

# Data Oriented Programming

Unlearning objects

Yehonathan Sharvit



MANNING



**MEAP Edition**  
**Manning Early Access Program**  
**Data Oriented Programming**  
Unlearning objects  
**Version 14**

Copyright 2022 Manning Publications

For more information on this and other Manning titles go to  
[manning.com](http://manning.com)

# welcome

---

Thank you for purchasing the MEAP for *Data Oriented Programming*.

The book is written for developers having experience in a **high level programming language**. It could be a classic Object Oriented language like Java or C# or a dynamically typed language like JavaScript, Ruby or Python. We assume that you have already built (alone or in a team) a couple of **web systems** either backend or frontend.

**Data-Oriented Programming** (DOP) is a programming paradigm that makes the systems we build **less complex**. The cool thing is that DOP is **language agnostic**: it is applicable to any programming language.

I discovered Data Oriented programming 10 years ago when I started to code in Clojure. Since then, the quality of my design and my code has increased significantly and the systems I build in Clojure and in other programming languages are much simpler and much more flexible.

DOP is based on **4 fundamental principles** that we expose briefly in Chapter 0. The principles might seem basic at first sight, but when you apply them in the context of a production ready information system, they become very powerful.

Chapter 1 exposes some **common pains** that **Object Oriented developers** experience when they develop a system. Please don't read it as a critics of Object Oriented Programming. The main purpose of Chapter 1 is to motivate you to learn a different programming paradigm.

Starting from Chapter 2, we expose — one by one — the four principles of DOP and their benefits in the context of a production ready information system.

In order to make the teachings very concrete, we demonstrate how the principles of DOP are translated in code. We have chosen JavaScript as the main language for the **code snippets** of the book, but the ideas are applicable to any programming language. We have chosen JavaScript because it supports both Object Oriented and Functional programming style and its syntax is easy to read even for folks not familiar with JavaScript.

The book is full of **diagrams** and **mind maps** that illustrate the ideas.

The teachings of the book are conveyed through a **story** of an Object-Oriented programmer that meets a Data-Oriented Programming expert and learn from them how DOP makes a system less complex and more flexible. I hope that you find the **conversation** between the developer and the expert fun to read and that it clarifies the teaching in the sense that the questions the developers ask the expert resonate well with the questions you ask yourself during reading.

I truly believe that Data-Oriented Programming will make you a **better developer**, as it has been the case for me since I discovered it 10 years ago.

I look forward to reading any questions or comments you may have along the way on Manning's [liveBook Discussion Forum](#). Your feedback is an invaluable part of making this book the best that it can be.

One last thing, the name of the main character of the book is: You!

— Yehonathan Sharvit

# *brief contents*

---

## **PART 1: FLEXIBILITY**

- 1 *Complexity of Object-Oriented Programming*
- 2 *Separation between code and data*
- 3 *Basic data manipulation*
- 4 *State management*
- 5 *Basic concurrency control*
- 6 *Unit tests*

## **PART 2: SCALABILITY**

- 7 *Basic data validation*
- 8 *Advanced concurrency control*
- 9 *Persistent data structures*
- 10 *Database operations*
- 11 *Web services*

## **PART 3: MAINTAINABILITY**

- 12 *Advanced data validation*
- 13 *Polymorphism*
- 14 *Advanced data manipulation*
- 15 *Debugging*

## **APPENDIXES**

- A *Principles of Data-Oriented Programming*
- B *Generic data access in statically-typed languages*
- C *Data-Oriented Programming: A link in the chain of programming paradigms*
- D *Lodash reference*

# Part I

## Flexibility

A new project

It's Monday morning, Theodore is sitting with Nancy on the terrace of "La vita è bella", an italian coffee shop near the San Francisco Zoo. Nancy is an entrepreneur looking for a development agency for her startup, Klaufim. Theo works for Albatross, a software development agency that seeks to regain the trust of startups.

Nancy and her business partner have raised seed money for Klaufim, a social network around books. Klaufim's unique value proposition is to combine the online world with the physical world by allowing users to borrow books from physical local libraries and to meet online to discuss the books. Most parts of the product rely on the integration of already existing online services. The only piece that requires software development is what Nancy calls a Global Library Management System.

