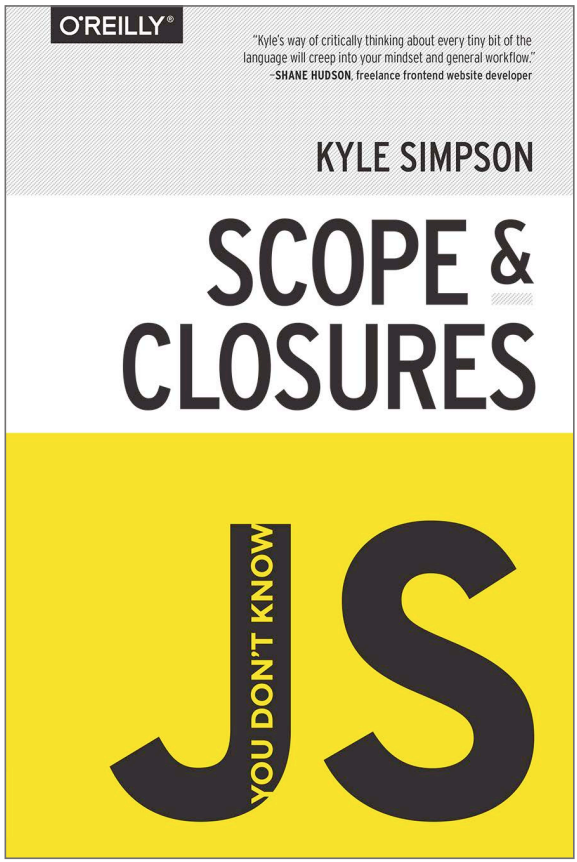

Hanbit eBook
Realtime 81
You Don't Know JS
스코프와 클로저
SCOPE & CLOSURES
카일 심슨 지음 / 최병현 옮김
O'REILLY
한빛미디어
Hanbit Media, Inc.



이 도서는 O'REILLY의
You don't know JS: Scope & Closures의
번역서입니다.

You Don't Know JS **스코프와 클로저**

초판발행 2014년 08월 29일

지은이 카일 심슨 / **옮긴이** 최병현 / **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 978-89-6848-633-3 15000 / **정가** 12,000원

책임편집 배용석 / **기획** 김창수 / **편집** 김창수, 정지연
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 최송실
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 김옥현

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

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

지은이_ **카일 심슨**Kyle Simpson

텍사스 오스틴 출신의 오픈 웹 전도사인 카일 심슨은 자바스크립트, HTML5, 실시간 P2P 통신과 웹 성능에 열정적인 관심을 갖고 있다. 열정이 없었다면, 이런 작업에 이미 진력이 났을 것이다. 그는 저술가이자 워크숍 강사와 기술 연사이며, 오픈 소스 커뮤니티에서도 열심히 활동하고 있다.

역자 소개

역자_ **최병현**

서강대학교에서 컴퓨터공학을 전공했다. 현재 인문과 IT 분야 번역 프리랜서로 일하고 있다.

저자 서문

독자 여러분도 알겠지만, 이 책 시리즈 명의 'JS'는 자바스크립트를 욕하는 단어가 아니다. 물론 자바스크립트에 별나게 짜증 나는 점이 있다는 것은 모두 동의하겠지만 말이다!

웹이 사용된 이래, 자바스크립트는 사용자들이 상호소통하며 콘텐츠를 이용할 수 있게 해주는 기반 기술이었다. 자바스크립트가 비록 마우스가 움직인 궤적을 반짝이게 하거나 짜증 나는 팝업 경고창을 띄우는 용도에서 시작했지만, 개발 후 거의 20년이 지난 지금은 비할 바 없이 성장했다. 세계에서 가장 널리 이용되는 소프트웨어 플랫폼이 웹이 된 상황에서 그 누구도 자바스크립트의 중요성을 의심하지 않는다.

그러나 하나의 언어로서 자바스크립트는 끊임없이 많은 비판의 대상이었다. 이는 부분적으로는 과거의 유산 때문이지만, 대부분은 자바스크립트의 '디자인 철학' 때문이다. 심지어 브랜든 아이츠Brendan Eich가 예전에 말했듯, 이름마저도 형인 '자바'의 '멍청한 동생' 같은 느낌을 준다. 물론 이름이야 그저 정치와 마케팅의 우연이었을 뿐이다. 이 두 언어는 여러 중요한 부분에서 엄청난 차이가 있다. '자바스크립트'와 '자바'는 '노트'와 '노트북'만큼이나 별 관련이 없다.

위풍당당한 C 스타일의 절차적 방식과 절묘하고 번하지 않은 Scheme/Lisp 스타일의 함수적 방식을 포함해 자바스크립트는 개념과 문법 표현양식을 여러 언어에서 빌려왔기 때문에 많은 방면의 개발자는 물론 심지어 프로그래밍 경험이 없는 이들도 자바스크립트를 굉장히 쉽게 이용할 수 있다. 'Hello World'를 출력해 보면 알겠지만, 자바스크립트는 매우 쉽고 금세 익숙해질 수 있다.

자바스크립트는 아마 작업하기 가장 쉬운 언어 중 하나겠지만, 얄궂은 데가 있어서

다른 언어보다 완전히 숙달한 사람은 훨씬 적다. C나 C++ 같은 언어로 완전한 프로그램을 짜려면 상당한 수준의 깊은 지식이 필요하지만, 자바스크립트는 언어에 대한 표면적인 지식만 겨우 알더라도 완전한 프로그램을 만들어낼 수 있다(사실 많은 자바스크립트 프로그램이 이렇게 작성됐다).

자바스크립트의 고급 개념은 콜백 함수를 전달하는 것과 같이 겉으로 보기에는 단순하게 보인다. 그래서 자바스크립트 개발자는 그저 보이는 대로 기능을 사용하고 기능이 수면 아래에서 어떻게 작동하는지는 별로 고민하지 않는다.

자바스크립트는 간단하고 쉽게 사용할 수 있어서 많은 이에게 매력적인 언어이지만, 세심히 탐구하지 않으면 경험 많은 자바스크립트 개발자도 제대로 이해하지 못할 정도로 복잡하고 미묘한 언어 메커니즘을 가지고 있다.

바로 이 부분이 자바스크립트의 모순이자 아킬레스건이고, 우리가 다룰 도전 과제다. 자바스크립트는 이해하지 않고도 사용할 수 있으므로 굳이 이해하려고 노력하지 않는 경우가 많다.

목적

자바스크립트를 다루면서 예상치 못한 결과가 나올 때마다 블랙리스트에 넣고 이용하는걸 꺼린다면, 결국에는 자바스크립트의 풍부한 기능 중 오직 일부만을 쓰게 될 것이다. 보통 이렇게 쓰이는 부분을 자바스크립트의 '좋은 부분'이라고 부른다. 그렇지만 필자는 감히 말하겠다. 그것은 그저 '쉬운 부분', '안전한 부분', 또는 '불완전한 부분'일 뿐이다.

『You Don't Know JS』 시리즈는 다른 종류의 도전과제를 제공한다. 자바스크립트의 모든 것을 깊게 배우고 이해해보자. '힘든 부분'까지도, 아니 특히 그 부분을 말이다.

이 책에서 필자는 자바스크립트 개발자가 간신히 일을 해나갈 수 있을 정도만 배우려는 경향, 자바스크립트의 작동 방식과 원리를 전혀 알려고 하지 않는 그 경향을 분명하게 비판할 것이다. 거친 길은 돌아가라지만, 이 책을 쓰는 데는 이런 말을 따르지 않을 것이다.

왜 그런지 알지도 못하면서 그저 작동한다고 만족할 수는 없다. 자바스크립트를 제대로 배우려면 여러분도 그래야 한다. 부탁하건대, 이 울퉁불퉁하고 "남들이 가지 않았던 길"을 걸어서 자바스크립트의 모든 것을 배워보시라. 그 어떤 테크닉, 프레임워크, 전문용어도 이 지식의 범위를 넘어서지 않는다.

『You Don't Know JS』 시리즈는 자바스크립트의 핵심 요소 중 가장 착각하기 쉽고 이해하기 어려운 부분을 깊고 철저하게 파고든다. 다 알고 있을 것이란 자신감은 버리고 책을 읽는 게 좋을 것이다. 이론적인 부분만이 아니라 실용적인 '알아야 할 필요가 있는' 부분에 대해서도 말이다.

많은 개발자가 자바스크립트를 불확실하게 알고 있는 이들에게 배운다. 그렇게 배운 자바스크립트는 진짜 자바스크립트의 껍데기에 불과하다. 『You Don't Know JS』 시리즈를 파고든다면 진정한 자바스크립트를 배우게 될 것이다. 계속 읽어보길 바란다. 자바스크립트가 여러분을 기다리고 있으니 말이다.

리뷰

자바스크립트는 멋진 언어다. 배우기 쉽지만, 완전하게 또는 충분히 배우기는 아주 어렵다. 보통 개발자가 작업하다가 헷갈리면 자신의 이해가 부족한 탓을 하기보다는 언어 탓을 한다. 이 시리즈는 바로 이 점을 바꾸려는 것이다. 이 시리즈를 읽고 나면 자바스크립트의 진면목이 이해되므로 더는 그런 혼란이 없을 테니까.

이 책의 많은 예제는 ECMA.Script version 6(ES6) 같은 최신의 자바 엔진 환경에서 실행될 것을 가정한다. 그래서 몇몇 코드는 이전 환경(ES6 이전)에서 제대로 작동하지 않을 수 있다.

추천사

어릴 적 나는 오래된 휴대전화, hi-fi 스테레오 같은 잡동사니를 잡히는 대로 분해하고 조립하면서 놀았다. 이런 기기를 사용하기에 너무 어렸지만, 망가진 기기를 볼 때마다 기기가 어떻게 작동했는지 이해하려고 노력했다.

언젠가 오래된 라디오의 회로판을 보았던 기억이 난다. 그 회로판에는 구리선이 감긴 이상하게 긴 튜브가 있었다. 튜브의 역할을 알아내지는 못했지만, 바로 연구에 들어갔다. 튜브는 무슨 역할을 하지? 튜브는 왜 라디오 안에 들어 있지? 튜브는 회로판의 다른 부분과 다르게 생겼는데, 왜 그럴까? 왜 구리선이 튜브를 감고 있는 걸까? 구리선을 벗겨내면 어떤 일이 생길까? 지금은 그 튜브가 루프 안테나고, 페라이트 막대에 구리선을 감아서 만들며 트랜지스터 라디오에 많이 사용된다는 것을 안다.

당신은 왜 그럴까라는 질문의 답을 찾기 위해 골몰해본 적이 있는가? 대다수 아이는 이런 호기심이 있다. 사실 내가 아이들을 좋아하는 가장 큰 이유도 아이들의 배우고자 하는 욕구일 것이다.

다행인지 불행인지, 나는 이제 전문가라고 불리며 무언가를 만드는 일을 한다. 어릴 적에는 분해한 기기들을 언젠가 내가 만들 수 있길 바랐다. 물론, 지금 내가 만드는 것들은 페라이트 막대가 아니라 자바스크립트로 만들지만 말이다. 뭐, 결국은 비슷한 일이긴 하다! 예전에는 무언가를 만들고 싶다고 생각했지만, 지금은 무언가를 분석하고 이해하는 것을 더 좋아한다. 그래서 문제를 해결하거나 버그를 고치는 가장 좋은 방법을 찾아내는 데 골몰했지만, 정작 내 도구에 대해서는 신경 쓰지 못했다.

바로 이점 때문에 『You Don't Know JS』 시리즈를 정말 좋아한다. 맞는 말이다. 나는 자바스크립트를 모른다. 자바스크립트를 매일 매일 수십 년간 사용했지만, 정말 자바스크립트를 이해하냐고 묻는다면 아니라고밖에 대답할 수 없다. 물론, 자바

스크립트의 많은 부분을 이해하고 많은 관련 설명서와 인터넷 문서를 읽는다. 그럼에도 모든 것을 이해하길 바랐던 어릴 적 바람만큼 완전하게 자바스크립트를 이해하지는 못한다.

스코프와 클로저는 『You Don't Know JS』 시리즈를 시작하기에 아주 좋은 주제다. 특히 나 같은 사람에게 아주 유용한 주제다(당신에게도 그렇길 빈다). 자바스크립트를 한 번도 사용하지 않은 이들에게 가르칠 만한 내용은 아니지만, 이 주제를 통해 자바스크립트 사용자들이 이 언어가 내부적으로 어떻게 작동하는지를 이해할 수 있을 것이다. 이 책은 참 적절한 시기에 나왔다. 이제 ES6는 정착되고 있고, 여러 웹 브라우저에서 잘 구현되고 있다. 아직 let이나 const 같은 새로운 기능을 익힐 시간을 내지 못했던 이들에게 이 책은 좋은 입문서가 될 것이다.

이 책이 도움되길 바라고, 무엇보다 여러분이 자바스크립트의 세세한 부분이 어떻게 작동하는지에 대한 저자 카일의 중요한 지적들을 전반적 사고와 작업 과정에 녹여 내기를 바란다. 그저 안테나를 사용할 것이 아니라, 안테나가 어떻게 작동하는지 이해하길 바란다.

셰인 허드슨Shane Hudson

www.shanehudson.net

역자 서문

이 책의 주요 독자 중에는 전문 프로그래머로서 훈련받은 이들이 많을 것이고, 아마도 C나 C++ 같은 언어를 능숙하게 사용할 줄 아는 이들도 적지 않을 것이다.

C나 C++를 처음 배웠을 때를 떠올려 보자. 다짜고짜 "Hello World" 출력을 배우는 경우도 있겠지만, 기초를 튼튼히 다지면서 배웠다면 함수의 선언, 형태, 입출력, 지역변수와 전역변수의 차이, 컴파일 과정 등도 익혔을 것이다. 이것은 언어의 규범을 익히는 과정이고, 해당 언어로 쓰인 코드 한 줄 한 줄이 어떤 역할을 하는지 이해해 나가는 과정이다. 그리고 이런 언어의 규범을 익혀야 그 언어를 한계까지 최대한 활용할 수 있다.

재귀호출은 고급 알고리즘을 구현하는 데 필수다. 그런데 재귀호출과 그 과정에서 생성되는 변수들의 스택이 어떤 식으로 작동하는지 이해하지 못하면 사용할 수 없다.

C나 C++ 컴파일러가 상당히 엄격하게 굴기는 하지만, 일반적으로 C나 C++ 사용자들은 이런 규범을 익히는 과정을 거치는 경우가 많다. 하지만 자바스크립트는 어떤가? 기본적으로 상당히 쉽게 배울 수 있는 언어라서 대학에서 배우는 경우도 드물고 배우더라도 언어 자체의 구조와 작동 방식을 배우기보다는 자바스크립트를 가지고 브라우저에서 무언가 동작시키는 실습 수업이 대부분이다. 직장에서 배우는 경우는 말할 것도 없다. 늘 촉박하게 결과를 내야 하는 기업에서 한가하게 언어의 구조를 고찰하고 있을 수는 없다(더군다나 배우기 어렵지도 않으니 말이다).

자바스크립트 엔진은 상당히 너그러워서 '적당히' 코딩해도 오류를 뿜어내는 일 없이 '적당히' 처리해준다. 그래서 자바스크립트를 사용하며 "뭘 했는지는 모르겠지만, 어쨌든 원하던 결과는 나왔어!"라고 약간은 찜찜해하며 넘어간 적이 있을 것이다. 아마도 이런 특성 때문에 자바스크립트를 학교에서도 실습에 사용하고 기업에

서도 아무나 시켜서 빨리 배우라고 할 수 있는 것이리라(그리고 많은 이들이 자바스크립트를 욕하는 이유기도 하다).

그러나 저자 카일은 『You Don't Know JS』 시리즈에서 자바스크립트를 사용할 때도 C나 C++ 같은 아니 어쩌면 더 까다로운 규범을 익혀야 한다고 말한다. 더 깔끔하고 더 안전한 코드를 작성하고 자바스크립트의 능력을 최대한 활용하기 위해 알아야 할 것이 있다는 것이다. 자바스크립트를 수년째 사용하는 프로그래머로서 또 이 책의 번역자로서, 언어의 한계를 최대한 활용하여 효율적인 코딩을 하길 좋아하는 고급 프로그래머라면 이 책을 즐겁게 읽을 것이다.

예제 파일

이 책에서 사용하는 예제 코드는 http://bit.ly/1c8HEWF에서 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 찾는 것도 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 Only - 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 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 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 위하여 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 힘든 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

1 | 스코프란 무엇인가

프로그래밍 언어의 기본 패러다임 중 하나는 변수에 값을 저장하고, 저장된 값을 가져다 쓰고 수정하는 기능이다. 이 기능은 프로그램에서 상태를 나타낼 수 있게 해준다. 이와 같은 개념이 없다면 프로그램은 상당히 제한적이고 지극히 심심한 작업만 할 수 있을 것이다. 그러나 변수를 프로그램에 추가하면 다음과 같은 재미있는 질문이 생긴다.

- 변수는 어디에 살아있는가? 다른 말로 하면, 변수는 어디에 저장되는가?
- 필요할 때 프로그램은 어떻게 변수를 찾는가?

이 질문을 통해 알 수 있는 것은 특정 장소에 변수를 저장하고 나중에 그 변수를 찾는 데는 잘 정의된 규칙이 필요하다는 점이다. 바로 이런 규칙을 '스코프Scope'라 한다.

그렇다면 스코프 규칙은 어디서 어떻게 정의될까?

1.1 컴파일러 이론

여러 언어를 다루어 봤다면 자명할 수도 있겠고 아니라면 놀라울 수도 있겠지만, 자바스크립트는 일반적으로 '동적' 또는 '인터프리터' 언어로 분류하지만, 사실은 '컴파일러 언어'다. 물론 자바스크립트가 전통적인 많은 컴파일러 언어처럼 코드를 미리 컴파일하거나 컴파일한 결과를 분산 시스템에서 이용할 수 있는 것은 아니다. 하지만 자바스크립트 엔진은 전통적인 컴파일러 언어에서 컴파일러가 하는 일의 상당 부분을 우리가 아는 것보다 세련된 방식으로 처리한다.

전통적인 컴파일러 언어의 처리 과정에서는 프로그램을 이루는 소스 코드가 실행되기 전에 보통 3단계를 거치는데, 이를 '컴파일레이션compilation'이라고 한다.

토크나이징Tokenizing/렉싱Lexing

문자열을 나누어 '토큰token'이라 불리는 (해당 언어에) 의미 있는 조각으로 만드는 과정이다. 예를 들어, "var a =2;"라는 프로그램을 보자. 이 프로그램은 다음의 토큰으로 나눌 수 있다.

