른 함수를 호출하는, 함수 호출 사슬의 최대 허용 길이
- 자바스크립트 프로그램이 연속적으로 브라우저를 블로킹한 채 실행 가능한 시간(초)
- 변수명 최대 길이

이러한 한계를 고려해야 할 상황은 결코 흔하지 않지만, 한계가 존재한다는 사실만은 꼭 인지해야 하며, 엔진마다 구체적인 한계치가 조금씩 다르다는 것도 알아두기 바란다.

부록. 8 정리하기

자바스크립트 언어가 단일 표준 명세를 따르며 대부분의 현대 브라우저/엔진이 예측 가능한 방향으로 이 명세를 구현했다는 사실을 모르는 사람은 없다. 정말 다행스러운 일이다!

하지만 혼자 동떨어져 실행되는 자바스크립트 코드는 매우 드물다. 서드파티 제공 라이브러리 코드가 한데 뒤섞인 환경에서 실행되기도 하고, 브라우저 아닌 다른 엔진/환경에서 실행될 때도 있다.

이런 이슈들에 대해 주의를 기울인다면 좀 더 탄탄하고 신뢰성 있는 코드를 작성할 수 있을 것이다.