Their discussion is momentarily interrupted by the waiter who brings Theo his tight espresso and Nancy her americano with milk on the side.

**THEO:** What's a Global Library Management System in your mind?

**NANCY:** It's a software system that handles the basic housekeeping functions of a library, mainly around the book catalog and the library members.

**THEO:** Could you be a little bit more specific?

**NANCY:** Sure. For the moment, we need a quick prototype. If the market response to Klaufim is positive, we will move forward with a big project.

**THEO:** What features do you need for the prototype phase?

Nancy grabs the napkin under her coffee mug and she writes down a couple of bullet points on the napkin.

**SIDE BAR****The requirements for the Klaflim prototype**

- Two kinds of users: library members and librarians
- Users log in to the system via email and password.
- Members can borrow books
- Members and librarians can search books by title or by author
- Librarians can block and unblock members (e.g. when they are late in returning a book)
- Librarians can list the books currently lent by a member
- There could be several copies of a book
- A book copy belongs to a physical library.

**THEO:** Well, that's pretty clear.

**NANCY:** How much time would it take for your company to deliver the prototype?

**THEO:** I think we should be able to deliver within a month. Let's say Wednesday the 30th.

**NANCY:** That's too long. We need it in two weeks!

**THEO:** That's tough! Can you cut a feature or two?

**NANCY:** Unfortunately, we cannot cut any feature, but if you like you can make the search very basic.

You really don't want to lose this contract. You are willing to work hard and sleep late.

**THEO:** I think it should be doable by Wednesday the 16th.

**NANCY:** Perfect!

# *Complexity of Object-Oriented Programming*



## This chapter covers

- The tendency of OOP to increase system complexity
- What makes OOP systems hard to understand
- The cost of mixing of code and data together into objects

### **1.1 A capricious entrepreneur**

In this chapter, we explore why Object-Oriented Programming (OOP) systems tend to be complex.

This complexity is not related to the syntax or the semantics of a specific OOP language. It is something that is inherent to OOP's fundamental insight that programs should be composed from objects that consist of some state together with methods for accessing and manipulating that state.

Over the years, OOP ecosystems have alleviated this complexity increase by adding new features to the language (e.g. anonymous classes and anonymous functions) and by developing frameworks that hide some of this complexity by providing a simpler interface to developers (e.g. Spring and Jackson in Java). Internally, they rely on advanced features of the language like reflection and custom annotations.

This chapter is not meant to be read as critical of OOP. Its purpose is to raise awareness of the tendency towards increased complexity of OOP as a programming paradigm and to motivate you to discover a different programming paradigm where the system complexity tends to be reduced, namely Data Oriented programming.

## 1.2 OOP design: classic or classical?

**WARNING** Theo, Nancy, and their new project were introduced in the opener for Part 1. Please take a moment to read the opener if you missed it.

Theo gets back to the office with Nancy's napkin in his pocket and a lot of anxiety in his heart as he knows he has committed to a tough deadline. But he had no choice. Last week, Monica, his boss, told him quite clearly that he had to close the deal with Nancy no matter what!

Albatross is a software consulting company with customers all over the world. It used to have lots of customers among startups, but over the last year many projects were badly managed and the Startup department lost the trust of its customers. That's why they moved Theo from the Enterprise department to the Startup department as a senior Tech lead. His job is to close deals and to deliver on time!

### 1.2.1 The design phase

Before rushing to his laptop to code the system, Theo grabs a sheet of paper—much bigger than the napkin—and starts to draw a UML class diagram of the system that will implement the Klafim prototype.

Theo is an Object-Oriented programmer. For him, there is no question: Every business entity is represented by an object and every object is made from a class.

**SIDE BAR** The requirements for the Klafim prototype

- Two kinds of users: library members and librarians
- Users log in to the system via email and password.
- Members can borrow books
- Members and librarians can search books by title or by author
- Librarians can block and unblock members (e.g. when they are late in returning a book)
- Librarians can list the books currently lent by a member
- There can be several copies of a book
- A book copy belongs to a physical library.