- var,
- a
- =
- 2
- ;
- 빈칸은 하나의 토큰으로 남을 수도 있고 아닐 수도 있다. 이는 빈칸이 의미가 있느냐 없느냐에 달렸다.

> **NOTE_**
>
> 토크나이징과 렉싱은 미묘하고 학술적인 차이가 있는데, 토큰을 인식할 때 무상태 방식으로 하는지 상태유지 방식으로 하는지에 따라 구분한다. 쉽게 말해, '토크나이저tokenizer'가 상태유지 파싱 규칙을 적용해 a가 별개의 토큰인지 다른 토큰의 일부인지를 파악한다면 렉싱이다.

파싱Parsing

토큰 배열을 프로그램의 문법 구조를 반영하여 중첩 원소를 갖는 트리 형태로 바꾸는 과정이다. 파싱의 결과로 만들어진 트리를 AST(추상구문트리abstract syntax tree)라 부른다.

"var a =2;"의 트리는 먼저 변수선언VariableDeclaration이라 부르는 최상위 노드에서 시작하고, 최상위 노드는 'a'의 값을 가지는 확인자Identifier와 대입수식AssignmentExpression

이라 부르는 자식 노드를 가진다. 대입수식 노드는 '2'라는 값을 가지는 숫자리터럴 NumericLiteral을 자식 노드로 가진다.

코드 생성Code-Generation

AST를 컴퓨터에서 실행 코드로 바꾸는 과정이다. 이 부분은 언어에 따라 또는 목표하는 플랫폼에 따라 크게 달라진다. 코드 생성에 대한 세부사항을 보며 끙끙대기보다는 일단 앞서 말한 "var a = 2;"를 나타내는 AST를 기계어 집합으로 바꾸어 실제로 'a'라는 변수를 생성(메모리를 확보하는 일 등)하고 값을 저장할 방법이 있다고 치자.

> **NOTE_**
>
> 엔진이 시스템 리소스를 실제 어떻게 관리하는지에 관한 세부사항은 살펴볼 범위를 넘어서므로 엔진이 필요한 변수를 생성하고 저장할 것이라고 가정하고 넘어가겠다.

자바스크립트 엔진은 이 세 가지 단계뿐만 아니라 많은 부분에서 다른 프로그래밍 언어의 컴파일러보다 훨씬 복잡하다. 예컨대, 자바스크립트 엔진는 파싱과 코드 생성 과정에서 불필요한 요소를 삭제하는 과정을 거쳐 실행 시 성능을 최적화한다.

여기서는 개괄적인 설명만 한다. 그러나 계속 읽다 보면 여기서 왜 대략적이나마 앞의 부분을 다루었는지 알게 될 것이다.

자바스크립트 엔진이 기존 컴파일러와 다른 한 가지 이유는 우선 다른 언어와 다르게 자바스크립트 컴파일레이션이 미리 수행되지 않아서 최적화할 시간이 많지 않기 때문이다.

자바스크립트 컴파일레이션은 보통 코드가 실행되기 겨우 수백만 분의 일초 전에 수행한다. 가능한 한 가장 빠른 성능을 내기 위해 자바스크립트 엔진은 이 책에서

다룰 범위를 가볍게 넘어서는 여러 종류의 트릭(레이지 컴파일lazy compile이나 핫 리컴파일hot recompile 같은 JITs)을 사용한다.

간단히 말하자면, 어떤 자바스크립트 조각이라도 실행되려면 먼저(보통 바로 직전에!) 컴파일되어야 한다는 것이다. 즉, 자바스크립트 컴파일러는 프로그램 "var a = 2;"를 받아 컴파일하여 바로 실행할 수 있게 한다.

1.2 스코프 이해하기

스코드를 좀 더 재미있고 쉽게 설명하기 위해서 대화 형식으로 스코프를 살펴보겠다. 먼저 등장인물을 살펴보자.

1.2.1 출연진

프로그램 "var a =2;"를 처리할 주역들을 만나보자, 그래야 나중에 들을 대화를 이해할 수 있다.

- 엔진: 컴파일레이션의 시작부터 끝까지 전 과정과 자바스크립트 프로그램 실행을 책임진다.
- 컴파일러: 엔진의 친구로, 파싱과 코드 생성의 모든 잡일을 도맡아 한다.
- 스코프: 엔진의 또 다른 친구로, 선언된 모든 확인자(변수) 검색 목록을 작성하고 유지한다. 또한, 엄격한 규칙을 강제하여 현재 실행 코드에서 확인자의 적용 방식을 정한다.

만약 자바스크립트가 어떻게 작동하는지 완전히 이해한다면 엔진(그리고 그 친구들)처럼 생각해보자. 그들이 던지는 질문을 던지고 그들과 똑같이 답해보라.

1.2.2. 앞과 뒤

프로그램 "var a = 2;"를 보면 하나의 구문으로 보인다. 그러나 우리의 새로운 친구 엔진은 그렇게 보지 않는다. 사실 엔진은 두 개의 서로 다른 구문으로 본다. 하나는 컴파일러가 컴파일레이션 과정에서 처리할 구문이고, 다른 하나는 실행 과정에서 엔진이 처리할 구문이다.

그럼 이제 엔진과 친구들이 프로그램 "var a =2;"에 어떻게 접근하는지 낱낱이 살펴보자.

이 프로그램에서 컴파일러가 할 첫 번째 일은 렉싱을 통해 구문을 토큰으로 쪼개는 것이다. 그 후 토큰을 파싱해 트리 구조로 만든다. 그러나 코드 생성 과정에 들어가면 컴파일러는 몇몇 독자의 추측과는 다르게 프로그램을 처리한다.

컴파일러가 다음의 의사 코드pseudo-code로 요약될 수 있는 코드를 생성한다고 생각할 수 있다.

변수를 위해 메모리를 할당하고 할당된 메모리를 a라 명명한 후 그 변수에 값 2를 넣는다.

안타깝지만, 이는 그리 정확한 설명이 아니다. 컴파일러는 다음 일을 진행한다.

① 컴파일러가 'var a'를 만나면 스코프에 변수 a가 특정한 스코프 컬렉션 안에 있는지 묻는다. 변수 a가 이미 있다면 컴파일러는 선언을 무시하고 지나가고, 그렇지 않으면 컴파일러는 새로운 변수 a를 스코프 컬렉션 내에 선언하라고 요청한다.

② 그 후 컴파일러는 'a = 2' 대입문을 처리하기 위해 나중에 엔진이 실행할 수 있는 코드를 생성한다. 엔진이 실행하는 코드는 먼저 스코프에 a라 부르는 변수가 현재 스코프 컬렉션 내에서 접근할 수 있는지 확인한다. 가능하다면 엔진은 변수 a를 사용하고, 아니라면 엔진은 다른 곳(중첩 스코프 부분을 보라)을 살핀다.

엔진이 마침내 변수를 찾으면 변수에 값 2를 넣고, 못 찾는다면 엔진은 손을 들고 에러가 발생했다고 소리칠 것이다!

요약하면, 별개의 두 가지 동작을 취하여 변수 대입문을 처리한다. 첫째, 컴파일러가 변수를 선언한다(현재 스코프에 미리 변수가 선언되지 않은 경우). 둘째, 엔진이 스코프에서 변수를 찾고, 변수가 있다면 값을 대입한다.

1.2.3 컴파일러체[01]

더 나아가기 전에 컴파일러 관련 용어를 약간 더 살펴보자.

2단계에서 컴파일러가 생성한 코드를 실행할 때 엔진은 변수 a가 선언된 적 있는지 스코프에서 검색한다. 이때 엔진이 어떤 종류의 검색을 하느냐에 따라 검색 결과가 달라진다. 앞의 경우에서 엔진은 변수 a를 찾기 위해 LHS 검색을 수행한다. 다른 종류의 검색은 RHS라 부른다. 여기서 'L'과 'R'이 무엇을 뜻하는지 예상할 수 있을 것이다. L과 R은 각각 '왼쪽 방향Left-hand Side'과 '오른쪽 방향Right-hand Side'을 뜻한다.

방향? 어떤 방향? 바로 대입 연산의 방향을 말한다.

다른 말로 하면 LHS 검색은 변수가 대입 연산자의 왼쪽에 있을 때 수행하고, RHS 검색은 변수가 대입 연산자의 오른쪽에 있을 때 수행한다.

좀 더 엄밀하게 살펴보자. RHS 검색은 단순히 특정 변수의 값을 찾는 것과 다를 바 없다. 반면, LHS 검색은 값을 넣어야 하므로 변수 컨테이너 자체를 찾는다. 따라서 정확히 말하면 RHS는 그 자체로는 '대입문의 오른쪽'이 아니다. 좀 더 정확히 말하면 RHS는 '왼편이 아닌 쪽'에 가깝다.

---

01 컴파일러체(Compiler Speak): 컴파일러의 속어로, 사투리 정도로 이해하면 된다.

좀 더 쉽게 말하면 RHS를 "Retrieve(가져오라) his/her(그의/그녀의) source(소스)"의 약자라고 보면 RHS는 "가서 값을 가져오라"라는 뜻으로 이해할 수 있다. 이제 좀 더 깊게 파보도록 하자. 다음 구문을 보자.

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

a에 대한 참조는 RHS 참조다. 구문에서 a에 아무 것도 대입하지 않기 때문이다. 대신 a의 값을 가져와 console.log(…)에 넘겨준다. 다른 예제를 보자.

```
a = 2;
```

a에 대한 참조는 LHS 참조다. 현재 a 값을 신경 쓸 필요 없이 '= 2' 대입 연산을 수행할 대상 변수를 찾기 때문이다.

NOTE_

LHS와 RHS가 '대입문의 왼쪽/오른쪽'을 뜻한다고 해서 반드시 문자 그대로 '대입 연산자(=)의 왼쪽과 오른쪽'을 지칭하는 것은 아니다. 대입 연산은 다른 여러 방식으로 일어날 수 있다. 따라서 개념적으로는 다음과 같이 생각하는 것이 더 낫다.

- 대입할 대상(LHS)과 대입한 값(RHS)

LHS와 RHS 참조를 모두 수행하는 다음 프로그램을 보자.

```
function foo(a) {
    console.log( a ); // 2
}
foo( 2 );
```

마지막 줄에서 foo(…) 함수를 호출하는 데 RHS 참조를 사용한다. 즉 "가서 foo의 값을 찾아 내게 가져와라"라는 뜻이다. 여기서 (…)는 실행된다는 뜻이므로 foo는 함수여야 한다.

이 부분에 미묘하지만 중요한 대입이 수행된다. 무엇을 가리키는지 알겠는가?

앞의 코드 속에 내재된 'a = 2'를 놓쳤을지도 모르겠다. 인수로 값 2를 함수 foo(…)에 넘겨줄 때 값 2를 매개변수 a에 대입하는 연산이 일어난다. 이 (내재된) 매개변수 a에 대한 대입 연산을 위해 LHS 검색이 수행된다.

변수 a에 대한 RHS 참조 역시 수행되는데, 그 결과값은 console.log(…) 함수에 넘겨진다. 또 console.log(…)가 실행되려면 참조가 필요하다. console 객체를 RHS 검색하여 log 메소드가 있는지 확인한다.

마지막으로 값 2를 RHS로 불러온 변수 a를 통해 log(…)에 넘겨주는 과정에서 LHS/RHS를 주고받는 작업에 대한 개념을 짚어 보자. 구현된 log(…)의 내부에는 매개변수가 있을 것이고, 첫 번째 매개변수(어쩌면 arg1라 부를 것)를 LHS 검색으로 찾아 2를 대입할 것이다.

NOTE_

어쩌면 함수선언문 "function foo(a) { …"를 "var foo"와 "foo = function(a) { …" 같은 일반적인 변수 선언 및 대입과 같다고 생각할지도 모르겠다. 그렇게 생각한다면 이 함수 선언이 LHS 검색을 사용할 것이라 단정할 수도 있겠다.

그러나 여기에는 변수 처리 과정과 다른 미묘하지만 중요한 차이점이 있다. 컴파일러는 앞의 선언문과 값 정의문을 코드 생성 과정에서 처리하여 엔진이 코드를 실행할 때는 foo에 함수값을 대입하는 과정이 필요 없다. 따라서 함수 선언을 앞에서 살펴본 변수의 경우와 같이 LHS 검색 및 대입 과정이라고 생각하는 것은 적절하지 않다.

1.2.4 엔진과 스코프의 대화

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

이 코드의 실행 과정을 대화라고 상상해보자. 그 대화는 이럴 것이다.

엔진 : 안녕, 스코프. foo에 대한 RHS 참조가 필요해. foo라고 들어본 적 있니?

스코프: 응, 들어봤어. 컴파일러가 좀 전에 선언하더라고. foo는 함수야. 이걸 보면 돼.

엔진 : 좋아, 고마워! 자, 이제 나는 foo를 실행해야겠어.

엔진 : 이봐, 스코프. a에 대한 LHS 참조도 구해야 하는데, 들어본 적 있어?

스코프: 물론이지. 컴파일러가 a를 foo의 매개변수로 좀 전에 선언했어. 이걸 봐.

엔진 : 항상 도와줘서 고마워, 스코프. 정말 고마워. 이제 2를 a에 대입할 시간이야.

엔진 : 스코프, 자꾸 귀찮게 해서 미안해. console에 대한 RHS 검색이 필요해. 해줄 수 있겠니?

스코프: 문제 없어, 엔진. 이게 내가 항상 하는 일이잖니. 자, console을 찾았어. 내장돼 있더라. 이거야.

엔진 : 완벽해. 이제 log(…)를 찾아볼까. 좋아 멋져, 이건 함수구나.

엔진 : 요~ 스코프! a의 RHS 참조 찾는 것 좀 도와줄 수 있을까? 나한테도 있긴 할 테지만, 확실히 해두고 싶어.

스코프: 옳은 말이야, 엔진. 변함 없이 엄밀하구나. 여기 있어.

엔진 : 멋져. 이제 a의 값을…. 값은 2구나. log(…)에 넘기자.

…

1.2.5 퀴즈

지금까지 배운 것을 확인해보자. 엔진의 역할을 맡아서 스코프와 대화해보자.

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

① 모든 LHS 검색을 찾아보라(모두 3개다!).

② 모든 RHS 검색을 찾아보라(모두 4개다!).

> **NOTE_**
>
> 퀴즈의 정답은 이번 장 복습하기에서 볼 수 있다.

1.3 중첩 스코프

스코프는 확인자 이름으로 변수를 찾기 위한 규칙의 집합이라고 앞서 말한 바 있다. 그러나 대개 고려해야 할 스코프는 여러 개다.

하나의 블록이나 함수는 다른 블록이나 함수 안에 중첩될 수 있으므로 스코프도 다른 스코프 안에 중첩될 수 있다. 따라서 대상 변수를 현재 스코프에서 발견하지 못하면 엔진은 다음 바깥의 스코프로 넘어가는 식으로 변수를 찾거나 글로벌 스코프라 부르는 가장 바깥 스코프에 도달할 때까지 계속한다.

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

b에 대한 RHS 참조는 함수 foo 안에서 처리할 수 없고, 함수를 포함하는 스코프(이 경우에는 글로벌 스코프)에서 처리한다.

엔진과 스코프의 대화를 다시 보자.

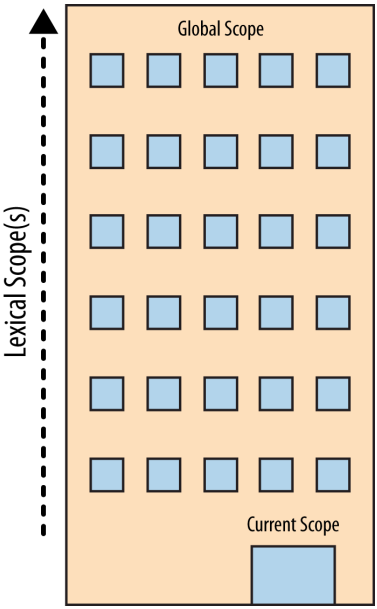
> **엔진** : 이봐 foo의 스코프, b에 대해 들어본 적 있나? b에 대한 RHS 참조가 필요한데 말야.
>
> **스코프**: 아니, 못 들어봤어. 딴 데 가봐.
>
> **엔진** : 이봐, foo의 바깥 스코프! 아, 니가 글로벌 스코프구나, 멋지군. 혹시 b에 대해 들어봤니? b에 대한 RHS 참조가 필요해.
>
> **스코프**: 응, 물론 들어봤지. 여기 있어.

중첩 스코프를 탐사할 때 사용하는 간단한 규칙은 다음과 같다.

- 엔진은 현재 스코프에서 변수를 찾기 시작하고, 찾지 못하면 한 단계씩 올라간다.
- 최상위 글로벌 스코프에 도달하면 변수를 찾았든 못 찾았든 검색을 멈춘다.

1.3.1 비유로 배워보기

중첩 스코프 검색 과정을 다음과 같은 큰 빌딩으로 상상해보자.



이 빌딩은 프로그램의 중첩 스코프 규칙 집합을 나타낸다. 어디에 있든 1층은 현재 실행 중인 스코프를 뜻한다. LHS/RHS를 참조하려면 현재 층을 둘러보고, 찾지 못하면 엘리베이터를 타고 다음 층으로 가서 찾고, 또 다음 층으로 이동하는 식이다. 최상위 층(글로벌 스코프)에 도달했을 때 찾던 것을 발견했을 수도 있고 아닐 수도 있다. 그러나 어쨌든 검색은 거기서 중단한다.

1.4 오류

LHS와 RHS를 구분하는 것이 왜 중요할까? 이 두 종류의 검색 방식은 변수가 아직 선언되지 않았을 때(검색한 모든 스코프에서 찾지 못했을 때) 서로 다르게 동작하기 때문이다.

```
function foo(a) {
    console.log(a + b);
    b = a;
}

foo( 2 );
```

b에 대한 첫 RHS 검색이 실패하면 다시는 b를 찾을 수 없다. 이렇게 스코프에서 찾지 못한 변수는 '선언되지 않은 변수'라 한다. RHS 검색이 중첩 스코프 안 어디에서도 변수를 찾지 못하면 엔진이 'ReferenceError'를 발생시킨다. 여기서 중요한 점은 발생된 오류가 ReferenceError 타입이라는 것이다.

반면에, 엔진이 LHS 검색을 수행하여 변수를 찾지 못하고 꼭대기 층(글로벌 스코프)에 도착할 때 프로그램이 'Strict Mode'[02]로 동작하고 있는 것이 아니라면, 글로벌 스코프는 엔진이 검색하는 이름을 가진 새로운 변수를 생성해서 엔진에게 넘겨준다. 즉, "없어, 없었지만 내가 널 위해 하나 만들어주지"라고 생각하면 된다.

