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

## 추천의 글

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 초창기 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.

chapter 2

# 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 `+0`이 된다.

## 2.4 특수 값

타입별로 자바스크립트 개발자들이 조심해서 사용해야 할 특수한 값들이 있다.

### 2.4.1 값 아닌 값

`Undefined` 타입의 값은 `undefined`밖에 없다. `null` 타입도 값은 `null`뿐이다. 그래서 이 둘은 타입과 값이 항상 같다.

`undefined`와 `null`은 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.