Here are the main classes that Theo identifies for Klafim Global Library Management System:

**SIDE BAR****The main classes of the library management system**

- **Library:** The central part for which the system is designed
- **Book:** A book
- **BookItem:** A book can have multiple copies, each copy is considered as a book item
- **BookLending:** When a book is lent, a book lending object is created
- **Member:** A member of the library
- **Librarian:** A librarian
- **User:** A base class for Librarian and Member
- **Catalog:** Contains list of books
- **Author:** A book author

That was the easy part. Now comes the difficult part: the relationships between the classes.

After two hours or so, Theo comes up with a first draft of a design for the Library Management System. It looks like the diagram shown in [1.1](#).

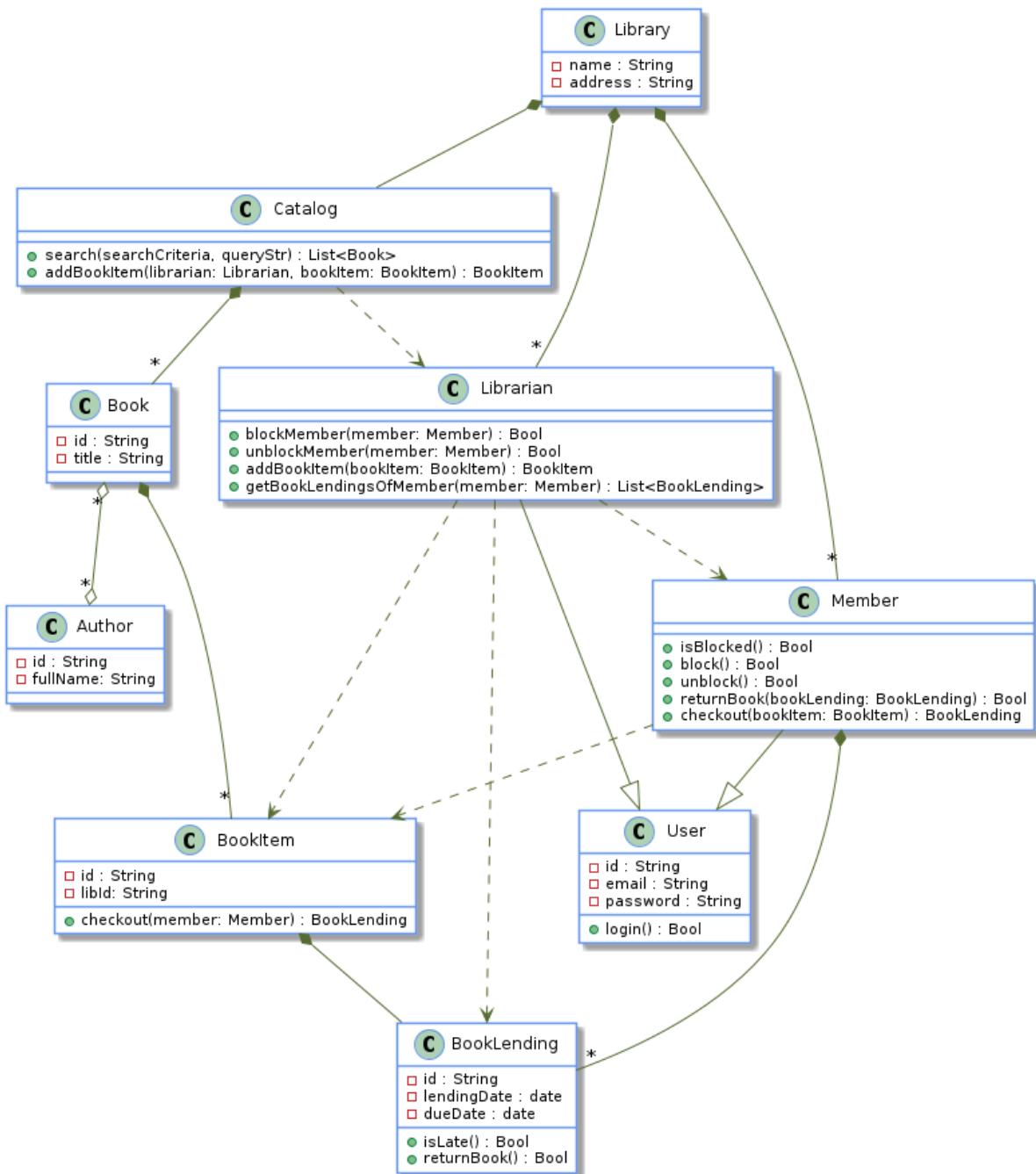


Figure 1.1 A class diagram for a Library management system

This design is meant to be very naive and by no means pretends to cover all the features of the system.