ES5부터 지원하는 'Strict Mode'는 normal/relaxed/lazy mode와는 여러 면에서 다르게 작동한다. 예를 들어, strict mode에서는 글로벌 변수를 자동으로 또는 암시적으로 생성할 수 없다. 그래서 앞의 상황이 닥치면 글로벌 스코프는 변수를

02 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

생성하지 않아서 LHS 검색은 아무 것도 얻지 못하고, 엔진은 RHS의 경우와 비슷하게 ReferenceError를 발생시킨다.

이제, RHS 검색 결과 변수를 찾았지만 그 값을 가지고 불가능한 일을 하려고 할 경우를 보자. 예를 들어, 함수가 아닌 값을 함수처럼 실행하거나 null이나 undefined 값을 참조할 때 엔진은 TypeError를 발생시킨다.

ReferenceError는 스코프에서 대상을 찾았는지와 관계 있지만, TypeError는 스코프 검색은 성공했으나 결과값을 가지고 적법하지 않거나 불가능한 시도를 한 경우를 의미한다.

1.5 복습하기

스코프는 어디서 어떻게 변수(확인자)를 찾는가를 결정하는 규칙의 집합이다. 변수를 검색하는 이유는 변수에 값을 대입하거나(LHS 참조) 변수의 값을 얻어오기 위해서다(RHS 참조).

LHS 참조는 대입 연산 과정에서 일어난다. 스코프와 관련된 대입 연산은 '=' 연산자가 사용되거나 인자를 함수의 매개변수로 넘겨줄 때 일어난다.

자바스크립트 엔진은 코드를 실행하기 전에 먼저 컴파일하는데, 이 과정에서 엔진은 "var a = 2;"와 같은 구문을 독립된 두 단계로 나눈다.

① var a은 변수 a를 해당 스코프에 선언한다. 이 단계는 코드 실행 전에 처음부터 수행된다.
② a = 2는 변수 a를 찾아 값을 대입한다(LHS 참조).

LHS와 RHS 참조 검색은 모두 현재 실행 중인 스코프에서 시작한다. 그리고 필요하다면(대상 변수를 찾지 못했을 경우) 한 번에 한 스코프씩 중첩 스코프의 상위 스코프

로 넘어가며 확인자를 찾는다. 이 작업은 글로벌 스코프(꼭대기 층)에 이를 때까지 계속하고, 대상을 찾았든 못 찾았든 작업을 중단한다.

RHS 참조가 대상을 찾지 못하면 ReferenceError가 발생한다. LHS 참조가 대상을 찾지 못하면 자동적, 암시적으로 글로벌 스코프에 같은 이름의 새로운 변수가 생성된다(만약 'Strict Mode'일 경우 ReferenceError가 발생함).

1.5.1. 퀴즈 답안

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

① 모든 LHS 검색을 찾아보라(모두 3개다!).

c = …;, a = 2(암시적 매개변수 대입), b = …

② 모든 RHS 검색을 찾아보라(모두 4개다!).

foo(2 …, = a;, a …, … b

2 | 렉시컬 스코프

1장에서 '스코프'를 엔진이 확인자 이름으로 현재 스코프 또는 중첩 스코프 내에서 변수를 찾을 때 사용하는 '규칙의 집합'이라고 정의했다.

스코프는 두 가지 방식으로 작동한다. 첫 번째 방식은 다른 방식보다 훨씬 더 일반적이고 다수의 프로그래밍 언어가 사용하는 방식이다. 이 방식을 '렉시컬 스코프Lexical Scope'라고 부른다. 이 장에서는 렉시컬 스코프에 관해 면밀히 검토하겠다. 두 번째 방식은 Bash Scripting이나 Perl의 일부 모드와 같은 몇몇 언어에서 사용하는 방식으로 '동적 스코프Dynamic Scope'라고 부른다.

동적 스코프에 대해서는 부록 A에서 다루고, 이번 장에서는 오직 자바스크립트에서 채용한 렉시컬 스코프와 대비하기 위해 잠깐 언급한다. 동적 스코프를 알고 싶은 독자는 부록 A를 참고하길 바란다.

2.1 렉스타임

1장에서 다룬 것처럼 일반적인 언어의 컴파일러는 첫 단계를 전통적으로 토크나이징 또는 렉싱이라 불리는 작업으로 시작한다. 렉싱 처리 과정에서는 소스 코드 문자열을 분석하여 상태유지 파싱의 결과로 생성된 토큰에 의미를 부여한다. 바로 이 개념이 렉시컬 스코프가 무엇인지, 어원이 어디인지를 알 게 해주는 바탕이 된다.

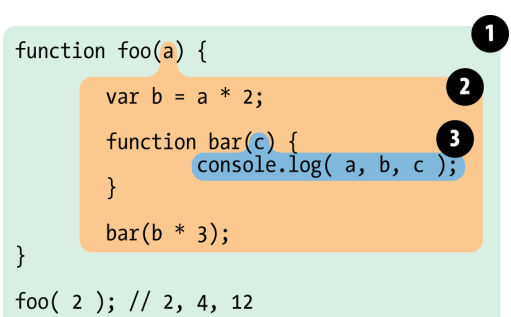
약간 순환적인 정의하면 렉시컬 스코프는 렉싱 타임lexing time에 정의되는 스코프다. 바꿔 말해, 렉시컬 스코프는 프로그래머가 코드를 짤 때 변수와 스코프 블록을 어디서 작성하는가에 기초해서 렉서lexer가 코드를 처리할 때 확정된다.

NOTE_

뒤에서 렉시컬 스코프를 속여 렉서가 지나간 이후 수정하는 방법에 대해 간단히 알아볼 것이다. 그러나 이는 권장하지 않는 방법이다. 렉시컬 스코프를 수정하는 가장 좋은 방법은 오직 구문과 관련해서 변경하는 것, 즉 사실상 코드 작성 때만 수정하는 것이다.

```
function foo(a) {
    var b = a * 2;
    function bar(c) {
        console.log( a, b, c );
    }
    bar( b * 3 );
}
foo( 2 ); // 2, 4, 12
```

이 예제 코드에는 3개의 중첩 스코프가 있다. 스코프를 다음과 같이 겹쳐진 버블이라고 가정하면 이해하기 쉽다.



버블 ❶은 글로벌 스코프를 감싸고 있고, 그 스코프 안에는 오직 하나의 확인자(foo)만 있다.
버블 ❷는 foo의 스코프를 감싸고 있고, 그 스코프는 3개의 확인자(a, bar, b)를 포함한다.
버블 ❸은 bar의 스코프를 감싸고 있고, 그 스코프는 하나의 확인자(c)만을 포함한다.

스코프 버블은 스코프 블록이 쓰이는 곳에 따라 결정되는데, 스코프 블록은 서로 중첩될 수 있다. 다음 장에서 다르게 구성되는 스코프도 다루겠지만, 지금은 각각의 함수가 새로운 스코프 버블을 생성한다고 가정하자.

bar의 버블은 foo의 버블 내부에 완전히 포함된다. 바로 foo의 내부에서 bar 함수를 정의했기 때문이다.

그림에서 중첩 버블의 경계가 엄밀하게 정해져 있는 것이 보이는가? 보고 있는 버블은 서로의 경계가 교차할 수 있는 벤다이어그램이 아니다. 다시 말해, 어떤 함수의 버블도 동시에 (일부라도) 다른 두 스코프 버블 안에 존재할 수 없다. 어떤 함수도 두 개의 부모 함수 안에 존재할 수 없는 것처럼 말이다.

2.1.1 검색

엔진은 스코프 버블의 구조와 상대적 위치를 통해 어디를 검색해야 확인자를 찾을 수 있는지 안다.

앞의 코드를 보면 엔진은 console.log(…) 구문을 실행하고 3개의 참조된 변수 a, b, c를 검색한다. 검색은 가장 안쪽의 스코프 버블인 bar(…) 함수의 스코프에서 시작한다. 여기서 a를 찾지 못하면 다음으로 가장 가까운 스코프 버블인 foo(…)의 스코프로 한 단계 올라가고, 이곳에서 a를 찾아 사용한다. 똑같은 방식이 b에도 적용된다. 단, c는 bar(…) 내부에서 찾을 수 있다.

변수 c가 bar(…)와 foo(…) 내부에 모두 존재한다고 가정하면, console.log(…) 구문은 bar(…) 내부에 있는 c를 찾아서 사용하고 foo(…)에는 c를 찾으러 가지도 않는다.

스코프가 목표와 일치하는 대상을 찾는 즉시 검색을 중단한다. 여러 중첩 스코프 층에 걸쳐 같은 확인자 이름을 정의할 수 있다. 이를 '섀도우잉shadowing'이라 한다

(더 안쪽의 확인자가 더 바깥쪽의 확인자를 가리는 것이다). 섀도우잉과 상관없이 스코프 검색은 항상 실행 시점에서 가장 안쪽 스코프에서 시작하여 최초 목표와 일치하는 대상을 찾으면 멈추고, 그 전까지는 바깥/위로 올라가면서 수행한다.

NOTE_

글로벌 변수는 자동으로 웹 브라우저의 window 같은 글로벌 객체에 속한다. 따라서 글로벌 변수를 직접 렉시컬 이름으로 참조하는 것뿐만 아니라 글로벌 객체의 속성을 참조해 간접적으로 참조할 수도 있다.

```
window.a
```

가려져 있어서 다른 방식으로는 접근할 수 없는 글로벌 변수에는 이 방법을 통해 접근할 수 있다. 그러나 글로벌이 아닌 섀도우 변수는 접근할 수 없다.

어떤 함수가 어디서 또는 어떻게 호출되는지에 상관없이 함수의 렉시컬 스코프는 함수가 선언된 위치에 따라 정의된다.

렉시컬 스코프 검색 과정은 a, b, c 같은 일차 확인자 검색에만 적용된다. 코드에서 foo.bar.baz의 참조를 찾는다고 하면 렉시컬 스코프 검색은 foo 확인자를 찾는 데 사용되지만, 일단 foo를 찾고 나서는 객체 속성 접근 규칙을 통해서 bar와 baz의 속성을 각각 가져온다.

2.2 렉시컬 속이기

렉시컬 스코프는 프로그래머가 작성할 때 함수를 어디에 선언했는지에 따라 결정된다. 그렇다면 런타임 때 어떻게 렉시컬 스코프를 수정할 (또는 속일) 수 있을까?

자바스크립트에서는 렉시컬 스코프를 속일 수 있는 두 가지 방법이 있다. 두 방법

모두 개발자 커뮤니티에서는 코드 작성할 때 권장하지 않는 방법이다. 그러나 많은 사림이 이런 방법을 비판하지만, 가장 중요한 논점을 빠트린다. 바로 렉시컬 스코프를 속이는 방법은 성능을 떨어뜨린다는 점이다.

성능 문제를 설명하기 전에 앞서 말한 두 가지 방법이 어떻게 동작하는지 살펴보자.

2.2.1 eval

자바스크립트의 eval(…) 함수는 문자열을 인자로 받아들여 실행 시점에 문자열의 내용을 코드의 일부분처럼 처리한다. 즉, 처음 작성한 코드에 프로그램에서 생성한 코드를 집어넣어 마치 처음 작성될 때부터 있던 것처럼 실행한다.

eval(…)의 성격을 생각해보면 eval(…)를 통해 어떻게 렉시컬 스코프를 수정하고 원래 작성했던 코드인양 속일 수 있는지 이해할 수 있다. eval(…)이 실행된 후 코드를 처리할 때 엔진은 지난 코드가 동적으로 해석되어 렉시컬 스코프를 변경시켰지 알 수도 없고 관심도 없다. 엔진은 그저 평소처럼 렉시컬 스코프를 검색할 뿐이다.

```
function foo(str, a) {
    eval( str ); // cheating!
    console.log( a, b );
}
var b = 2;
foo( "var b = 3;", 1 ); // 1, 3
```

문자열 “var b = 3;”은 eval(…)이 호출되는 시점에 원래 있던 코드인 것처럼 처리된다. 이 코드는 새로운 변수 b를 선언하면서 이미 존재하는 foo(…)의 렉시컬 스코프를 수정한다. 사실, 앞에서 언급한 것처럼 이 코드는 실제로 foo(…) 안에 변수 b를 생성하여 바깥(글로벌) 스코프에 선언된 변수 b를 가린다.

console.log(…)가 호출될 때 a와 b 모두 foo(…)의 스코프에서 찾을 수 있으므로 바깥의 b는 아예 찾지도 않는다. 따라서 결과값은 일반적인 경우처럼 "1, 2"가 아니라 "1, 3"이 나온다.

NOTE_

이 예제에서는 설명을 위해 넘겨주는 코드 문자열을 단순히 고정된 문자로 한정했다. 그러나 이런 문자열은 프로그램 로직을 이용하여 문자를 합쳐서 쉽게 생성할 수 있다. eval(…)은 흔히 동적으로 생성된 코드를 실행할 때 사용된다. 이는 고정된 문자열에서 정적 코드를 동적으로 생성하는 것은 코드를 직접 입력하는 것보다 이득이 없기 때문이다.

기본적으로 코드 문자열이 하나 이상의 변수 또는 함수 선언문을 포함하면 eval(…)이 그 코드를 실행하면서 eval(…)이 호출된 위치에 있는 렉시컬 스코프를 수정한다. 기술적으로 다양한 트릭(여기서 다룰 범위를 넘어선다)을 이용해 eval(…)을 간접적으로 호출할 수 있고, 이 경우 코드는 글로벌 스코프 속에서 실행되고 글로벌 스코프를 수정한다. 어떤 경우든 eval(…)은 프로그래머가 작성했던 때의 렉시컬 스코프를 런타임에서 수정할 수 있다.

NOTE_

Strict Mode 프로그램에서 eval(…)을 사용하면 eval(…)은 자체적인 렉시컬 스코프를 이용한다. 즉, eval( ) 내에서 실행된 선언문은 현재 위치의 스코프를 실제로 수정하지 않는다.

```
function foo(str) {
   "use strict";
   eval( str );
   console.log( a ); // ReferenceError: a is not defined
}
foo( "var a = 2" );
```

자바스크립트에는 eval(…)과 매우 비슷한 효과를 내는 다른 방법이 있다. setTimeout(…)과 setInterval(…)은 첫째 인자로 문자열을 받을 수 있고, 문자열의 내용은 동적 생성된 함수 코드처럼 처리된다. 이 방법은 구식에다가 없어질 예정이니 사용하지 말자!

함수 생성자 new Function(…)도 비슷한 방식으로 코드 문자열을 마지막 인자로 받아서 동적으로 생성된 함수로 바꾼다(입력받은 시작 인자들이 있으면 생성된 함수의 매개변수로 사용된다). 이 함수 생성자 문법은 eval(…)보다는 좀 더 안전하지만, 여전히 코드에 사용하지 않는 것이 좋다.

동적으로 생성한 코드를 프로그램에서 사용하는 경우는 굉장히 드물다. 사용할 때 성능 저하를 감수할 만큼 활용도가 높지 않기 때문이다.

2.2.2 with

키워드 with는 렉시컬 스코프를 속일 수 있는 자바스크립트의 또 다른 기능이다(하지만 사용을 권장하지 않는다. 이 기능은 곧 없어질 예정이다). with를 설명하는 방법이 여럿 있지만, 필자는 with가 렉시컬 스코프와 어떻게 상호작용하고 스코프에 어떤 영향을 주느냐는 관점에서 설명하고자 한다.

with는 일반적으로 한 객체의 여러 속성을 참조할 때 객체 참조를 매번 반복하지 않기 위해 사용하는 일종의 속기법이라 할 수 있다.

```
var obj = {
    a: 1,
    b: 2,
    c: 3
};
// more "tedious" to repeat "obj"
```

```
obj.a = 2;
obj.b = 3;
obj.c = 4;

// "easier" short-hand
with (obj) {
    a = 3;
    b = 4;
    c = 5;
}
```

---

with는 단순히 객체 속성을 편하게 접근할 수 있는 속기법 이상의 효과가 있다.

```
function foo(obj) {
    with (obj) {
        a = 2;
    }
}

var o1 = {
    a: 3
};

var o2 = {
    b: 3
};

foo( o1 );
console.log( o1.a );   // 2

foo( o2 );
```

```
console.log( o2.a );   // undefined
console.log( a );      // 2—Oops, leaked global!
```

이 예제에서 객체 o1과 o2가 생성됐다. 하나는 a라는 속성이 있고, 다른 하나는 그런 속성이 없다. 함수 foo(…)는 객체 참조 obj를 인자로 받아서 with (obj) { … }를 호출하고, with 블록 내부에서 변수 a에 대한 평범한 렉시컬 참조를 수행한다. 이는 LHS 참조로 변수 a를 찾아 값 2를 대입하는 작업이다.

o1을 인자로 넘기면 "a = 2" 대입문 처리 과정에서 o1.a를 찾아 값 2를 대입한다. 그 결과는 다음 줄에 호출된 console.log(o1.a) 문에 반영된다. 그러나 o2를 인자로 넘길 때는 o2에 a라는 속성이 없으므로 새로이 속성이 생성되지 않고 o2.a는 undefined로 남는다.

이때 발생하는 상당히 특이한 부작용에 주목해야 한다. 바로 대입문 "a = 2"가 글로벌 변수 a를 생성한다는 점이다. 어떻게 이런 일이 가능할까?

with 문은 속성을 가진 객체를 받아 마치 하나의 독립된 렉시컬 스코프처럼 취급한다. 따라서 객체의 속성은 모두 해당 스코프 안에 정의된 확인자로 간주된다. 물론 with 블록이 객체를 하나의 렉시컬 스코프로 취급하기는 하지만, with 블록 안에서 일반적인 var 선언문이 수행될 경우 선언된 변수는 with 블록이 아니라 with를 포함하는 함수의 스코프에 속한다.

eval(…)은 인자로 받은 코드 문자열에 하나 이상의 선언문이 있을 경우 이미 존재하는 렉시컬 스코프를 수정할 수 있지만, with 문은 넘겨진 객체를 가지고 난데없이 사실상 하나의 새로운 렉시컬 스코프를 생성한다.

이렇게 볼 때, o1을 넘겨 받은 with 문은 o1이라는 스코프를 선언하고 그 스코프는 o1.a 속성에 해당하는 확인자를 가진다. 그러나 o2가 스코프로 사용되면 그 스

코프에는 a 확인자가 없으므로 이후 작업은 일반적인 LHS 확인자 검색 규칙(1장 참조)에 따라 진행된다.

o2의 스코프, foo(…)의 스코프, 그리고 글로벌 스코프에서도 a 확인자는 찾을 수 없다. 따라서 "a = 2"가 수행되면 자동으로 그에 해당하는 글로벌 변수가 생성된다(현재 작업 환경은 non-strict mode다).

