Extracted from:

CoffeeScript

Accelerated JavaScript Development, Second Edition

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

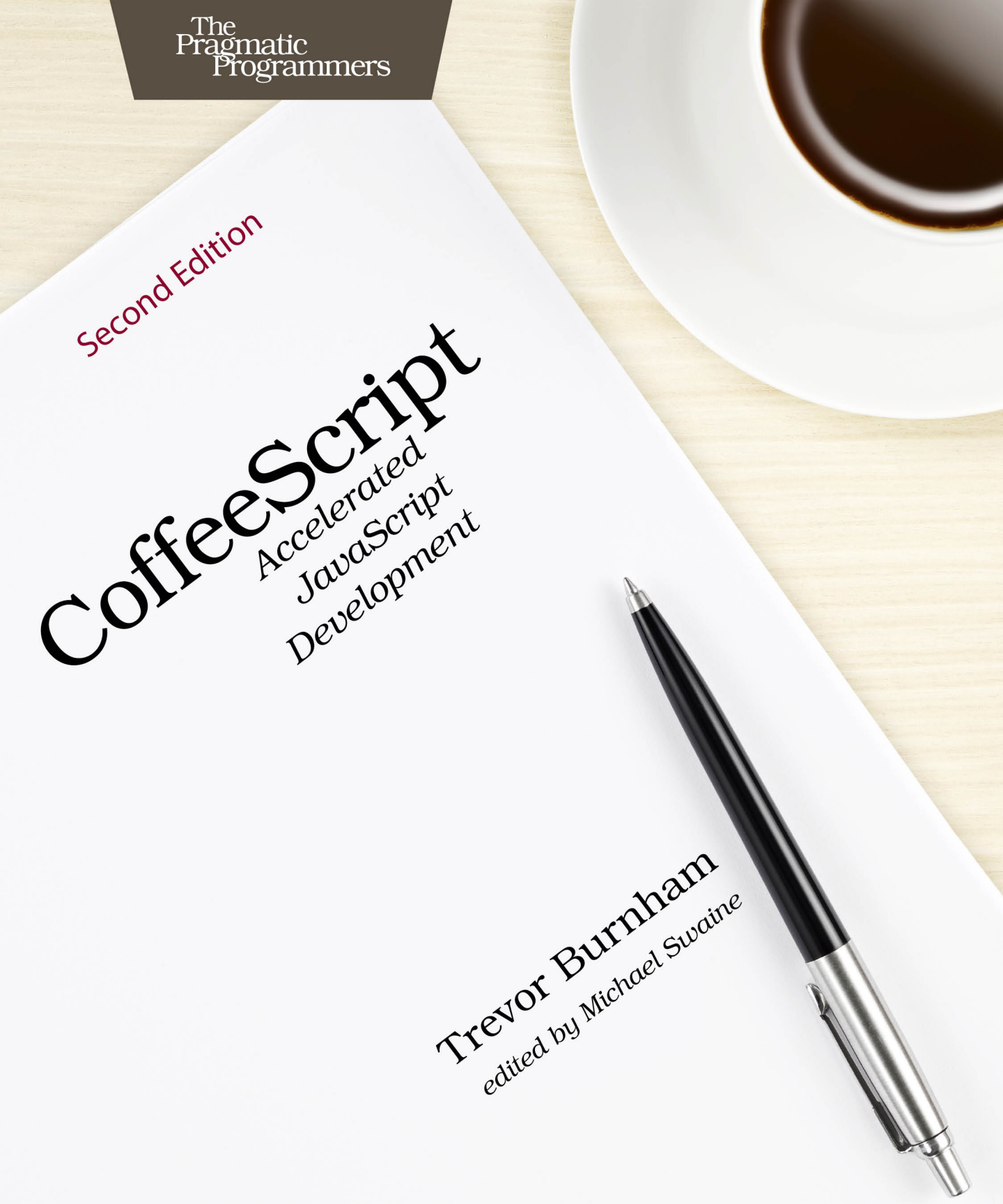
Second Edition

CoffeeScript

Accelerated JavaScript Development

Trevor Burnham

edited by Michael Swaine



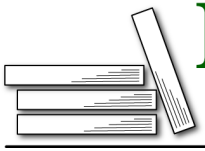
CoffeeScript

Accelerated JavaScript Development, Second Edition

Trevor Burnham

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing (indexer)
Cathleen Small (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-94122-226-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2015

Mini-Project: Refactored Checkbook Balancer

Let's take what we've learned from this chapter and use it to add a new feature to the checkbook balancer from *Mini-Project: Checkbook Balancer*, on page ? —while making the code more maintainable!

The original checkbooks program allowed you to pick one of three accounts ("checking," "savings," and "mattress") and deposit or withdraw money from that account. The program had one limitation that made it slightly impractical: no persistence. When you closed the program, all account balances were reset to $0. The new checkbooks2 will remedy that by serializing account objects to a JSON file. We'll also add a "transfer" action that moves money from one account to another.