**WARNING** The design presented here doesn't pretend to be the smartest OOP design: experienced OOP developers would probably leverage a couple of design patterns and suggest a much better design.

This design serves two purposes:

1. For Theo - *the developer* - it is rich enough to start coding
2. For me - *the author of the book* - it is rich enough to illustrate the complexity of a typical OOP system

Theo feels proud of himself and of the design diagram he produced. He definitely deserves a cup of coffee.

Near the coffee machine, Theo meets Dave, a junior software developer who joined Albatross a couple of weeks ago. Theo and Dave appreciate each other. Dave's curiosity leads him to ask challenging questions. Meetings near the coffee machine often turn into interesting discussions about programming.

**THEO:** Hey Dave! How's it going?

**DAVE:** Today? Not great. I'm trying to fix a bug in my code! I can't understand why the state of my objects always changes. I'll figure it out thought, I'm sure. How's your day going?

**THEO:** I just finished the design of a system for a new customer.

**DAVE:** Cool! Would it be OK for me to see it? I'm trying to improve my design skills.

**THEO:** Sure! I have the diagram on my desk. We can take a look now if you like.

## 1.2.2 UML 101

Latte in hand, Dave follows Theo to his desk. Theo proudly shows Dave his piece of art: the UML diagram for the Library Management System in [1.1](#).

Dave seems really excited.

**DAVE:** Wow! Such a detailed class diagram.

**THEO:** Yeah. I'm pretty happy with it.

**DAVE:** The thing is that I can never remember the meaning of the different arrows.

**THEO:** There are 4 types of arrows in my class diagram: *composition*, *association*, *inheritance* and *usage*.

**DAVE:** What's the difference between composition and association?

**WARNING**     Don't worry if you're not familiar with OOP jargon. We're going to leave it aside in the next chapter.

**THEO:** It's all about whether the objects can live without each other: with composition, when

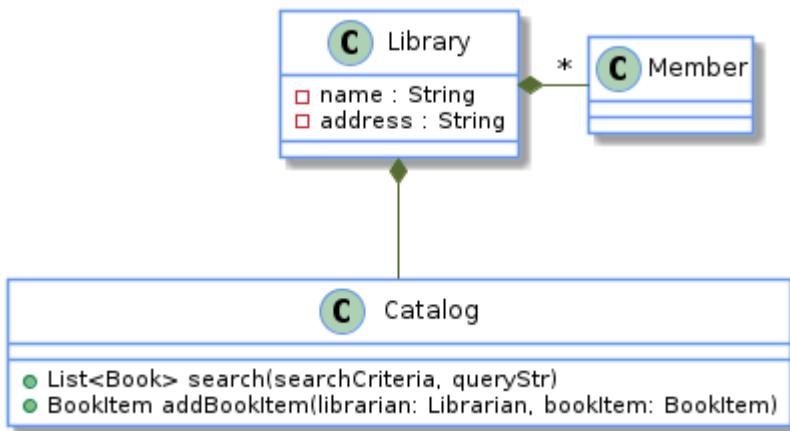
one object dies, the other one dies also, while in an association relation, each object has an independent life.

**TIP**

**In a composition relation, when one object dies, the other one dies also, while in an association relation, each object has an independent life cycle.**

In the class diagram there are two kinds of composition relation symbolized by an arrow with a plain diamond at one edge, and an optional star at the other edge:

1. A `Library` owns a `Catalog`: That's a one-to-one composition relation: if a `Library` object dies, then its `Catalog` object dies with it.
2. A `Library` owns many `Members`: That's a one-to-many composition relation: if a `Library` object dies, then all its `Member` objects die with it.