런타임에 with 문이 하나의 객체와 그 속성을 하나의 스코프와 확인자로 바꾸는 동작은 매우 이해하기 어렵고 이상한 일이다. 하지만 이것이 결과를 가장 잘 설명한다.

> **NOTE_**
>
> eval(…)과 with의 사용은 권장하지 않을 뿐만 아니라 Strict Mode에서는 이 둘 모두 사용이 제한된다. with는 명시적으로 사용이 금지되었고, eval(…)은 핵심 기능은 남았지만 간접적이고 위험한 사용 방식은 금지되었다.

2.2.3 성능

런타임에 스코프를 수정하거나 새로운 렉시컬 스코프를 만드는 방법으로 eval(…)과 with 모두 원래 작성된 렉시컬 스코프를 속인다. 이게 무슨 문제냐고 질문할 수도 있다. 이런 요소로 더 복잡한 기능과 코딩의 유연성을 얻을 수 있다면 좋은 특성이라 할 수 있지 않을까? 그렇지 않다.

자바스크립트 엔진은 컴파일레이션 단계에서 상당수의 최적화 작업을 진행한다. 이 최적화의 일부분이 하는 핵심 작업은 렉싱된 코드를 분석하여 모든 변수와 함수 선언문이 어디에 있는지 파악하고 실행 과정에서 확인자 검색을 더 빠르게 하는 것이다.

그러나 eval(…)이나 with가 코드에 있다면 엔진은 미리 확인해둔 확인자의 위치

가 틀릴 수도 있다고 가정해야 한다. 렉싱 타임에는 엔진은 eval(…)에 어떤 코드가 전달되어 렉시컬 스코프가 수정될지 정확하게 알 수 없고, with에 넘긴 객체의 내용에 따라 새로운 렉시컬 스코프가 생성될 수 있기 때문이다. 즉, eval(…)이나 with가 코드에 있다면 대다수 최적화가 의미 없어져서 아무런 최적화도 하지 않은 것이나 마찬가지가 되어 버린다.

따라서 단순히 코드 어딘가에서 eval(…)이나 with를 사용했다는 사실 하나만으로 그 코드는 거의 확실히 더 느리게 동작할 것이다. 엔진이 아무리 똑똑하게 이런 부작용을 줄이려고 노력해도 최적화 없이는 코드가 느리게 동작한다는 사실은 피할 수 없다.

2.3 복습하기

렉시컬 스코프란 프로그래머가 코드를 작성할 때 함수를 어디에 선언하는지에 따라 정의되는 스코프를 말한다. 컴파일레이션의 렉싱 단계에서는 모든 확인자가 어디서 어떻게 선언됐는지 파악하여 실행 단계에서 어떻게 확인자를 검색할지 예상할 수 있도록 도와준다.

자바스크립트에는 렉시컬 스코프를 속이는 두 가지 방식이 있는데, eval(…)과 with다. eval(…)은 하나 이상의 선언문을 포함하는 코드 문자열을 해석하여 렉시컬 스코프가 있다면 런타임에 이를 수정한다. with는 객체 참조를 하나의 스코프로, 속성을 확인자로 간주하여 런타임에 완전히 새로운 렉시컬 스코프를 생성한다.

이런 방식의 단점은 eval(…)과 with가 엔진이 컴파일 단계에서 수행한 스코프 검색과 관련된 최적화 작업을 무산시킨다는 점이다. 이들이 수행되면 엔진은 최악의 경우를 대비해 진행했던 최적화 결과가 무효화됐다고 가정해야 하기 때문이다. 따라서 eval(…)과 with 중 하나라도 사용하면 코드는 더 느리게 동작하므로 이 방식은 사용하지 말자.

3 | 함수 vs 블록 스코프

2장에서 살펴본 것처럼 스코프는 컨테이너 또는 바구니 구실을 하는 일련의 '버블'이고 변수나 함수 같은 확인자가 그 안에서 선언된다. 이 버블은 경계가 분명하게 중첩되고, 그 경계는 프로그래머가 코드를 작성할 때 결정된다.

그렇다면 정확히 어떤 것이 새로운 버블을 만들까? 함수만 버블을 만들까? 자바스크립트의 다른 자료 구조는 스코프 버블을 생성하지 못할까?

3.1 함수 기반 스코프

앞의 질문에 대한 가장 일반적인 답변은 자바스크립트가 함수 기반 스코프를 사용하기 때문이라는 것이다. 즉, 각각의 선언된 함수는 저마다의 버블을 생성하지만 다른 어떤 자료구조도 자체적인 스코프를 생성하지 않는다는 것이다. 뒤에서 살펴보겠지만, 이는 전혀 사실이 아니다. 먼저 함수 스코프와 그 암시적 용례를 살펴보자.

```
function foo(a) {
    var b = 2;

    // some code

    function bar() {
        // ...
    }

    // more code

    var c = 3;
}
```

앞의 코드에서 foo(…)의 스코프 버블은 확인자 a, b, c와 bar를 포함한다. 선언문이 스코프의 어디에 있는지는 중요하지 않다. 스코프 안에 있는 모든 변수와 함수는 그 스코프 버블에 속한다. 다음 장에서는 스코프가 어떻게 이런 식으로 동작하는지에 관해 알아본다.

bar(…)는 자체 스코프 버블이 있고, 글로벌 스코프도 마찬가지다. 그리고 글로벌 스코프에는 하나의 확인자가 있는데, 바로 'foo'다.

a, b, c, bar 모두 foo(…)의 스코프 버블에 속하므로 foo(…) 바깥에서는 이들에게 접근할 수 없다. 따라서 다음 코드는 호출된 확인자가 글로벌 스코프에는 없기 때문에 ReferenceError 오류를 발생시킨다.

```
bar();                  // fails
console.log( a, b, c );    // all 3 fail
```

하지만 이 모든 확인자(a, b, c, foo, bar)는 foo(…) 안에서 접근할 수 있고, bar(…) 안에서도 이용할 수 있다(bar(…) 내부에서 새도우 확인자가 선언되지 않았을 때).

함수 스코프는 모든 변수가 함수에 속하고 함수 전체(심지어 중첩된 스코프에서도)에 걸쳐 사용되며 재사용된다는 개념을 확고하게 한다. 이런 디자인 접근법은 상당히 유용하고 자바스크립트 변수의 '동적' 특성을 완전히 살려 다른 타입의 값을 필요에 따라 가져올 수 있지만, 스코프 전체에서 변수가 살아있다는 점이 예상치 못한 문제를 일으킬 수도 있다.

3.2 일반 스코프에 숨기

함수에 대한 전통적인 개념은 이렇다.

- 함수를 선언하고 그 안에 코드를 넣는다. 바꿔 생각해보는 것도 꽤 유용하다.
- 작성해놓은 코드의 임의의 부분을 함수 선언문으로 감싼다. 이는 해당 코드를 '숨기는' 효과를 낸다.

이렇게 하면 해당 코드 주위에 새로운 스코프 버블이 생성된다. 즉, 감싸진 코드 안에 있는 '모든 변수' 또는 '함수 선언문'은 이전 코드에 포함됐던 스코프가 아니라 새로이 코드를 감싼 함수의 스코프에 묶인다. 달리 말하면, 함수의 스코프로 둘러싸서 변수와 함수를 '숨길' 수 있다는 말이다. 그렇다면 코드를 '숨기는' 테크닉이 어디에 유용할까?

스코프를 이용해 숨기는 방식을 사용하는 이유는 여러 가지가 있는데, 소프트웨어 디자인 원칙인 '최소 권한의 원칙'('최소 권위' 또는 '최소 노출'이라고도 부른다)과 관련이 있다. 이 원칙은 모듈/객체의 API 같은 소프트웨어를 설계할 때 필요한 것만 최소한으로 남기고 나머지는 '숨겨야' 한다는 것이다.

이 원칙은 어떤 스코프가 변수와 함수를 포함하는지에 관한 문제와도 관련이 있다. 모든 변수와 함수가 글로벌 스코프에 존재한다면 어느 중첩된 하위 스코프에서도 이들에 접근할 수 있다. 이는 '최소 …'의 원칙을 어기는 것이고, 코드를 적절하게 사용했을 때 접근할 필요가 없어서 비공개로 남겨둬야 할 많은 변수나 함수를 노출시키게 된다.

```
function doSomething(a) {
    b = a + doSomethingElse( a * 2 );
```

```
    console.log( b * 3 );
}

function doSomethingElse(a) {
    return a - 1;
}

var b;

doSomething( 2 ); // 15
```

이 코드에서 변수 b와 함수 doSomethingElse(…)는 doSomething(…)이 어떤 작업을 하는지 보여주는 '비공개' 부분이라고 할 수 있다. 변수 b와 doSomethingElse(…)에 '접근'할 수 있도록 내버려 두는 것은 불필요할 뿐 아니라 '위험'할 수 있다. 접근 가능한 확인자는 의도적이든 아니든 생각지 못한 방식으로 사용될 수 있으며, doSomething(…)의 실행에 전제되는 가정들이 훼손될 수 있다.

더 '적절하게' 설계하려면 다음과 같이 앞의 비공개 부분은 doSomething(…) 스코프 내부에 숨겨야 한다.

```
function doSomething(a) {
    function doSomethingElse(a) {
        return a - 1;
    }
    var b;
    b = a + doSomethingElse( a * 2 );
    console.log( b * 3 );
}

doSomething( 2 ); // 15
```

이제 b와 doSomethingElse(…)는 외부에서 접근할 수 없어서 더는 바깥의 영향을 받지 않고, 오직 doSomething(…)만이 이들을 통제한다. 기능과 최종 결과는 달라지지 않았지만, 변경된 디자인은 비공개로 해야 할 내용을 확실하게 비공개로 둔다. 일반적으로 이것이 더 나은 코드라고 본다.

3.2.1 충돌 회피

변수와 함수를 스코프 안에 '숨기는 것'의 또 다른 장점은 같은 이름을 가졌지만 다른 용도를 가진 두 확인자가 충돌하는 것을 피할 수 있다는 점이다. 이런 충돌은 종종 예상하지 못한 변수값의 겹쳐 쓰기를 초래한다.

```
function foo() {
    function bar(a) {
        i = 3; // changing the `i` in the enclosing scope's
        // for-loop
        console.log( a + i );
    }

    for (var i=0; i<10; i++) {
        bar( i * 2 ); // oops, inifinite loop ahead!
    }
}

foo();
```

bar(…) 내부의 대입문 "i = 3"은 예기치 않게 foo(…)에서 for 반복문을 위해 선언된 변수 i의 값을 변경한다. 그 결과 이 코드는 무한 반복에 빠진다. 변수 i의 값이 3으로 고정되어 영원히 'i<10'인 상태로 머물기 때문이다.

bar(…) 내부의 대입문은 어떤 확인자 이름을 고르든 지역 변수로 선언해서 사용해야 한다. "var i = 3;"으로 변경하면 문제를 해결할 수 있다(이는 앞서 언급한 i에 대한 '가려진 변수'를 선언하는 것이다). 이를 대체할 수는 없지만, 추가적인 선택안은 "var j = 3;"과 같이 완전히 다른 확인자 이름을 고르는 것이다. 그러나 소프트웨어 설계를 하다 보면 자연스럽게 같은 확인자 이름을 사용하므로 스코프를 이용해서 내부에 선언문을 '숨기는' 것이 가장 좋은, 그리고 유일한 선택지다.

글로벌 '네임스페이스'

글로벌 스코프에서 변수 충돌이 특히 일어나기 쉬운 경우에 대해 알아보자. 내부/비공개 함수와 변수가 적절하게 숨겨 있지 않은 여러 라이브러리를 한 프로그램에서 불러오면 라이브러리들은 서로 쉽게 충돌한다.

이런 라이브러리는 일반적으로 글로벌 스코프에 하나의 고유 이름을 가지는 객체 선언문을 생성한다. 이후 객체는 해당 라이브러리의 '네임스페이스'로 이용된다. 네임스페이스를 통해 최상위 스코프의 확인자가 아니라 속성 형태로 라이브러리의 모든 기능들이 노출된다.

```
var MyReallyCoolLibrary = {
    awesome: "stuff",
    doSomething: function() {
        // ...
    },
    doAnotherThing: function() {
        // ...
    }
};
```

모듈 관리

좀 더 현대적인 충돌 방지 옵션으로는 다양한 의존성 관리자를 이용한 '모듈' 접근법이 있다. 이 도구를 사용하면 어떤 라이브러리도 확인자를 글로벌 스코프에 추가할 필요 없이, 특정 스코프로부터 의존성 관리자를 이용한 다양한 명시적인 방법으로 확인자를 가져와 사용할 수 있다.

기억할 것은 이런 도구를 사용한다고 '마법'처럼 렉시컬 스코프 규칙을 벗어날 수 있는 것이 아니라는 점이다. 의존성 관리자는 그저 여기서 설명한 스코프 규칙을 적용해 모든 확인자가 공유 스코프에 누출되는 것을 방지하고, 우발적인 스코프 충돌을 예방하기 위해 충돌 위험이 없는 비공개의 스코프에 확인자를 보관한다.

물론 하려고 한다면 방어적으로 코딩하여 실제 의존성 관리자를 사용하지 않고도 사용한 것과 같은 결과를 얻을 수 있다. 모듈 패턴에 대한 좀 더 자세한 정보는 5장을 참고하라.

3.3 스코프 역할을 하는 함수

지금까지 코드를 함수로 감싸 내부에 변수나 함수 선언문을 바깥 스코프로부터 함수의 스코프 안에 '숨기는' 것을 살펴보았다.

```
var a = 2;

function foo() { // <-- insert this
    var a = 3;
    console.log( a ); // 3
} // <-- and this

foo(); // <-- and this
console.log( a ); // 2
```

이 방식은 작동하기는 하지만, 결코 이상적인 방식은 아니다. 이 방식에는 몇 가지 문제가 있다. 첫째, foo()라는 이름의 함수를 선언해야 한다. 즉, foo라는 확인자 이름으로 둘러싸인 스코프(이 경우에는 글로벌 스코프)를 '오염시킨다'는 의미다. 또한, 그 함수를 직접 이름(foo())으로 호출해야만 실제 감싼 코드를 실행할 수 있다.

함수를 이름 없이 (아니면 그 이름이 둘러싸인 스코프를 오염시키지 않고) 선언하고 자동으로 실행된다면 더 이상적일 것이다.

다행히도 자바스크립트에서는 두 가지 문제를 모두 해결할 방법이 있다.

```
var a = 2;

(function foo(){ // <-- insert this
    var a = 3;
    console.log( a ); // 3
})(); // <-- and this

console.log( a ); // 2
```

이 코드가 어떤 식으로 작동하는지 하나 하나 살펴보자.

먼저, 코드를 감싼 함수는 'function …'이 아니라 '(function …' 으로 시작한다. 별 다를 바 없어 보일 수도 있지만, 실제로는 큰 변화를 가져온다. 이 코드에서 함수는 보통의 선언문이 아니라 함수 표현식으로 취급된다.

NOTE_

선언문과 표현식을 구분하는 가장 쉬운 방법은 'function'이라는 단어가 구문에서 어디에 위치하는가를 살펴보면 된다(한 줄에서가 아니라 하나의 독립 구문에서 봐야 한다). 'function'이 구문의 시작 위치에 있다면 함수 선언문이고, 다른 경우는 함수 표현식이다.

여기서 볼 수 있는 함수 선언문과 함수 표현식의 중요한 차이는 함수 이름이 어디의 확인자로 묶이느냐와 관련 있다.

앞의 두 코드를 비교해보자. 첫째 코드에서 함수 이름 foo는 함수를 둘러싼 스코프에 묶이고, foo( )라는 이름을 통해 직접 호출했다. 두 번째 코드에서 함수 이름 foo는 함수를 둘러싼 스코프에 묶이는 대신 함수 자신의 내부 스코프에 묶였다. 즉, "(function foo( ) { … })"라는 표현식에서 확인자 foo는 오직 '…'가 가리키는 스코프에서만 찾을 수 있고 바깥 스코프에서는 발견되지 않는다. 함수 이름 foo를 자기 내부에 숨기면 함수를 둘러싼 스코프를 불필요하게 오염시키지 않는다.

3.3.1. 익명 vs 기명

다음과 같이 함수 표현식을 콜백 매개변수로 사용하는 사례에 익숙할 것이다:

```
setTimeout( function(){
    console.log("I waited 1 second!");
}, 1000 );
```

이런 방식을 '익명 함수 표현식'이라 부르는데, 이는 "function( ) …"에 확인자 이름이 없기 때문이다. 함수 표현식은 이름 없이 사용할 수 있지만, 함수 선언문에는 이름이 빠져서는 안 된다. 이름 없는 함수 선언문은 자바스크립트 문법에 맞지 않다.

익명 함수 표현식은 빠르고 쉽게 입력할 수 있어서 많은 라이브러리와 도구가 이 자바스크립트 특유의 표현법을 권장한다. 그러나 함수 표현식은 몇 가지 기억할 단점이 있다.

① 익명 함수는 스택 추적 시 표시할 이름이 없어서 디버깅이 더 어려울 수 있다.
② 이름 없이 함수 스스로 재귀 호출을 하려면 불행히도 폐기 예정인 arguments.

callee 참조가 필요하다. 자기 참조가 필요한 또 다른 예로는 한 번 실행하면 해제되는 이벤트 처리 함수가 있다.

③ 이름은 보통 쉽게 이해하고 읽을 수 있는 코드 작성에 도움이 되는데, 익명 함수는 이런 이름을 생략한다. 기능을 잘 나타내는 이름은 해당 코드를 그 자체로 설명하는 데 도움된다.

인라인 함수 표현식은 매우 효과적이고 유용하다. 익명이냐 기명이냐의 문제가 이 사실을 퇴색시키지는 않는다. 함수 표현식에 이름을 사용하면 특별한 부작용 없이 상당히 효과적으로 앞의 단점을 해결할 수 있다. 따라서 함수 표현식을 사용할 때 이름을 항상 쓰는 것이 가장 좋다.

```
setTimeout( function timeoutHandler(){ // <-- Look, I have a name!
    console.log( "I waited 1 second!" );
}, 1000 );
```

3.3.2 함수 표현식 즉시 호출하기

```
var a = 2;

(function foo(){
    var a = 3;
    console.log( a ); // 3
})();

console.log( a ); // 2
```

( )로 함수를 감싸면 함수를 표현식으로 바꾸는데, "(function foo( ) { … }) ( )" 처럼 마지막에 또 다른 ( )를 붙이면 함수를 실행할 수 있다. 함수를 둘러싼 첫 번째 ( )

는 함수를 표현식으로 바꾸고, 두 번째 ( )는 함수를 실행시킨다.