Let's start by creating a new directory for our project and installing the same dependencies as before:

```
$ npm init
$ npm install --save inquirer
$ npm install --save numeral
```

And we'll add one new one, jsonfile:[4]

```
$ npm install --save jsonfile
```

Now into the code! We're going to make three changes to our createAccount function. First, we're going to add a transfer method to the account object. Second, we're going to call a utility saveState function every time we perform an action. Third, we're going to switch from having argument lists to having a single "options" argument and extracting the values we want using the destructuring syntax. This gives us a lot more flexibility if we add new features to a function, because we don't have to add more and more arguments in an increasingly hard-to-remember order.

Collections/checkbooks2/checkbooks2.coffee

```
createAccount = ({name}) ->
  {
    name: name
    balance: 0

    description: ->
      "#{@name}: #{dollarsToString(@balance)}"

    deposit: ({amount}) ->
      @balance += amount
```

4. https://github.com/jprichardson/node-jsonfile

```
      saveState()
      @

    withdraw: ({amount}) ->
      @balance -= amount
      saveState()
      @

    transfer: ({toAccount, amount}) ->
      @balance -= amount
      toAccount.balance += amount
      saveState()
      @
  }
```

We're going to start with the same accounts as the original checkbooks, but unlike before, we're going to store them in an array:

Collections/checkbooks2/checkbooks2.coffee

```
accounts = [
  createAccount({name: 'Checking'})
  createAccount({name: 'Savings'})
  createAccount({name: 'Mattress'})
]
```

Now for the tricky part: defining the interface. In the original checkbooks I tried to make this code as linear as possible, which worked reasonably well because our three prompts were always given in the same order: pick an account, pick an action, enter an amount. But the new "transfer" action requires a additional input (the destination account), which we'd like to prompt for before we prompt for the amount. Writing all of this logic in a linear fashion is still possible, but the resulting code wouldn't be much fun to read. So let's separate out the details—the parameters we pass to Inquirer.js to define each prompt—from the core application logic. This makes the core logic nice and easy to grok, with just two "steps" (one before an action is selected, and the other after):

Collections/checkbooks2/checkbooks2.coffee

```
inquirer = require('inquirer')

mainStep = ->
  inquirer.prompt([
    makeAccountPrompt()
    makeActionPrompt()
  ], postActionStep)

postActionStep = ({account, action}) ->
  prompts = [makeAmountPrompt({action})]
  if action is 'transfer'
```

```
    prompts.unshift makeToAccountPrompt({fromAccount: account})
  inquirer.prompt(prompts, ({amount, toAccount}) ->
    amount = inputToNumber(amount)
    account[action]({amount, toAccount})
    mainStep()
  )
```

Once again, we're going heavy on object arguments so that we don't have to worry about the order of arguments. CoffeeScript's destructuring syntax really shines here.

And now for the prompt definitions:

Collections/checkbooks2/checkbooks2.coffee

```
makeAccountPrompt = ->
  {
    name: 'account'
    message: 'Pick an account:'
    type: 'list'
    choices: for account in accounts
      {name: account.description(), value: account}
  }

makeActionPrompt = ->
  {
    name: 'action'
    message: 'Pick an action:'
    type: 'list'
    choices: [
      {name: 'Deposit $ into this account', value: 'deposit'}
      {name: 'Withdraw $ from this account', value: 'withdraw'}
      {name: 'Transfer $ to another account', value: 'transfer'}
    ]
  }

makeToAccountPrompt = ({fromAccount}) ->
  {
    name: 'toAccount'
    message: 'Pick an account to transfer $ to:'
    type: 'list'
    choices: for account in accounts when account isnt fromAccount
      {name: account.description(), value: account}
  }

makeAmountPrompt = ({action}) ->
  {
    name: 'amount'
    message: "Enter the amount to #{action}:"
    type: 'input'
    validate: (input) =>
      if isNaN(inputToNumber(input))
```

```
      return 'Please enter a numerical amount.'
    if inputToNumber(input) < 0
      return 'Please enter a non-negative amount.'
    true
}
```

We need to define the same utility functions as in the original checkbooks, plus a new one to save the state of our accounts to a JSON file:

Collections/checkbooks2/checkbooks2.coffee

```
numeral = require ('numeral')
jsonfile = require('jsonfile')

dollarsToString = (dollars) ->
  numeral(dollars).format('$0,0.00')

inputToNumber = (input) ->
  parseFloat input.replace(/[$,]/g, ''), 10

saveState = ->
  jsonfile.writeFileSync('./data.json', accounts)
```

Now when we start the program, we try to load that JSON file and set the appropriate balance value for each account. Then we go to the main step:

Collections/checkbooks2/checkbooks2.coffee

```
try
  data = jsonfile.readFileSync('./data.json')
  for account, i in accounts
    account.balance = data[i].balance

mainStep()
```

Here's what it looks like in action:

```
$ coffee checkbooks2.coffee
[?] Pick an account: Checking: $0.00
[?] Pick an action: deposit
[?] Enter the amount to deposit: 1000000
[?] Pick an account: Checking: $1,000,000.00
[?] Pick an action: transfer
[?] Pick an account to transfer $ to: (Use arrow keys)
> Savings: $0.00
  Mattress: $0.00
```