**Figure 1.2 Two kinds of composition: one-to-one and one-to-many. In both cases, when an object dies, the composed object dies with it.**

**TIP**

**A composition relation is represented by a plain diamond at one edge and an optional star at the other edge.**

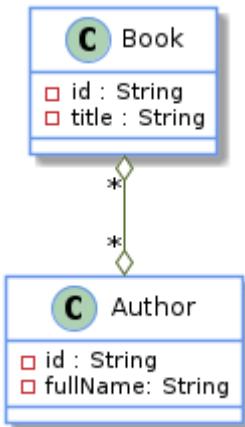
**DAVE:** Do you have association relations in your diagram?

**THEO:** Take a look at the arrow between `Book` and `Author`. It has an empty diamond and a star at both edges: it's a many to many association relation.

A book can be written by multiple authors and an author can write multiple books. Moreover, `Book` and `Author` objects can live independently: the relation between books and authors is a many-to-many association relation.

**TIP**

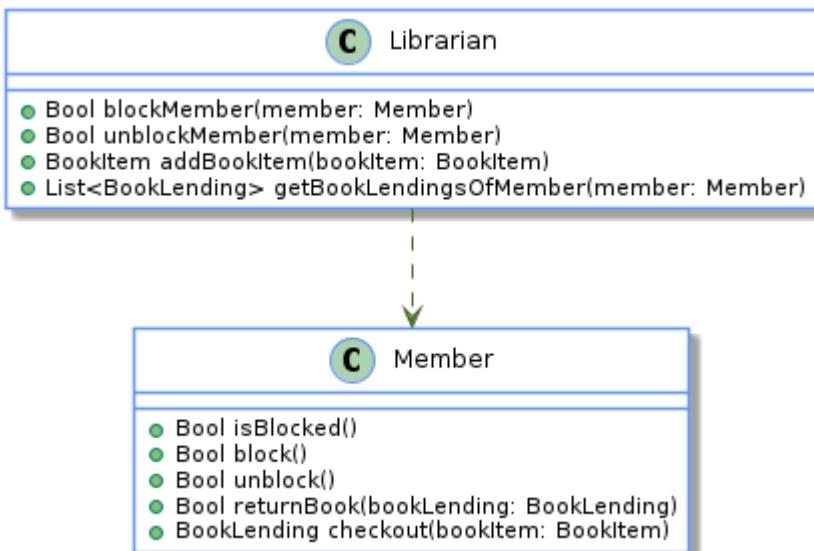
**A many-to-many association relation is represented by an empty diamond and a star at both edges.**



**Figure 1.3** Many to many association relation: each object lives independently

**DAVE:** I also see a bunch of dashed arrows in your diagram.

**THEO:** Dashed arrows are for usage relations: when a class uses a method of another class. Consider for example, at the `Librarian::blockMember` method. It calls `Member::block`.



**Figure 1.4** Usage relation: a class uses a method of another class

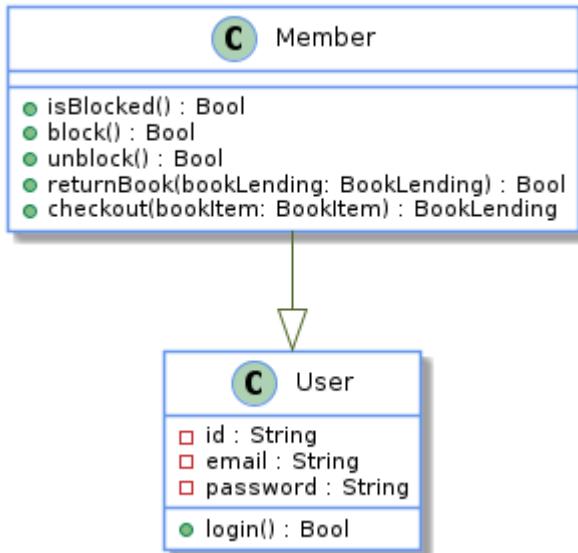
**TIP**      Dashed arrows are for usage relations: for instance, when a class uses a method of another class.

**DAVE:** I see. And I guess a plain arrow with an empty triangle—like the one between Member and User—represents inheritance.

**THEO:** Absolutely.

**TIP**

Plain arrows with empty triangles represent class inheritance, where the arrow points towards the superclass.