이런 패턴은 굉장히 흔해서 개발자 커뮤니티에서는 몇 년 전 이것을 부르는 용어를 정하기도 했다. 즉시호출함수표현식(IIFE)은 즉시(Immediately) 호출(Invoked) 함수(Function) 표현식(Expression)의 합성어다.

물론, IIFE는 이름이 꼭 필요하지는 않다. IIFE는 익명 함수 표현식으로 가장 흔하게 사용된다. 분명히 덜 흔하기는 하지만, IIFE를 기명으로 사용하면 익명 함수 표현식을 사용하는 것보다 앞서 언급한 것처럼 더 나은 면이 있다. 따라서 기명 IIFE를 사용하는 것은 좋은 습관이다.

```
var a = 2;

(function IIFE(){

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

어떤 이들은 전통적인 IIFE 형태를 약간 변형하여 "(function foo( ) { … }) ( )"로 사용하기도 한다. 자세히 살펴보면 차이가 보일 것이다. 첫째 형태에서 함수 표현식은 ( ) 안에 싸여있고, 호출에 사용되는 ( )가 밖에 바로 붙어 있다. 둘째 형태에서 호출에 사용되는 ( )는 둘러싼 ( ) 안으로 옮겨졌다. 두 형태 모두 똑같이 기능한다. 순전히 어떤 스타일을 선호하느냐의 문제일 뿐이다. 널리 사용되는 또 다른 IIFE의 변형은 IIFE가 결국은 함수라는 사실을 이용해 인자를 넘기는 방식이다.

```
var a = 2;

(function IIFE( global ){
    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2
})( window );

console.log( a ); // 2
```

예제에서는 window 객체 참조를 global이라 이름 붙인 매개변수에 넘겨서 글로벌 참조와 비 글로벌 참조 사이에 명확한 차이를 만들었다. 물론, 해당 스코프에 무엇이든 넘길 수 있고 매개변수 이름도 마음대로 지을 수 있다. 이는 어디까지나 스타일 선호의 문제일 뿐이다.

이 패턴의 다른 예제를 통해 기본 확인자 undefined의 값이 잘못 겹쳐 쓰여 예상치 못한 결과를 야기하는 (드문) 문제에 대해 살펴보자. 매개변수를 undefined라고 이름 짓고 인자로 아무 값도 넘기지 않으면, undefined 확인자의 값은 코드 블록 안에서 undefined 값을 가진다.

```
undefined = true; // setting a land-mine for other code! avoid!

(function IIFE( undefined ){
    var a;
    if (a === undefined) {
        console.log( "Undefined is safe here!" );
    }
})();
```

IIFE의 변형된 형태를 하나 더 보자. 여기서 실행할 함수는 호출문과 넘겨진 매개변수 뒤쪽에 온다. 이 패턴은 UMD[Universal Module Definition](범용 모듈 정의) 프로젝트에서 사용한다. 어떤 이들은 이 방식이 약간은 장황해도 이해하기에는 좀 더 깔끔하다고 생각한다.

```
var a = 2;

(function IIFE( def ){
    def( window );
})(function def( global ){
    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2
});
```

함수 표현식 def는 코드 후반부에 정의되어 코드 전반부에 정의된 IIFE 함수에 (def라는 이름의) 매개변수로 넘겨진다. 결국, 매개변수 함수 def가 호출되고 window가 global 매개변수로 넘겨진다.

3.4 스코프 역할을 하는 블록

함수가 가장 일반적인 스코프 단위이자 현재 자바스크립트에서 통용되는 가장 널리 퍼진 디자인 접근법이기는 하지만, 다른 스코프 단위도 존재하고 이를 이용하면 더 좋은 깔끔한 코드를 작성할 수 있다.

자바스크립트를 제외하고도 많은 언어들이 블록 스코프를 지원한다. 그래서 다른 언어를 사용하던 개발자들은 블록 스코프라는 개념에 익숙하지만, 자바스크립트만을 써왔던 개발자에게는 이 개념이 약간은 어색할 수도 있다.

그러나 아직 블록 스코프 방식으로 한 줄도 코딩해 본 적이 없더라도 다음과 같은 자바스크립트의 코드는 매우 익숙할 것이다:

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

변수 i를 for 반복문의 시작부에 선언하는 이유는 보통 i를 오직 for 반복문과 관련해서 사용하려는 것이다. 그러고는 변수 i가 실제로는 둘러싼 (함수 또는 글로벌) 스코프에 포함된다는 사실을 무시한다. 블록 스코프의 목적이 바로 이것이다. 변수를 최대한 사용처 가까이에서 최대한 작은 유효 범위를 갖도록 선언하는 것 말이다.

```
var foo = true;

if (foo) {
    var bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}
```

변수 bar는 오직 if 문 안에서만 사용하므로, bar를 if 블록 안에 선언하는 것은 타당하다. 그러나 사실 var를 사용할 때 변수를 어디에서 선언하는지는 중요한 문제가 아니다. 선언된 변수는 항상 둘러싸인 스코프에 속하기 때문이다. 앞의 코드는 보기에만 스코프처럼 보이는 '가짜' 블록 스코프로, bar를 의도치 않게 다른 곳에서 사용하지 않도록 상기시키는 역할을 할 뿐이다.

블록 스코프는 앞서 언급한 '최소 권한 노출의 원칙'을 확장하여 정보를 함수 안에 숨기고, 나아가 정보를 코드 블록 안에 숨기기 위한 도구다.

for 반복문 예제를 다시 한 번 보자.

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

오직 for 반복문에서만 사용될 (또는 사용되어야만 할) 변수 i로 왜 함수 스코프 전체를 오염시켜야 할까?

무엇보다 개발자들은 의도하지 않게 변수가 원래 용도 이외의 곳에서 (재)사용됐는지 점검하고 싶어 한다. 예를 들어, 정해진 장소 밖에서 변수가 사용되면 알려지지 않은 변수라는 오류가 발생한다는 식으로 말이다. 블록 스코프를 사용한다면(가능했다고 치자) 변수 i는 오직 for 반복문 안에서만 사용할 수 있고, 이외 함수 어느 곳에서 접근하더라도 오류가 발생할 것이다. 이는 변수가 혼란스럽고 유지 보수하기 어려운 방식으로 재사용되지 않도록 막는다.

그러나 슬픈 현실은 적어도 외견상으로 자바스크립트는 블록 스코프를 지원하지 않는다. 물론, 좀 더 파고들면 방법은 있다.

3.4.1 with

2장에서 with에 대해 배웠다. 비록 with는 지양해야 할 구조이긴 하지만 블록 스코프의 형태를 보여주는 한 예로, with 문 안에서 생성된 객체는 바깥 스코프에 영향 주는 일 없이 with 문이 끝날 때까지만 존재한다.

3.4.2 try/catch

잘 알려지지 않은 사실이지만, 자바스크립트 ES3에서는 try/catch 문 중 catch

부분에서 선언된 변수는 catch 블록 스코프에 속한다.

```
try {
    undefined(); // illegal operation to force an exception!
}
catch (err) {
    console.log( err ); // works!
}
console.log( err ); // ReferenceError: `err` not found
```

예제에서 보듯, 변수 err는 오직 catch 문 안에만 존재하므로 다른 곳에서 참조하면 오류가 발생한다.

NOTE_

이런 동작은 명시되어 있고 실제 (아마도 구식 IE를 제외하면) 모든 표준 자바스크립트 환경에서 똑같이 작동하지만, 많은 린터linter(자바스크립트 스타일 오류를 찾아주는 도구)가 여전히 같은 스코프 내에 같은 확인자 이름으로 오류 변수를 선언한 catch 문이 둘 이상 있을 경우 경고를 보낸다. 사실 이런 선언은 재정의가 아니다. 그 변수들은 안전하게 각기 다른 블록 스코프에 속하기 때문이다. 그래도 린터는 짜증나게 이 부분을 계속 경고한다. 이 불필요한 경고를 피하고자 catch 변수 이름을 err1, err2와 같은 식으로 정하거나 그냥 린터에서 변수 이름 중복 확인 옵션을 꺼버리기도 한다.

catch 문의 블록 스코프 효과는 쓸데없고 학술적인 것처럼 느껴질 수도 있지만, 부록 B를 보면 이 기능을 어떻게 유용하게 활용할 수 있는지 알 수 있다.

3.4.3 let

지금까지 살펴본 자바스크립트의 블록 스코프 기능은 비주류적인 요소를 통해서 구현된 것이다. 자바스크립트의 블록 스코프 기능이 이것뿐이었다면(사실 오랜 기간

동안 이것밖에 없었다) 자바스크립트 개발자들에게 블록 스코프는 별로 유용하지 않았을 것이다.

다행히도, ES6에서 이런 상황이 바뀌면서 새로운 키워드 let이 채택됐다. let은 var 같이 변수를 선언하는 다른 방식이다. 키워드 let은 선언된 변수를 둘러싼 아무 블록(일반적으로 { … })의 스코프에 붙인다. 바꿔 말해, 명시적이진 않지만 let은 선언한 변수를 위해 해당 블록 스코프를 이용한다고도 말할 수 있다.

```
var foo = true;

if (foo) {
    let bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}

console.log( bar ); // ReferenceError
```

let을 이용해 변수를 현재 블록에 붙이는 것은 약간 비명시적이다. 코드를 작성하다 보면 블록이 왔다갔다하고 다른 블록으로 감싸기도 하는데, 이럴 때 주의하지 않으면 변수가 어느 블록 스코프에 속한 것인지 착각하기 쉽다.

블록 스코프에 사용하는 블록을 명시적으로 생성하면 이런 문제를 해결할 수 있다. 변수가 어느 블록에 속했는지 훨씬 더 명료해지기 때문이다. 일반적으로 명시적인 코드가 암시적이고 미묘한 코드보다 낫다. 이런 명시적 블록 스코프 스타일은 쉽게 사용할 수 있고, 다른 언어에서 블록 스코프가 작동하는 방식과도 더 자연스럽게 만난다.

```
var foo = true;

if (foo) {
    { // <-- explicit block
        let bar = foo * 2;
        bar = something( bar );
        console.log( bar );
    }
}

console.log( bar ); // ReferenceError
```

그저 { … }를 문법에 맞게 추가만 해도 let을 통해 선언된 변수를 묶을 수 있는 임의의 블록을 생성할 수 있다. 앞의 코드에서 if 문 안에 명시적인 블록을 만들었다. 이렇게 하면 나중에 리팩토링하면서 if 문의 위치나 의미를 변화시키지 않고도 전체 블록을 옮기기가 쉬워진다.

NOTE_

다른 방식으로 명시적인 블록 스코프를 표현하는 법을 알고 싶다면 부록 B를 참고하라.

4장에서는 호이스팅hoisting(끌어올리기)에 대해 배울 것이다. 호이스팅은 선언문이 어디에서 선언됐든 속하는 스코프 전체에서 존재하는 것처럼 취급되는 작용을 말한다. 그러나 let을 사용한 선언문은 속하는 스코프에서 호이스팅 효과를 받지 않는다. 따라서 let으로 선언된 변수는 실제 선언문 전에는 명백하게 '존재'하지 않는다.

```
{
    console.log( bar ); // ReferenceError!
    let bar = 2;
}
```

가비지 콜렉션Garbage collection

블록 스코프가 유용한 또 다른 이유는 메모리를 회수하기 위한 클로저 그리고 가비지 콜렉션과 관련 있다. 여기서는 간단히 다루지만, 클로저의 메커니즘은 5장에서 자세히 설명하겠다.

```
function process(data) {
    // do something interesting
}
var someReallyBigData = { .. };

process( someReallyBigData );

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

클릭을 처리하는 click 함수는 someReallyBigData 변수가 전혀 필요없다. 따라서 이론적으로는 process(…)가 실행된 후, 많은 메모리를 먹는 자료구조인 someReallyBigData는 수거할 수도 있다. 그러나 (어떻게 구현됐는지에 따라 다를 수 있지만) 자바스크립트 엔진은 그 데이터를 여전히 남겨둘 것이다. click 함수가 해당 스코프 전체의 클로저를 가지고 있기 때문이다. 블록 스코프는 엔진에게 someReallyBigData가 더는 필요없다는 사실을 더 명료하게 알려서 이 문제를 해결할 수 있다.

```
function process(data) {
    // do something interesting
}

// anything declared inside this block can go away after!
{
    let someReallyBigData = { .. };

    process( someReallyBigData );
}

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

명시적으로 블록을 선언하여 변수의 영역을 한정하는 것은 효과적인 코딩 방식이므로 익혀두면 좋다.

let 반복문

let은 앞에서 살펴본 for 반복문에서 특히 유용하게 사용할 수 있다.

```
for (let i=0; i<10; i++) {
    console.log( i );
}

console.log( i ); // ReferenceError
```

let은 단지 i를 for 반복문에 묶었을 뿐만 아니라 반복문이 돌 때마다 변수를 다시 묶어서 이전 반복의 결과값이 제대로 들어가도록 한다.

다음 예제는 반복마다 다시 묶는 작용을 보여준다.

```
{
    let j;
    for (j=0; j<10; j++) {
        let i = j; // re-bound for each iteration!
        console.log( i );
    }
}
```

이런 반복마다 다시 묶는 작용이 흥미로운 이유는 5장에서 클로저를 배울 때 명확하게 알 수 있다.

let 선언문은 둘러싼 함수 (또는 글로벌) 스코프가 아니라 가장 가까운 임의의 블록에 변수를 붙인다. 따라서 이전에 var 선언문을 사용해서 작성된 코드는 함수 스코프와 숨겨진 연계가 있을 수 있으므로 코드 리팩토링을 위해서는 단순히 var를 let으로 바꾸는 것 이상의 노력이 필요하다.

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;
    if (baz > bar) {
        console.log( baz );
    }
    // …
}
```

앞의 코드는 다음과 같이 상당히 쉽게 리팩토링된다.

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;
    // ...
}

if (baz > bar) {
    console.log( baz );
}
```

그러나 블록 스코프 변수를 사용할 때는 이와 같은 수정을 하기 전에 주의해야 한다.

```
var foo = true, baz = 10;

if (foo) {
    let bar = 3;

    if (baz > bar) { // <-- don't forget `bar` when moving!
        console.log( baz );
    }
}
```

부록 B에는 대안적 (좀 더 명시적) 스타일의 블록 스코프 생성법이 나와 있는데, 어쩌면 코드를 유지보수하고 리팩토링을 더 쉽게 하는 데 도움이 될 것이다.

3.4.4 Const

키워드 let과 함께 ES6에서는 const도 추가됐다. 키워드 const 역시 블록 스코프를 생성하지만, 선언된 값은 고정된다(상수). 선언된 후 const의 값을 변경하려고 하면 오류가 발생한다.

```
var foo = true;

if (foo) {
    var a = 2;
    const b = 3; // block-scoped to the containing `if`
    a = 3; // just fine!
    b = 4; // error!
}

console.log( a ); // 3
console.log( b ); // ReferenceError!
```

3.4 복습하기

자바스크립트에서 함수는 스코프를 이루는 가장 흔한 단위다. 다른 함수 안에서 선언된 변수와 함수는 본질적으로 다른 '스코프'로부터 '숨겨진' 것이다. 이는 좋은 소프트웨어를 위해 적용해야 할 디자인 원칙이다.

그러나 함수는 결코 유일한 스코프 단위가 아니다. 블록 스코프는 함수만이 아니라 (일반적으로 { … } 같은) 임의의 코드 블록에 변수와 함수가 속하는 개념이다.

ES3부터 시작해서 try/catch 구조의 catch 부분은 블록 스코프를 가진다. ES6에서는 키워드 let(키워드 var와 비슷하다)이 추가되어 임의의 코드 블록 안에 변수를 선

언할 수 있게 되었다. “if (…) { let a = 2; }”에서 변수 a는 if 문의 { … } 블록 스코프에 자신을 붙인다.

쉽게 착각하지만, 블록 스코프는 var 함수 스코프를 완전히 대체할 수 없다. 두 기능은 공존하고, 개발자들은 함수 스코프와 블록 스코프 기술을 같이 사용할 수 있어야 하고 그래야 한다. 상황에 따라 더 읽기 쉽고 유지보수가 쉬운 코드를 작성하기 위해 두 기술을 적절한 곳에 사용하면 된다.

4 | 호이스팅

이제 스코프라는 개념에 어느 정도 익숙해졌을 것이다. 어디서 어떻게 선언되는지에 따라 변수가 다른 여러 수준의 스코프에 붙게 되는 과정도 이해했을 것이다. 함수 스코프와 블록 스코프 모두 이 점에서는 똑같은 규칙에 따라 작동한다. 한 스코프 안에서 선언된 변수는 바로 그 스코프에 속한다.

선언문이 스코프의 어디에 있는지에 따라 스코프에 변수가 추가되는 과정에 미묘한 차이가 있다. 여기서 그 차이에 대해 살펴보자.

4.1 닭이 먼저냐 달걀이 먼저냐

자바스크립트 프로그램이 실행되면 코드가 한 줄 한 줄 위에서부터 차례대로 해석될 것이라고 생각하기 쉽다. 대체로 옳은 생각이지만, 바로 이런 추정 때문에 프로그램을 잘못 이해하는 경우가 있다.

```
a = 2;

var a;

console.log( a );
```

console.log(…)의 결과로 무엇이 출력될까? 많은 개발자는 결과값이 undefined로 나오리라 예상한다. var a 선언이 a = 2 뒤에 있어서 해당 변수가 재정의되어 기본값인 undefined를 가질 것이라 추측하는 것은 자연스러운 일이다. 그러나 출력 결과는 2다.

```
console.log( a );

var a = 2;
```

이전 코드가 위에서 아래로 처리되는 방식이 아니었으니 이번 코드에서도 그런 식으로 처리되어 똑같이 2가 출력되리라 생각할 지도 모른다. 또는 a가 선언되기 전에 사용되었으니 ReferenceError가 발생한다고 생각할 수도 있다.

불행히도 둘 다 틀렸다. 출력 결과는 undefined다.

자, 대체 이 부분은 어떤 식으로 처리되는 걸까? 이것은 어쩌면 닭과 달걀의 문제처럼 보일 수 있다. 무엇이 먼저일까, 선언문(달걀)일까 아니면 대입문(닭)일까?

4.2 컴파일러는 두 번 공격한다

이 질문에 답하려면 1장으로 돌아가 컴파일러에 대해 다시 참고할 필요가 있다. 자바스크립트 엔진이 코드를 인터프리팅하기 전에 컴파일한다는 사실을 기억해보자. 컴파일레이션 단계 중에는 모든 선언문을 찾아 적절한 스코프에 연결해주는 과정이 있었다. 2장에서는 바로 이 과정이 렉시컬 스코프의 핵심이라고 배웠다.

자, 이제 변수와 함수 선언문 모두 코드가 실제 실행되기 전에 먼저 처리된다고 보면 된다. 어쩌면 "var a = 2;"를 하나의 구문이라고 생각할 수 있다. 그러나 자바스크립트는 다음 두 개의 구문으로 본다.

- `var a;`
- `a = 2;`

첫째 구문은 선언문으로 컴파일레이션 단계에서 처리된다. 둘째 구문은 대입문으로 실행 단계까지 내버려둔다.

따라서 첫 번째 코드 조각은 다음과 같이 처리된다.