**Figure 1.5 Inheritance relation: a class derives from another class**

### 1.2.3 Explaining each piece of the class diagram

**DAVE:** Thank's for the UML refresher! Now I remember what the different arrows mean.

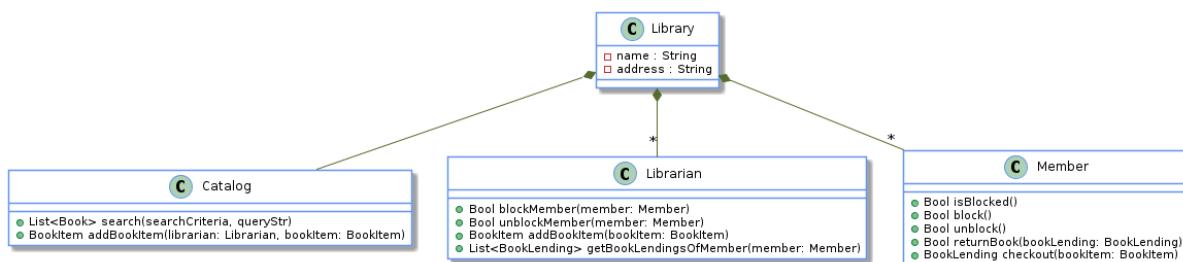
**THEO:** My pleasure.

**DAVE:** What class should I look at first?

**THEO:** I think you should start from **Library**.

#### THE LIBRARY CLASS

The **Library** is the root class of the system.



**Figure 1.6 The Library class**

In terms of code (behavior), a **Library** object does nothing on its own, it delegates everything to

objects it owns.

In terms of data, a `Library` object owns:

1. Multiple `Member` objects
2. Multiple `Librarian` objects
3. A single `Catalog` object

**WARNING** We use the terms code and behavior interchangeably.

## LIBRARIAN, MEMBER AND USER CLASSES

`Librarian` and `Member` who both derive from `User`.

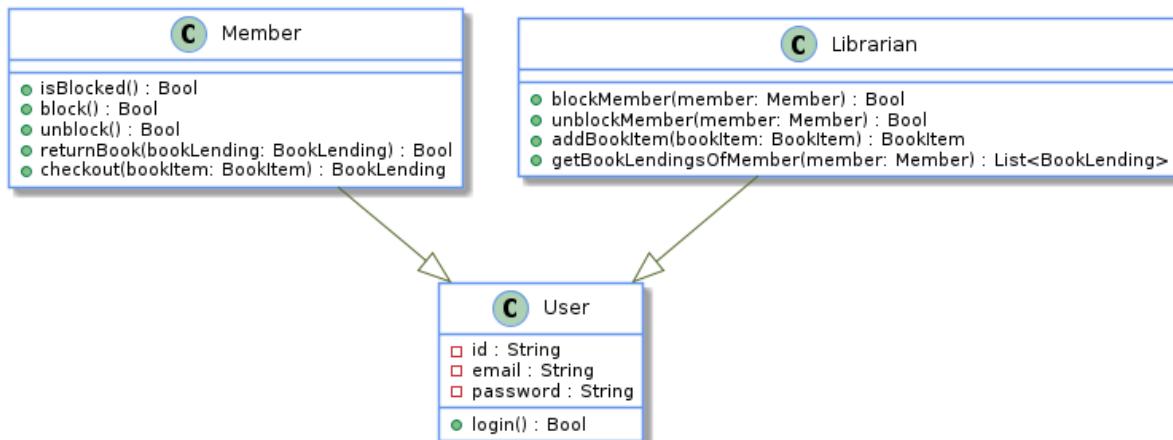


Figure 1.7 `Librarian` and `Member` derive from `User`

The `User` class represents a user of the library.

1. In terms of data members, it sticks to the bare minimum: it has a `id`, `email` and `password` (with no security and encryption for now).
2. In terms of code, it can login via `login`

The `Member` class represents a member of the library.