```
var a;

a = 2;

console.log( a );
```

첫째 부분은 컴파일레이션 과정이고, 둘째 부분은 실행 과정이다. 비슷한 방식으로 두 번째 코드 조각은 다음과 같이 처리된다:

```
var a;

console.log( a );

a = 2;
```

이 과정을 비유적으로 말하면 변수와 함수 선언문은 선언된 위치에서 코드의 꼭대기로 '끌어올려'진다. 이렇게 선언문을 끌어올리는 동작을 '호이스팅hoisting'이라고 한다. 즉, 달걀(선언문)이 닭(대입문)보다 먼저다.

NOTE_

선언문만 끌어올려지고 다른 대입문이나 실행 로직 부분은 제자리에 그대로 둔다. 호이스팅으로 코드 실행 로직 부분이 재배치된다면 큰 혼란이 생길 수 있다.

```
foo();

function foo() {
   console.log( a ); // undefined
   var a = 2;
}
```

함수 foo의 선언문(앞의 예에서 실제 함수의 값은 포함한다)은 끌어올려졌으므로 foo를 첫째 줄에서도 호출할 수 있었다.

호이스팅이 스코프별로 작동한다는 점도 중요하다. 앞에서는 오직 글로벌 스코프만 포함된 단순한 상황을 예로 들었지만, 예제의 함수 foo(…) 내에서도 변수 a가 (명백하게도 프로그램의 꼭대기가 아니라) foo(…)의 꼭대기로 끌어올려진다. 예제 코드에 호이스팅을 적용해 좀 더 정확히 해석하면 다음과 같다.

```
function foo() {
    var a;
    console.log( a ); // undefined
    a = 2;
}

foo();
```

함수 선언문은 이와 같이 끌어올려지지만, 함수 표현식은 다르다.

```
foo(); // not ReferenceError, but TypeError!

var foo = function bar() {
    // ...
};
```

변수 확인자 foo는 끌어올려져 둘러싼 (글로벌) 스코프에 붙으므로 foo() 호출은 실패하지 않고, ReferenceError도 발생하지 않는다. 그러나 foo는 아직 값을 가지고 있지 않는데(마치 foo가 함수 표현식이 아니라 진짜 선언문으로 생성된 것처럼), foo()가 undefined 값을 호출하려 해서 TypeError라는 오작동을 발생시킨다.

또 기억할 것은 함수 표현식이 이름을 가져도 그 이름 확인자는 해당 스코프에서 찾을 수 없다는 점이다.

```
foo(); // TypeError
bar(); // ReferenceError

var foo = function bar() {
    // …
};
```

이 코드에 호이스팅을 적용하면 다음과 같이 해석된다.

```
var foo;

foo(); // TypeError
bar(); // ReferenceError

foo = function() {
    var bar = …self…
    // …
}
```

4.3 함수가 먼저다

함수와 변수 선언문은 모두 끌어올려진다. 그러나 (코드에서 여러 개의 '중복' 선언을 하면 확인할 수 있는) 미묘한 차이가 있는데, 먼저 함수가 끌어올려지고, 다음으로 변수가 올려진다.

```
foo(); // 1

var foo;

function foo() {
    console.log( 1 );
}

foo = function() {
    console.log( 2 );
};
```

결과값으로 2가 아니라 1이 출력된다! 엔진은 이 코드를 다음과 같이 해석한다.

```
function foo() {
    console.log( 1 );
}

foo(); // 1

foo = function() {
    console.log( 2 );
};
```

var foo가 중복 (그래서 무시된) 선언문이라는 점을 보자. var foo는 function foo() 선언문 보다 앞서 선언됐지만, 함수 선언문이 일반 변수 위로 끌어올려졌다. 많은 중복 변수 선언문이 사실상 무시됐지만, 중복 함수 선언문은 앞선 것들을 겹쳐 쓴다.

```
foo(); // 3

function foo() {
    console.log( 1 );
}

var foo = function() {
    console.log( 2 );
};

function foo() {
    console.log( 3 );
}
```

앞에서 살펴본 점들이 그저 흥미로운 학술적 상식에 지나지 않는 것처럼 들릴 수도 있겠지만, 이를 통해 같은 스코프 내에서의 중복 정의가 얼마나 나쁜 방식이고, 혼란스러운 결과를 내는지 잘 알 수 있다.

일반 블록 안에서 보이는 함수 선언문은 보통 둘러싼 스코프로 끌어올려지지만, 다음 코드가 보여주듯 따르지 않을 수도 있다.

```
foo(); // "b"

var a = true;
if (a) {
    function foo() { console.log("a"); }
}
else {
    function foo() { console.log("b"); }
}
```

이 동작은 맹신할 수 있는 것이 아니라 자바스크립트 차후 버전에서는 바뀔 수 있다 것이란 점을 알아야 한다. 즉, 블록 내 함수 선언은 지양하는 것이 가장 좋다.

4.4 복습하기

"var a = 2;"는 하나의 구문처럼 보이지만, 자바스크립트 엔진은 그렇게 보지 않는다. 엔진은 이를 "var a"와 "a = 2"라는 두 개의 독립된 구문으로 보고, 첫째 구문은 컴파일러 단계에서 처리하고 둘째 구문은 실행 단계에서 처리한다.

이것이 의미하는 바는 스코프의 모든 선언문은 어디서 나타나든 실행 전에 먼저 처리된다는 점이다. '호이스팅'이라 불리는 이 과정은 (변수와 함수) 선언문 각각이 속한 스코프의 꼭대기로 '끌어올려'지는 작업이라고 생각할 수 있다. 그 과정에서 선언문 자체는 옮겨지지만, 함수 표현식의 대입문을 포함한 모든 대입문은 끌어올려지지 않는다.

중복 선언을 조심하자. 일반 변수 선언과 함수 선언을 섞어 사용하면 특히 더 위험하다!

5 | 스코프 클로저

지금까지 잘 따라왔다면 스코프가 어떻게 작동하는지 확실히 이해했을 것이다.

이제 자바스크립트의 굉장히 중요하지만, 자주 잊어버리곤 해서 거의 신화적인 부분인 클로저closure를 살펴보겠다. 렉시컬 스코프에 대한 설명을 제대로 이해했다면 클로저는 앞에서 배웠던 것들보다 쉽고 번해 보일 것이다. 렉시컬 스코프에 대해 아직 미심쩍은 부분이 남아있다면 지금이 기회이니 더 읽기 전에 2장으로 돌아가서 복습해도 좋을 것이다.

5.1 깨달음

자바스크립트를 사용해봤지만 단 한 번도 클로저 개념을 완전히 이해한 적이 없는 이들이라면, 클로저가 열반에 드는 것처럼 고된 노력을 들여야 이해할 수 있는 것이라 생각할지도 모르겠다.

몇 년 전에 나는 자바스크립트에 대해서 완전히 이해하고 있었지만, 클로저가 무엇인지도 몰랐던 적이 있었다. 알고 있던 언어에 내가 모르는 더 강력한 면이 있다는 사실이 약 올리고 도발하는 것 같았다. 당시 초기 프레임워크의 소스 코드를 읽으면서 어떤 식으로 작동하는지 이해해보려고 노력했다. 처음으로 '모듈 패턴' 비슷한 것이 내 머리 속에 떠오르며 오호라! 했던 그 순간을 나는 아직도 생생하게 기억한다.

그 당시 몰랐고 이해하기까지 수년이 걸린, 지금 설명하려는 비밀은 바로 이것이다. 클로저는 자바스크립트 모든 곳에 존재한다. 그저 인식하고 받아들이면 된다. 클로저는 새롭게 문법과 패턴을 배워야 할 특별한 도구가 아니다. 루크가 포스를 수련한 것처럼 따로 익혀야 할 무기도 아니다.

클로저는 렉시컬 스코프에 의존해 코드를 작성한 결과로 그냥 발생한다. 이용하려고 굳이 의도적으로 클로저를 생성할 필요도 없다. 모든 코드에서 클로저는 생성되고 사용된다. 그러므로 여기서 적절히 클로저의 전반을 파악하면 클로저를 목적에 따라 확인하고, 받아들이고, 이용할 수 있다.

깨달음의 순간이 이럴 것이다. "아, 클로저는 내 코드 전반에서 이미 일어나고 있었구나! 이제 난 클로저를 볼 수 있어." 클로저를 이해하는 것은 네오가 매트릭스를 처음 봤을 때와 같을 것이다.

5.2 핵심

좋다, 이제 과장법과 같잖은 영화 인용은 그만하겠다. 클로저를 이해하고 파악해야 할 특징 위주로 가감 없이 정의해보자.

클로저는 함수가 속한 렉시컬 스코프를 기억하여 함수가 렉시컬 스코프 밖에서 실행될 때에도 이 스코프에 접근할 수 있게 하는 기능을 뜻한다.

코드를 보면서 앞의 정의가 설명한 바를 살펴보자.

```
function foo() {
    var a = 2;
    function bar() {
        console.log( a ); // 2
    }
    bar();
}

foo();
```

앞의 코드는 중첩 스코프를 다룰 때 보았던 예제와 비슷하다. 함수 bar()는 렉시컬

스코프 검색 규칙을 통해 바깥 스코프의 변수 a에 접근할 수 있다(이 경우는 RHS 참조 검색이다).

이것이 '클로저'인가?

음, 기술적으로 보자면 그렇게 볼 수 있다. 그러나 앞에서 말한 정의에 따르자면 꼭 그렇지는 않다. a를 참조하는 bar()를 설명하는 가장 정확한 방식은 렉시컬 스코프 검색 규칙에 따라 설명하는 것이고, 이 규칙은 클로저의 일부(이 점이 중요하다!)일 뿐이다.

순전히 학술적인 관점에서 앞의 코드에 대해 말하면 함수 bar()는 foo() 스코프에 대한 클로저를 가진다(물론 bar()는 그 외의 접근 가능한 (이 경우는 글로벌) 스코프에 대한 클로저도 가진다). 달리 말하면 bar()는 foo() 스코프에서 닫힌다. 왜 일까? bar()는 중첩되어 foo() 안에 존재하기 때문이다. 이 얼마나 쉽고 단순한가.

그러나 이런 방식으로 정의된 클로저는 바로 알아보기 힘들고 앞의 코드에서 클로저가 작동하는 방식을 볼 수도 없다. 렉시컬 스코프는 분명하게 볼 수 있지만, 클로저는 여전히 코드 뒤에 숨겨진 불가사의한 음모의 그림자로 남아있다.

그럼 클로저의 정체를 완전히 드러낼 코드를 살펴보자.

```
function foo() {
    var a = 2;
    function bar() {
        console.log( a );
    }
    return bar;
}
var baz = foo();
baz(); // 2 -- Whoa, closure was just observed, man.
```

함수 bar()는 foo()의 렉시컬 스코프에 접근할 수 있고, bar() 함수 자체를 값으로 넘긴다. 이 코드는 bar를 참조하는 함수 객체 자체를 반환한다.

foo()를 실행하여 반환한 값(bar() 함수)을 baz라 불리는 변수에 대입하고 실제로는 baz() 함수를 호출했다. 이는 당연하게도 그저 다른 확인자 참조로 내부 함수인 bar()를 호출한 것이었다. bar()는 의심할 여지 없이 실행됐다. 그러나 이 경우에 함수 bar는 함수가 선언된 렉시컬 스코프 밖에서 실행됐다.

일반적으로 foo()가 실행된 후에는 foo()의 내부 스코프가 사라졌다고 생각할 것이다. 이것은 엔진이 가비지 콜렉터garbage collector를 고용해 더는 사용하지 않는 메모리를 해제시킨다는 사실을 알기 때문이다. 더는 foo()의 내용을 사용하지 않는 상황이라면 사라졌다고 보는 게 자연스럽다.

그러나 클로저의 '마법'이 이를 내버려두지 않는다. 사실 foo의 내부 스코프는 여전히 '사용 중'이므로 해제되지 않는다. 그럼 누가 그 스코프를 사용 중인가? 바로 bar() 자신이다. 선언된 위치 덕에 bar()는 foo() 스코프에 대한 렉시컬 스코프 클로저를 가지고, foo()는 bar()가 나중에 참조할 수 있도록 스코프를 살려둔다. 즉, bar()는 여전히 해당 스코프에 대한 참조를 가지는데, 그 참조를 바로 클로저라고 부른다.

foo() 선언이 끝나고 수 밀리초 후 변수 baz를 호출(bar라 명명했던 내부 함수를 호출)할 때, 해당 함수는 원래 코드의 렉시컬 스코프에 접근할 수 있고 예상한 것처럼 이는 함수가 변수 a에 접근할 수 있다는 의미다.

함수는 원래 코드의 렉시컬 스코프에서 완전히 벗어나 호출됐다. 클로저는 호출된 함수가 원래 선언된 렉시컬 스코프에 계속해서 접근할 수 있도록 허용한다. 물론, 어떤 방식이든 함수를 값으로 넘겨 다른 위치에서 호출하는 행위는 모두 클로저가 작용한 예다.

```
function foo() {
    var a = 2;
    function baz() {
        console.log( a ); // 2
    }
    bar( baz );
}

function bar(fn) {
    fn(); // look ma, I saw closure!
}
```

코드에서 함수 baz를 bar에 넘기고, 이제 fn이라 명명된 함수를 호출했다. 이때 foo()의 내부 스코프에 대한 fn의 클로저는 변수 a에 접근할 때 확인할 수 있다. 이런 함수 넘기기는 간접적인 방식으로도 가능하다.

```
var fn;

function foo() {
    var a = 2;
    function baz() {
        console.log( a );
    }
    fn = baz; // assign baz to global variable
}

function bar() {
    fn(); // look ma, I saw closure!
}

foo();
bar(); // 2
```

어떤 방식으로 내부 함수를 자신이 속한 렉시컬 스코프 밖으로 수송하든 함수는 처음 선언된 곳의 스코프에 대한 참조를 유지한다. 즉, 어디에서 해당 함수를 실행하든 클로저가 작용한다.

5.3 이제 나는 볼 수 있다

앞에서 본 코드들은 클로저 사용법을 보여주기 위해 다소 학술적이고 인위적으로 작성했다. 나는 클로저가 단순히 멋진 새 장난감 이상의 것이라고 말했다. 또 클로저가 모든 코드 안에 존재하는 무언가라고도 이야기했다. 이제 왜 그것이 사실인지 살펴보자.

```
function wait(message) {
    setTimeout( function timer(){
        console.log( message );
    }, 1000 );
}
wait( "Hello, closure!" );
```

내부 함수 timer를 setTimeout(…)에 인자로 넘겼다. timer 함수는 wait(…) 함수의 스코프에 대한 스코프 클로저를 가지고 있으므로 변수 message에 대한 참조를 유지하고 사용할 수 있다.

wait(…) 실행 1초 후, wait의 내부 스코프는 사라져야 하지만 익명의 함수가 여전히 해당 스코프에 대한 클로저를 가지고 있다.

엔진의 내부 깊숙한 곳에 내장 함수 setTimeout(…)에는 아마도 fn이나 func 정도로 불릴 매개변수의 참조가 존재한다. 엔진은 그 함수 참조를 호출하여 내장 함수 timer를 호출하므로 timer의 렉시컬 스코프는 여전히 온전하게 남아 있다.

클로저

jQuery (또는 다른 자바스크립트 프레임워크) 신봉자라면 다음을 보자.

```
function setupBot(name,selector) {
    $( selector ).click( function activator(){
        console.log( "Activating: " + name );
    } );
}

setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

여러분이 어떤 종류의 코드를 작성하는지 모르겠지만, 나는 자주 클로저 봇으로 이뤄진 글로벌 드론 부대 전체를 통제하는 코드를 작성하므로 앞의 코드는 완전히 현실적이다!

농담은 접어두고, 자체의 렉시컬 스코프에 접근할 수 있는 함수를 인자로 넘길 때 그 함수가 클로저를 사용하는 것을 볼 수 있다. 타이머, 이벤트 처리기, Ajax 요청, 윈도우 간 통신, 웹 워커와 같은 비동기적(또는 동기적!) 작업을 하며 콜백 함수를 넘기면 클로저를 사용할 준비가 된 것이다!

> **NOTE_**
>
> 3장에서 IIFE 패턴에 대해 살펴봤다. 흔히 IIFE 자체가 드러난 클로저의 예라고도 하는데, 나는 동의하지 않는다. IIFE만으로는 앞에서 정의한 요소를 갖추지 못하기 때문이다.

```
var a = 2;

(function IIFE(){
    console.log( a );
})();
```

이 코드는 '작동'하지만 엄격히 말해 클로저가 사용된 것은 아니다. 'IIFE' 함수가 자신의 렉시컬 스코프 밖에서 실행된 것이 아니기 때문이다. IIFE 함수는 선언된 바로 그곳의 스코프 안에서 호출됐다(a를 들고 있는 둘러싼 스코프와 글로벌 스코프). 변수 a는 클로저가 아니라 일반적인 렉시컬 스코프 검색을 통해 가져왔다.

클로저는 기술적으로 보면 선언할 때 발생하지만, 바로 관찰할 수 있는 것은 아니다. 누군가 말했던 것처럼 아무도 듣는 사람 없이 숲 속의 쓰러진 나무와 같다.[01]

IIFE 자체는 클로저의 사례가 아니지만, IIFE는 틀림없이 스코프를 생성하고, 클로저를 사용할 수 있는 스코프를 만드는 가장 흔한 도구 중 하나다. 따라서 IIFE 자체가 클로저를 작동시키지는 않아도 확실히 클로저와 연관이 깊다.

잠시 이 책을 덮어보라. 한 가지 과제를 주겠다. 가장 최근에 작성한 자바스크립트 코드를 열어보라. 함수를 값으로 사용한 경우를 찾아보고, 알지 못했지만 이미 클로저를 사용하고 있는 부분을 확인해보자.

기다리겠다.

자… 이제 보이는가!

01 역자주_ 숲 속에서 나무가 쓰러지는데, 아무도 그 소리를 듣지 못한다면 쓰러지는 나무가 소리를 낸다고 말할 수 있는가? 존재하는 것은 지각되는 것이다.(철학자 비숍 조지 버클리의 말)

5.4 반복문과 클로저

클로저를 설명하는 가장 흔하고 표준적인 사례는 for 반복문이다.

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

NOTE_

린터는 반복문 안에 함수를 넣을 경우 종종 경고한다. 이렇게 설정한 이유는 개발자들이 클로저를 자주 사용한다고 생각하지 않기 때문이다. 이 책에서는 올바른 방식으로 어떻게 클로저를 사용하면 그 힘을 최대한 끌어낼 수 있는지 설명한다. 그러나 린터는 이 미묘한 차이를 구별하지 못하고 개발자는 무엇을 하고 있는지 모른다고 취급하며 그저 경고를 내보낸다.

이 코드의 목적은 예상대로 '1', '2', … , '5'까지 한 번에 하나씩 일 초마다 출력하는 것이다. 그러나 실제로 코드를 돌려보면, 일 초마다 한 번씩 '6'만 5번 출력된다.

뭐?

먼저, 6이 어떻게 나오는지 알아보자. 반복문이 끝나는 조건은 i가 '<=5'가 아닐 때다. 처음으로 끝나는 조건이 갖춰졌을 때 i의 값은 6이다. 즉, 출력된 값은 반복문이 끝났을 때의 i 값을 반영한 것이다. 코드를 다시 보면 이 설명이 당연하게 느껴질 것이다. timeout 함수 콜백은 반복문이 끝나고 나서야 작동한다. 사실, 타이머를 차치하고 반복마다 실행된 것이 setTimeout(…, 0)이었다 해도 해당 함수 콜백은 확실히 반복문이 끝나고 나면 동작해서 결과로 매번 6을 출력한다.

여기서 더 심오한 문제가 제기된다. 애초에 문법적으로 기대한 것과 같이 이 코드를 작동시키려면 무엇이 더 필요할까?

그러기 위해 필요한 것은 반복마다 각각의 i 복제본을 '잡아'두는 것이다. 그러나 반복문 안 총 5개의 함수들은 반복마다 따로 정의되었음에도 모두 같이 글로벌 스코프 클로저를 공유하여 해당 스코프 안에는 오직 하나의 i만이 존재한다. 따라서 모든 함수는 당연하게도 같은 i에 대한 참조를 공유한다. 그냥 5개의 timeout 콜백을 쭉 이어서 반복문 없이 선언해도 결과는 똑같다.

자, 이제 다시 질문으로 돌아가보자. 무엇이 더 필요한가? 필요한 것은 더 많은 닫힌closured 스코프다. 구체적으로 말하면 반복마다 하나의 새로운 닫힌 스코프가 필요하다.

3장에서 IIFE가 함수를 정의하고 바로 실행시키면서 스코프를 생성한다고 배웠다. 시도해보자.

```
for (var i=1; i<=5; i++) {
    (function(){
        setTimeout( function timer(){
            console.log( i );
        }, i*1000 );
    })();
}
```

작동하는가? 시도해보라. 이번에도 기다리겠다.

결과를 기다리는 초조함은 내가 끝내주겠다. 결과는 작동하지 않는다. 왜 그럴까? 이제 분명 더 많은 렉시컬 스코프를 가지는데 말이다. 각각의 timeout 함수 콜백은 확실히 반복마다 각각의 IIFE가 생성한 자신만의 스코프를 가진다. 그러나 닫힌

스코프만으로는 부족하다. 이 스코프가 비어있기 때문이다.

자세히 살펴보자. IIFE는 아무 것도 하지 않는 빈 스코프일 뿐이니 무언가 해야 한다. 각 스코프는 자체 변수가 필요하다. 즉, 반복마다 i의 값을 저장할 변수가 필요한 것이다.

```
for (var i=1; i<=5; i++) {
    (function(){
        var j = i;
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })();
}
```

유레카! 성공했다!

약간 다른 버전을 선호하는 이들도 있을 것이다.

```
for (var i=1; i<=5; i++) {
    (function(j){
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })( i );
}
```

물론, 이 IIFE는 함수니까 i를 넘길 때 매개변수의 이름은 j로 할 수 있고 다시 i로 할 수도 있다. 어떻게 해도 코드는 잘 작동한다.

IIFE를 사용하여 반복마다 새로운 스코프를 생성하는 방식으로 timeout 함수 콜백은 원하는 값이 제대로 저장된 변수를 가진 새 닫힌 스코프를 반복마다 생성해 사용할 수 있다.

문제는 해결됐다!

5.4.1 다시 보는 블록 스코프

이전 문제의 해법을 잘 살펴보자. 반복마다 IIFE를 사용해 하나의 새로운 스코프를 생성했다. 다시 말하면, 실제 필요했던 것은 반복별 블록 스코프였다. 3장에서 블록을 이용해 해당 블록 스코프에 변수를 선언하는 let 선언문을 배웠다. 키워드 let은 본질적으로 하나의 블록을 닫을 수 있는 스코프로 바꾼다.

자, 다음의 멋진 코드가 어떻게 잘 작동하는지 보자.

```
for (var i=1; i<=5; i++) {
    let j = i; // yay, block-scope for closure!
    setTimeout( function timer(){
        console.log( j );
    }, j*1000 );
}
```

그러나 그게 다가 아니다! let 선언문이 for 반복문 안에서 사용되면 특별한 방식으로 작동한다. 반복문 시작 부분에서 let으로 선언된 변수는 한 번만 선언되는 것이 아니라 반복할 때마다 선언된다. 따라서 해당 변수는 편리하게도 반복마다 이전 반복이 끝난 이후의 값으로 초기화된다.

```
for (let i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

멋지지 않은가? 블록 스코프와 클로저가 함께 활약해서 모든 문제를 해결했다. 여러분은 어떨지 모르지만, 나는 자바스크립트 사용자로서 기쁘다.

5.5 모듈

클로저의 능력을 활용하면서 표면적으로는 콜백과 상관없는 코드 패턴들이 있다. 그 중 가장 강력한 패턴인 모듈을 살펴보자.

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];
    function doSomething() {
        console.log( something );
    }
    function doAnother() {
        console.log( another.join( " ! " ) );
    }
}
```

이 코드에는 클로저의 흔적이 보이지 않는다. 우리가 볼 수 있는 것은 몇 가지 비공개 데이터 변수인 something과 another 그리고 내부 함수 doSomething()과 doAnother()가 있다. 이들 모두 foo()의 내부 스코프를 렉시컬 스코프(당연히 클로저도 따라온다)로 가진다.

```
function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];
    function doSomething() {
        console.log( something );
    }
    function doAnother() {
        console.log( another.join( " ! " ) );
    }
    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}

var foo = CoolModule();
foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

이 코드와 같은 자바스크립트 패턴을 모듈이라고 부른다. 가장 흔한 모듈 패턴 구현 방법은 모듈 노출revealing module이고, 앞의 코드는 이것의 변형이다.

먼저, 앞의 코드에서 몇 가지를 살펴보자.

첫째, CoolModule()은 그저 하나의 함수일 뿐이지만, 모듈 인스턴스를 생성하려면 반드시 호출해야 한다. 최외각 함수가 실행되지 않으면 내부 스코프와 클로저는 생성되지 않는다.

둘째, CoolModule() 함수는 객체를 반환한다. 반환되는 객체는 객체-리터럴 문법 { key: value, … }에 따라 표기된다. 해당 객체는 내장 함수들에 대한 참조를 가지지만, 내장 데이터 변수에 대한 참조는 가지지 않는다. 내장 데이터 변수는 비공개로

숨겨져 있다. 이 객체의 반환값은 본질적으로 모듈의 공개 API라고 생각할 수 있다.

객체의 반환값은 최종적으로 외부 변수 foo에 대입되고, foo.doSomething()과 같은 방식으로 API의 속성 메소드에 접근할 수 있다.

> **NOTE_**
>
> 모듈에서 꼭 실제 객체(리터럴)를 반환할 필요 없이 직접 내부 함수를 반환해도 된다. jQuery가 이런 반환을 하는 좋은 사례다. jQuery와 $ 확인자는 jQuery '모듈'의 공개 API고, 동시에 그 자신들은 단순한 함수이기도 하다(모든 함수는 객체이므로 속성을 가질 수 있다).

함수 doSomething()과 doAnother()는 (CoolModule()을 호출하면 얻을 수 있는) 모듈 인스턴스의 내부 스코프에 포함하는 클로저를 가진다. 반환된 객체에 대한 속성 참조 방식으로 이 함수들을 해당 렉시컬 스코프 밖으로 옮길 때 클로저를 확인하고 이용할 수 있는 조건을 하나 세웠다.

쉽게 말해, 이 모듈 패턴을 사용하려면 두 가지 조건이 있다.

① 하나의 최외각 함수가 존재하고, 이 함수가 최소 한 번은 호출되어야 한다(호출할 때마다 새로운 모듈 인스턴스가 생성된다).
② 최외각 함수는 최소 한 번은 하나의 내부 함수를 반환해야 한다. 그래야 해당 내부 함수가 비공개 스코프에 대한 클로저를 가져 비공개 상태에 접근하고 수정할 수 있다.

하나의 함수 속성만을 가지는 객체는 진정한 모듈이 아니다. 함수 실행 결과로 반환된 객체에 데이터 속성들은 있지만 닫힌 함수가 없다면, 당연히 그 객체는 진정한 모듈이 아니다.

앞의 코드는 독립된 모듈 생성자 CoolModule()을 가지고, 생성자는 몇 번이든 호

출할 수 있고 호출할 때마다 새로운 모듈 인스턴스를 생성한다. 이 패턴에서 약간 변경된 오직 하나의 인스턴스, '싱글톤'만 생성하는 모듈을 살펴보자.

```
var foo = (function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];
    function doSomething() {
        console.log( something );
    }
    function doAnother() {
        console.log( another.join( " ! " ) );
    }
    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
})();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

앞의 코드에서 모듈 함수를 IIFE(3장 참조)로 바꾸고 즉시 실행시켜 반환값을 직접 하나의 모듈 인스턴스 확인자 foo에 대입시켰다.

모듈은 함수이므로 다음 코드처럼 매개변수를 받을 수 있다.

```
function CoolModule(id) {
    function identify() {
        console.log( id );
    }
```

```
    return {
        identify: identify
    };
}

var foo1 = CoolModule( "foo 1" );
var foo2 = CoolModule( "foo 2" );

foo1.identify(); // "foo 1"
foo2.identify(); // "foo 2"
```

약간 변형한 효과적인 모듈 패턴 중 또 하나는 다음 코드와 같이 공개 API로 반환하는 객체에 이름을 정하는 방식이다.

```
var foo = (function CoolModule(id) {
    function change() {
        // modifying the public API
        publicAPI.identify = identify2;
    }
    function identify1() {
        console.log( id );
    }
    function identify2() {
        console.log( id.toUpperCase() );
    }
    var publicAPI = {
        change: change,
        identify: identify1
    };
    return publicAPI;
```

```
})( "foo module" );

foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE
```

공개 API 객체에 대한 내부 참조를 모듈 인스턴스 내부에 유지하면, 모듈 인스턴스를 내부에서부터 메소드와 속성을 추가 또는 삭제하거나 값을 변경하는 식으로 수정할 수 있다.

5.5.1 현재의 모듈

많은 모듈 의존성 로더와 관리자는 본질적으로 이 패턴의 모듈 정의를 친숙한 API 형태로 감싸고 있다. 특정한 하나의 라이브러리를 살펴보기보다는 개념을 설명하기 위해 매우 단순한 증명을 제시하겠다.

```
var MyModules = (function Manager() {
    var modules = {};

    function define(name, deps, impl) {
        for (var i=0; i<deps.length; i++) {
            deps[i] = modules[deps[i]];
        }
        modules[name] = impl.apply( impl, deps );
    }

    function get(name) {
        return modules[name];
    }
```

```
        return {
            define: define,
            get: get
        };
    })();
```

이 코드의 핵심부는 "modules[name] = impl.apply(impl, deps)"다. 이 부분은 (의존성을 인자로 넘겨) 모듈에 대한 정의 래퍼 함수를 호출하여 반환값인 모듈 API를 이름으로 정리된 내부 모듈 리스트에 저장한다.

해당 부분(modules[name] = impl.apply(impl, deps))을 이용해 모듈을 정의하는 다음 코드를 보자.

```
MyModules.define( "bar", [], function(){
    function hello(who) {
        return "Let me introduce: " + who;
    }
    return {
        hello: hello
    };
} );

MyModules.define( "foo", ["bar"], function(bar){
    var hungry = "hippo";
    function awesome() {
        console.log( bar.hello( hungry ).toUpperCase() );
    }
    return {
        awesome: awesome
    };
```

```
} );

var bar = MyModules.get( "bar" );
var foo = MyModules.get( "foo" );

console.log(
    bar.hello( "hippo" )
); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

'foo'와 'bar' 모듈은 모두 공개 API를 반환하는 함수로 정의됐다. 'foo'는 심지어 'bar'의 인스턴스를 의존성 매개변수로 받아 사용할 수도 있다.

찬찬히 코드를 살펴보면 목적에 따라 사용된 클로저의 힘을 완전히 이해할 수 있다. 모듈 관리자를 만드는 특별한 마법이란 존재하지 않는다는 것을 기억해야 한다. 모든 모듈 관리자는 앞에서 언급한 모듈 패턴의 특성을 모두 가진다. 즉, 이들은 함수 정의 래퍼를 호출하여 해당 모듈의 API인 반환값을 저장한다. 좀 더 쓰기 편하게 포장한다고 해도 모듈은 그저 모듈일 뿐이다.

5.5.2 미래의 모듈

ES6는 모듈 개념을 지원하는 최신 문법을 추가했다. 모듈 시스템이 불러올 때 ES6는 파일을 개별의 모듈로 처리한다. 각 모듈은 다른 모듈 또는 특정 API 멤버를 불러오거나 자신의 공개 API 멤버를 내보낼 수도 있다.

NOTE_

함수 기반 모듈은 정적으로 알려진 (컴파일러가 읽을 수 있는) 패턴이 아니다. 따라서 이들 API의 의미는 런타임 전까지 해석되지 않는다. 즉, 실제로 모듈의 API를 런타임에 수정할 수 있다는 말이다(앞의 publicAPI에 대한 설명 참조).

반면, ES6 모듈 API는 정적이다(API가 런타임에 변하지 않는다). 따라서 컴파일러는 이 사실을 이미 알고 있어서 (파일 불러오는 때와) 컴파일레이션 중에 불러온 모듈의 API 멤버 참조가 실제로 존재하는지 확인할 수 있다(실제로 확인한다!). API 참조가 존재하지 않으면, 컴파일러는 컴파일 시 초기 오류를 발생시킨다. 전통적인 방식처럼 변수 참조(그리고 있다면 오류 발생도)를 위해 동적 런타임까지 기다리지 않는다.

ES6 모듈은 inline 형식을 지원하지 않고, 반드시 개별 파일(모듈당 파일 하나)에 정의되어야 한다. 브라우저와 엔진은 기본 모듈 로더(오버라이딩할 수 있다. 하지만 이에 대한 논의는 책의 범위를 넘어선다)를 가진다. 모듈을 불러올 때 모듈 로더는 동기적으로 모듈 파일을 불러온다.

```
// bar.js
    function hello(who) {
        return "Let me introduce: " + who;
    }

    export hello;

// foo.js: import only `hello()` from the "bar" module
    import hello from "bar";
    var hungry = "hippo";

    function awesome() {
        console.log(
            hello( hungry ).toUpperCase()
```

```
        );
    }

    export awesome;

// baz.js: import the entire "foo" and "bar" modules
    module foo from "foo";
    module bar from "bar";

    console.log(
        bar.hello( "rhino" )
    ); // Let me introduce: rhino

    foo.awesome(); // LET ME INTRODUCE: HIPPO
```

NOTE_

코드의 처음 두 부분을 가지고 해당 내용이 포함된 'foo.js'와 'bar.js' 파일이 각각 생성된다. 그러면 세번 째 부분 'baz.js' 프로그램이 이들 모듈을 불러와 사용한다.

키워드 import는 모듈 API에서 하나 이상의 멤버를 불러와 특정 변수(여기서는 hello)에 묶어 현재 스코프에 저장한다. 키워드 module은 모듈 API 전체를 불러와 특정 변수(여기서는 foo와 bar)에 묶는다. 키워드 export는 (변수와 함수) 확인자를 현재 모듈의 공개 API로 내보낸다. 이 연산자들은 모듈의 정의에 따라 필요하면 얼마든지 사용할 수 있다.

앞서 살펴본 함수-클로저 모듈처럼 모듈 파일의 내용은 스코프 클로저에 감싸진 것으로 처리된다.

5.6 복습하기

편견에 찬 이들은 클로저를 자바스크립트의 세계에서 홀로 떨어진, 가장 용감한 소수만이 닿을 수 있는 신비의 세계로 생각하는 것 같다. 그러나 클로저는 사실 표준이고, 함수를 값으로 마음대로 넘길 수 있는 렉시컬 스코프 환경에서 코드를 작성하는 방법이다.

클로저는 함수를 렉시컬 스코프 밖에서 호출해도 함수는 자신의 렉시컬 스코프를 기억하고 접근할 수 있는 특성을 의미한다.

반복문을 예로 들면, 클로저를 통해 설사 우리가 기억하지 못했을지라도 반복문이 어떻게 작동하는지 추적해갈 수 있다. 또한, 클로저는 다양한 형태의 모듈 패턴을 가능하게 하는 매우 효과적인 도구이기도 하다.

모듈은 두 가지 특징을 가져야 한다.

① 최외각 래퍼 함수를 호출하여 외각 스코프를 생성한다.
② 래핑 함수의 반환값은 반드시 하나 이상의 내부 함수 참조를 가져야 하고, 그 내부 함수는 래퍼의 비공개 내부 스코프에 대한 클로저를 가져야 한다.

이제 우리는 모든 코드에서 클로저를 볼 수 있고, 파악하고 활용할 수 있는 능력이 생겼다.

부록 A | 동적 스코프

2장에서 자바스크립트의 (그리고 많은 다른 언어가 사용하는) 렉시컬 스코프 모델과 비교하여 동적 스코프에 대해 언급했다.

간단하게 동적 스코프에 대해 좀 더 명료하게 살펴보겠다. 그러나 더 중요한 것은 동적 스코프가 자바스크립트의 또 다른 메커니즘(this)과 가까운 친척 관계라는 점이다. this에 관해서는 『You Don't Know JS: this와 객체 프로토타입』(한빛미디어, 출간 예정)에서 다룬다.

2장에서 보았듯이 렉시컬 스코프는 엔진이 변수를 찾는 검색 방식과 위치에 대한 규칙이다. 렉시컬 스코프의 주요 특성은 이 스코프가 프로그래머가 코드를 작성할 때 결정된다는 것이다 (eval()이나 with를 통해 스코프를 속이지 않는다는 가정 하에서 그렇다).

동적 스코프라는 말을 들으면 스코프를 코드 작성 때가 아니라 런타임 때에 동적으로 결정하는 모델이 존재하겠구나 하고 예상할 수 있다. 분명 그런 경우도 존재한다. 다음 코드를 통해 어떤 경우가 있는지 살펴보자.

```
function foo() {
    console.log( a ); // 2
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;
bar();
```

foo() 안에서 a에 대한 RHS 참고를 했을 때 결과값 2는 렉시컬 스코프를 이용해 글로벌 변수 a에서 가져온다. 하지만 동적 스코프는 함수와 스코프가 어떻게 어디서 선언됐는지는 상관없고, 오직 어디서 호출됐는지와 연관된다. 달리 말해, 동적 스코프 체인은 코드 내 스코프의 중첩이 아니라 콜-스택call-stack과 관련 있다.

따라서 자바스크립트가 동적 스코프를 사용한다면, foo()가 실행됐을 때 이론적으로 다음 코드의 출력 결과가 2가 아니라 3이 될 것이다.

```
function foo() {
    console.log( a ); // 3 (not 2!)
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

어떻게 이런 결과가 나오는 걸까? foo()는 a에 대한 변수 참조에 결과를 줄 수 없고, 엔진은 중첩 (렉시컬) 스코프 체인을 거슬러 오르는 대신 콜-스택을 거슬러 올라 foo()가 어느 시점에서 호출됐는지 찾는다. foo() 는 bar() 안에서 호출되었으므로 엔진은 bar()의 스코프에서 해당 변수를 찾고 값 3을 가지는 변수 a를 그곳에서 찾는다.

이상한가? 지금은 그렇게 생각할 수도 있다.

이것은 아마도 여러분이 렉시컬 스코프를 사용하는 (아니면 그렇게 생각되는) 코드만 작성했기 때문에 동적 스코핑이 낯설게 느껴질 수 있다. 여러분이 동적 스코프만으로 코드를 작성해 왔다면 동적 스코프가 자연스럽고 렉시컬 스코프는 이상하게 느껴질 것이다.

정확하게 말하면 자바스크립트는 동적 스코프를 사용하지 않고, 렉시컬 스코프만 사용한다. 단, this 메커니즘이 동적 스코프와 비슷한 면이 있다.

주요 차이점은 다음과 같다.

- 렉시컬 스코프는 작성할 때, 동적 스코프 (그리고 this!)는 런타임에 결정된다.
- 렉시컬 스코프는 어디서 함수가 선언됐는지와 관련 있지만, 동적 스코프는 어디서 함수가 호출됐는지와 관련 있다.

this는 함수가 어디서 호출됐는지와 관련 있는데, 이점은 this 메커니즘이 동적 스코핑 개념과 상당한 연관이 있다는 것을 보여준다. 좀 더 this에 대해 알아보고 싶다면 『You Don't Know JS: this와 객체 프로토타입』(한빛미디어, 출간 예정)을 읽어보라.

부록 B | 폴리필링 블록 스코프

3장에서 블록 스코프에 대해 배웠다. 블록 스코프의 사례로 with와 catch 문이 있다는 것을 보았고, 자바스크립트가 적어도 ES3부터 블록 스코프를 지원했다는 것도 배웠다.

ES6에서 키워드 let을 지원하면서 마침내 자바스크립트도 완전히 제한 없는 블록 스코핑을 지원하게 됐다. 블록 스코프를 통해 기능과 코드 스타일 면에서 많은 멋진 것들이 가능해진다.

그런데 ES6 이전 환경에서 블록 스코프를 사용하고 싶다면 어떻게 할까?

```
{
    let a = 2;
    console.log( a ); // 2
}

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

앞의 코드는 ES6 환경에서 잘 작동한다. ES6 이전 환경에서도 작동하게 할 수 있을까? 해답은 catch에 있다.

```
try{throw 2}catch(a){
    console.log( a ); // 2
}

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

우웩! 앞의 코드는 정말 못생기고 이상한 코드다. try/catch문은 강제로 오류를 발생시키는 것처럼 보이는데, 이 오류의 값은 항상 2고, 이 값을 받는 변수 선언문은 'catch(a)' 문에 있다. 돌아버리겠다.

그렇다, catch 문은 블록 스코핑을 사용한다. 그 말은 catch 문이 ES6 이전 환경에서도 블록 스코프 역할을 할 수 있다는 얘기다.

이렇게 말할지도 모르겠다. "그렇지만 아무도 그렇게 못생긴 코드를 작성하고 싶지 않아!" 사실이다. 커피스크립트 컴파일러의 결과물[01]도 사람이 작성하는 것은 아니다. 중요한 것은 그게 아니다. 중요한 것은 ES6 코드를 ES6 이전 환경에서 작동하도록 변형하여 컴파일해 주는 도구가 있다는 점이다. 블록 스코핑을 사용해 코드를 작성하고, 빌드 단계 도구로 실제 작동하는 코드를 생성해 사용하면 된다.

이 방식은 사실 모든 (흠… 대다수) ES6 환경으로 넘어갈 때 권장하는 방식이다. 즉, ES6 이전 환경에서 ES6로 이행하는 기간 동안 코드 변형 컴파일러를 통해 ES6 코드를 ES5 호환 코드로 변경해서 사용하는 것이다.

B.1 Traceur

구글은 Traceur[02]라는 프로젝트를 진행하고 있다. 이 프로젝트의 목적은 ES6 특성을 변형 컴파일해서 ES6 이전 환경에서 이용하게 하는 것이다(대부분 ES5까지 지원하지만, 모두가 ES5만 지원하는 것은 아니다!). TC39 위원회는 이 도구(와 다른 것을 같이) 사용하여 그들이 명시한 기능이 어떤 작용을 하는지 테스트한다.

Traceur는 앞의 코드로 어떤 코드를 생성할까? 상상해 보라!

---

01 역자주_자바스크립트를 보조하는 스크립트 컴파일러로, 결과물로 복잡한 자바스크립트 코드가 나온다.
02 http://google.github.io/traceur-compiler/demo/repl.html

```
{
    try {
        throw undefined;
    } catch (a) {
        a = 2;
        console.log( a );
    }
}

console.log( a );
```

대상이 ES6 환경이든 아니든 이런 도구를 사용하면 블록 스코프를 활용할 수 있다. try/catch는 ES3부터 존재했기 때문이다(그리고 계속 이런 방식으로 작동했다).

B.1 암시적 블록 vs 명시적 블록

3장에서 블록 스코핑을 살펴보면서 블록 스코프가 코드를 유지보수하고 리팩토링하는 데 잠재적 위험요소가 될 수 있다는 점을 확인했다. 블록 스코프를 활용하면서 단점을 최소화할 수 있는 다른 방법은 없을까?

(이전에 살펴본 let 선언문 비교하여) let 블록 또는 let 구문이라 불리는 대안적인 let 활용법을 살펴보자.

```
let (a = 2) {
    console.log( a ); // 2
}

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

암시적으로 기존의 블록을 가져오는 대신 let 구문은 자신의 스코프에 해당하는 명시적인 블록을 생성한다. 명시적 블록은 알아보기 쉽고 리팩토링하기도 더 쉽다. let 구문은 문법적으로 모든 선언문을 블록 상부에 위치하도록 강제하여 좀 더 깔끔한 코드를 생성한다. 이 덕분에 구문이 어떤 블록에 있더라도 쉽게 찾을 수 있고 스코프 내에 어떤 변수(또는 함수)가 있는지 알기도 쉽다.

하나의 패턴으로서 let 구문은 모든 변수 선언문을 수동으로 함수 꼭대기로 끌어올리는(hoist) 함수-스코핑에서 많이 사용된 접근법을 반영한다. let 구문을 사용할 때는 의식적으로 블록 상단에 선언문을 놓아야 하므로 let 선언문을 여기 저기 흩어놓지만 않는다면 블록 스코프에서 선언문을 확인하고 유지보수하는 것이 함수 스코프에서 하는 것보다 좀 더 쉽다.

문제가 하나 있는데, let 구문 형태를 ES6에서 지원하지 않는다는 것이다. 또 공식 Traceur 컴파일러도 해당 형태의 코드를 인정하지 않는다.

두 가지 해결 방법이 있다. 먼저, ES6에서 유효한 문법을 사용해 서식을 만들고 그에 따라 코딩하는 습관을 들이는 것이다.

```
/*let*/ { let a = 2;
    console.log( a );
}

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

다른 하나의 해결 방법은 명시적인 let 구문 블록을 사용하고 도구를 이용하여 해당 코드를 유효한 코드로 전환하는 것이다.

나는 let-er[03]라는 도구를 만들어 이 문제를 해결했다. let-er는 빌드 단계 변경 컴파일러로, let 구문 형태를 찾아 변형하는 일을 한다. 이 도구는 let 선언문을 포함해 다른 형태의 코드는 그대로 내버려 둔다. 따라서 ES6 변형 컴파일러를 사용할 때 let-er를 먼저 사용해도 안전하다. 필요하다면 변형된 결과 코드에 Traceur 같은 도구를 사용하면 된다.

let-er는 설정 플래그 `--es6`를 지원하는데, 해당 플러그가 켜지면(기본 꺼짐) 생성 코드를 변경한다. let-er는 try/catch 같은 ES3 폴리필링Polyfilling 꼼수를 사용하지 않고도 코드를 완전한 ES6 규약 준수 코드로 변경한다.

```
{
    let a = 2;
    console.log( a );
}

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

let-er를 사용해서 모든 ES6 이전 환경에 대비할 수 있고, ES6만 신경 쓰면 될 때는 해당 플래그를 추가하면 바로 ES6 호환 코드를 생성할 수 있다.

가장 중요한 점은 (아직은) 공식 ES 문법이 아니지만, 더 유용하고 더 명시적인 let 구문 형태를 사용할 수 있다는 것이다.

B.3 성능

try/catch의 성능에 대해 마지막으로 짧게 말하고 "왜 그냥 IIFE 사용해서 스코프를 생성하지 않는가?"라는 질문에 답하겠다.

03 https://github.com/getify/let-er

첫째, 현재는 try/catch의 성능이 더 느리다. 물론 앞으로도 이 구문이 더 느릴지 언제까지 이 상태일지 추정하기는 어렵다. TC39가 공식적으로 인정한 ES6 변형 컴파일러가 try/catch를 사용하고, Traceur 팀은 크롬 브라우저에서 try/catch의 성능을 향상시켜달라고 요청했고 크롬 팀은 분명히 노력할 의지가 있어 보인다.

둘째, IIFE는 try/catch와 1:1로 비교할 수 있는 대상이 아니다. 특정 코드를 함수로 감싸면 내부 코드 속의 this, return, break, continue 같은 명령의 의미가 달라진다. 따라서 IIFE는 일반적인 대체물이 될 수 없다. IIFE는 특정 상황에서 오직 수동으로 수정해 사용해야 한다.

이제 질문은 블록 스코핑을 원하는가 아닌가로 바뀐다. 원한다면 도구를 이용하고 아니라면 계속 var를 사용해서 코딩하면 된다!

부록 C | 렉시컬 this

제목은 this 메커니즘과 크게 상관없지만, ES6에서는 this와 렉시컬 스코프를 중요한 방식으로 연결한다. 빠르게 살펴보자.

ES6는 '화살표 함수'라는 특별한 함수 선언문 문법을 추가했다. 문법은 다음과 같다.

```
var foo = a => {
    console.log( a );
};

foo( 2 ); // 2
```

소위 '뚱뚱한 화살fat arrow'(=>)은 따분하도록 장황한 function 키워드의 줄임말로 흔히 언급된다.

화살표 함수에는 그저 선언할 때 키보드 몇 번 덜 치는 것보다 훨씬 더 중요한 요소가 있다. 간단히 문제가 있는 다음 코드를 살펴보자.

```
var obj = {
    id: "awesome",
    cool: function coolFn() {
        console.log( this.id );
    }
};

var id = "not awesome"
obj.cool(); // awesome
setTimeout( obj.cool, 100 ); // not awesome
```

앞 코드의 문제는 cool( ) 함수에 묶인 this가 사라진다는 점이다. 이 문제를 해결할 방법은 여러 가지가 있지만, 흔히 사용되는 해법은 "var self = this;"다. 적용하면 다음과 같다.

```
var obj = {
    count: 0,
    cool: function coolFn() {
        var self = this;

        if (self.count < 1) {
            setTimeout( function timer(){
                self.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // awesome?
```

헤맬 것 없이 "var self = this;" 해법은 this를 이해하고 적절히 사용하는 문제를 제쳐놓고, 어쩌면 좀 더 익숙한 렉시컬 스코프로 돌아간다. 렉시컬 스코프와 클로저를 이용해 얻을 수 있는 self를 사용하면 this 바인딩이 어떻게 되는지에 대한 고민은 사라져 버린다.

사람들은 장황하게 쓰는 걸 싫어한다. 특히 반복적으로 써야 할 때 더욱 그렇다. ES6는 이런 짜증을 덜어주고 이 같은 문제를 고치기 위해 고안됐다. ES6의 해법인 화살표 함수는 '렉시컬 this'라고 부르는 개념을 도입했다.

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( () => { // arrow-function ftw?
                this.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // awesome?
```

간단하게 설명하면, 화살표 함수는 this 바인딩과 연계될 때는 일반 함수처럼 작동하지 않는다. 화살표 함수는 모든 this 바인딩에 대한 일반 규칙을 폐기하고, 대신 자신 가까이의 둘러싼 렉시컬 스코프에서 this 값을 받아온다. 그래서 앞의 코드에서 화살표 함수는 예상치 못하게 this 바인딩이 해제되는 일 없이 cool( )함수의 this 바인딩을 승계한다(보이는 것처럼 호출해 보면 확인 가능!).

코드를 짧게 쓰는 용도도 있지만, 내가 볼 때 화살표 함수는 그저 'this 바인딩' 규칙과 '렉시컬 스코프' 규칙을 착각하고 혼합해 쓰는 개발자들의 흔한 실수를 성문화하여 언어의 문법으로 받아들인 것뿐이다.

왜 굳이 this 스타일 코딩 패러다임을 고집해서 그 어려움과 장황함을 겪어야 하는가, 결국 굴복하고 렉시컬 참조와 혼합할 거라면 말이다. 코드를 작성할 때 특정한 접근법을 받아들이는 것은 자연스러운 일이지만, 같은 코드에 여러 접근법을 섞는 것은 그렇지 않다.

NOTE_

화살표 함수의 또 다른 단점은 익명이란 점이다. 3장을 보면 왜 기명 함수가 익명 함수보다 바람직한지 이유를 설명했다.

내가 생각할 때 이 문제를 해결하는 더 적절한 방식은 this 메커니즘을 정확하게 받아들여 사용하는 것이다.

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( function timer(){
                this.count++; // `this` is safe
                // because of `bind(…)`
                console.log( "more awesome" );
            }.bind( this ), 100 ); // look, `bind()`!
        }
    }
};
obj.cool(); // more awesome
```

화살표 함수의 새로운 동작인 렉시컬 this를 선호하든 이미 증명된 bind()함수를 선호하든 화살표 함수가 단순히 'function'의 줄임말이 아니다. 키워드 function과 화살표 함수의 작용방식에는 차이가 있다. 이 차이점을 배우고 이해해야 필요할 때 활용할 수 있다.

이제 우리는 완전히 렉시컬 스코핑(과 클로저!)를 배웠다. 렉시컬 this를 이해하기는 어렵지 않았을 것이다!

부록 D | 감사의 말

이 책과 전체 시리즈를 만드는 데 많은 사람들의 도움이 있었다.

먼저, 맨날 컴퓨터 앞에 쭈그리고 앉아있던 날 참아준 아내 크리스틴 심슨와 두 아이들 에단과 에밀리에게 감사한다. 심지어 이 책을 쓰지 않았을 때도, 자바스크립트에 대한 집착으로 필요한 시간보다 훨씬 오랫동안 모니터에 붙어있었다. 가족들과의 시간을 줄였던 것은, 독자 여러분에게 자바스크립트를 깊고 완벽하게 설명할 수 있는 책을 만들기 위해서였다. 나는 모든 것을 가족들에게 빚졌다.

오라일리의 편집자들에게 감사를 표하고 싶다. 사이먼 St. 로렌트와 브라이언 맥도날드를 비롯해 다른 편집부와 마케팅부의 직원들에게 감사하다. 그들은 멋진 동료였고, 특히 '오픈소스'로 책을 쓰고, 편집하고, 출간하는 실험에 선뜻 부응해주었다.

편집에 대해 의견을 주고 오류를 수정하면서 이 책을 만드는 데 참여한 많은 이들에게 감사를 표한다. 샐리 파워즈, 팀 페로, 이반 보든, 포레스트 L 노벨, 제니퍼 다비스, 제시 할린을 비롯해 많은 이들이 도왔다. 멋진 추천사를 써준 쉐인 허드슨에게 감사한다.

TC39 위원회 위원을 포함해 커뮤니티의 셀 수 없이 많은 이들이 다양한 지식을 나눴다. 특히, 귀찮아하지 않고 나의 끊임없는 질문과 탐구에 응하여 참을성 있게 설명해준 데 감사한다. 존 데이비드 달튼, 쥬리 "캉액스" 자이체프, 마시아스 바이넨스, 릭 월드론, 악셀 로슈마이어, 니콜라스 자카스, 앙구스 크롤, 조단 하반드, 데이브 허만, 브랜든 아이크, 앨런 위츠-브룩, 브래들리 메크, 도메닉 데니콜라, 데이비드 월시, 팀 디즈니, 크리스 카월, 피터 반 데 지, 안드리아 기암마치, 키트 캠브리지 등 많은 이들이 도와주었지만, 이름을 다 적기는커녕 시작도 못했다.

『You Don't Know JS』 시리즈는 킥스타터를 통해 태어났다. (거의) 500 명에 달하는 너그러운 후원자들에게 감사를 표하고 싶다. 그들이 없었다면 이 시리즈는 나오지 못했을 것이다:

이 시리즈는 오픈소스로 작성됐다. 편집과 생산도 마찬가지다. 개발자 커뮤니티가 이번 일을 하면서 Github의 신세를 졌다. 덕분에 책을 낼 수 있었다!

내가 이름을 적지 못한 수많은 사람들에게 다시 한 번 감사를 표한다. 이 시리즈는 우리 '모두의 것'이다. 자바스크립트 언어를 이해하고 깨우치는 데 이 시리즈가 공헌하길 바란다. 현재와 미래의 개발자 커뮤니티 기여자들에게 도움이 되기를.