1. It inherits from `User`
2. In terms of data members, it has nothing more than `User`
3. In terms of code, it can:
  - A. Checkout a book via `checkout`
  - B. Return a book via `returnBook`
  - C. Block itself via `block`
  - D. Unblock itself via `unblock`
  - E. Answer if it is blocked via `isBlocked`
4. It owns multiple `BookLending` objects

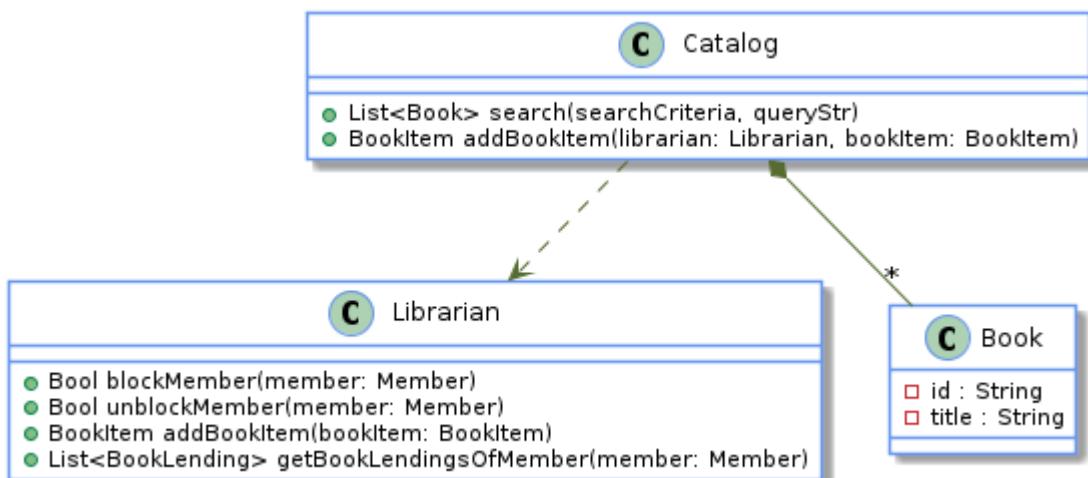
5. It uses `BookItem` in order to implement `checkout`

The `Librarian` class represents a librarian.

1. It derives from `User`
2. In terms of data members, it has nothing more than `User`
3. In terms of code, it can:
  - A. Block and unblock a `Member`
  - B. List the book lendings of a member via `getBookLendings`
  - C. Add book items to the library via `addBookItem`
4. It uses `Member` in order to implement `blockMember`, `unblockMember` and `getBookLendings`
5. It uses `BookItem` in order to implement `checkout`
6. It uses `BookLending` in order to implement `getBookLendings`

## THE CATALOG CLASS

The `Catalog` class is responsible for the management of the books.



**Figure 1.8 The Catalog class**

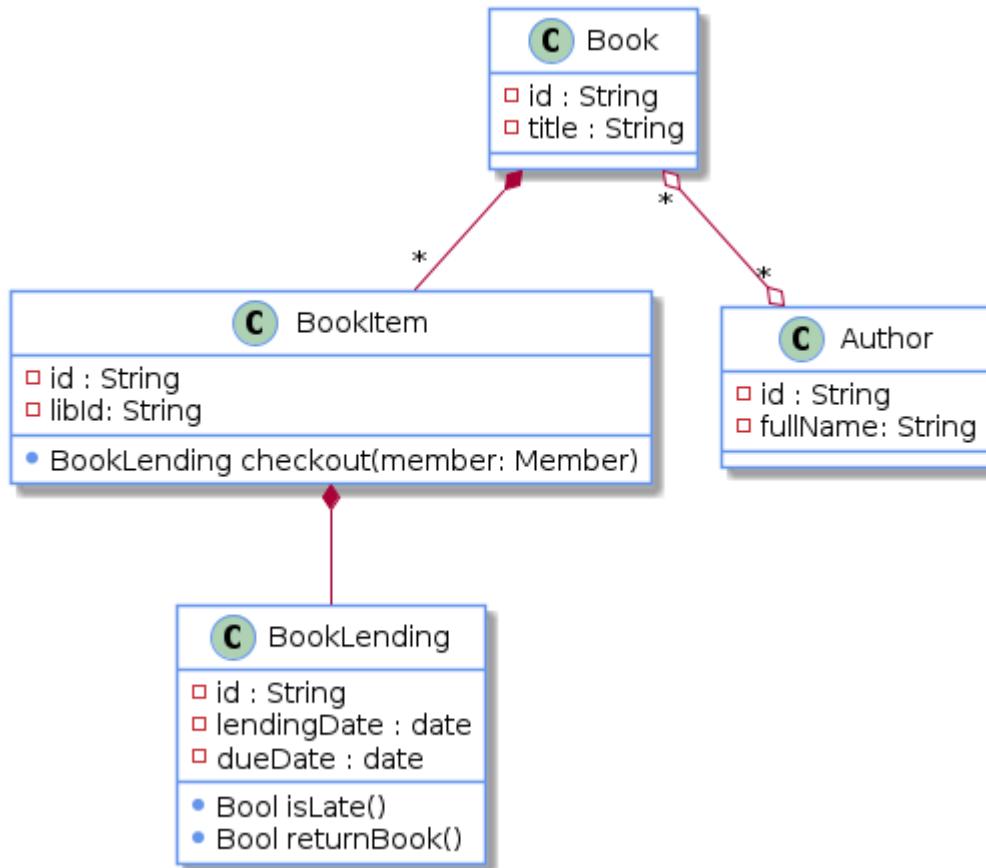
In terms of code, a `Catalog` object can:

- a. Search books via `search`
- b. Add book items to the library via `addBookItem`
  - A. It uses `Librarian` in order to implement `addBookItem`

In terms of data, a `Catalog` owns:

1. Multiple `Book` objects

## THE BOOK CLASS



**Figure 1.9 The Book class**

In terms of data, a Book object:

1. should have the bare minimum: an `id`, and a `title`
2. is associated with multiple `Author` objects (A book might have multiple authors)
3. owns multiple `BookItem` objects, one for each copy of the book

## THE BOOKITEM CLASS

The `BookItem` class represents a book copy. A book could have many copies.

In terms of data a `BookItem` object:

1. should have the bare minimum data for members : an `id`, and a `libID` (for its physical library ID)
2. owns multiple `BookLending` objects, one for each time the book is lent

In terms of code:

1. It can be checked out via `checkout`

### 1.2.4 The implementation phase

After this detailed investigation of Theo's diagram, Dave expresses his admiration.

**DAVE:** Wow! That's amazing.

**THEO:** Thank you.

**DAVE:** I didn't know people were really spending the time to write down their design in such detail, before coding.

**THEO:** I always do that. It saves me lot of time during the coding phase.

**DAVE:** When will you start coding?

**THEO:** When I finish my latte.

Theo grabs his coffee mug notices his latte has become an Iced Latte. He was so excited to show his class diagram to Dave that he forgot to drink it.

### 1.3 Sources of complexity

While Theo is getting himself another cup of coffee (a cappuccino this time), I would like to challenge his design. It might look beautiful and clear on the paper but I claim that this design makes the system hard to understand.

It's not that Theo picked the wrong classes or that he misunderstood the relationships between the classes. It's much deeper. It's about the programming paradigm he chose to implement the system. It's about the Object Oriented paradigm. It's about the tendency of OOP to increase the complexity of a system.

**TIP**

OOP has a tendency to create complex systems.

Throughout the book, the type of complexity I refer to is that which makes systems hard to understand - as it is defined in the paper "Out of the Tar Pit" (<https://github.com/papers-we-love/papers-we-love/blob/master/design/out-of-the-tar-pit.pdf>). It has nothing to do with the type of complexity that deals with the amount of resources consumed by a program.

Similarly, when I refer to simplicity, I mean *not complex*, in other words: *easy to understand*.

Keep in mind that complexity and simplicity (like hard and easy) are not absolute but relative concepts. We can compare the complexity of two systems and determine whether system A is more complex (or simpler) than system B.

**NOTE**

**Complex in the context of this book means: hard to understand**

As mentioned in the introduction of this chapter, there are many ways in OOP to alleviate complexity. The purpose of this book is not to be critical of OOP, but rather to present a programming paradigm called Data-Oriented Programming (DOP) that tends to build systems that are less complex. In fact, the DOP paradigm is compatible with OOP. If one chooses to build a OOP system that adheres to DOP principles, the system will be less complex.

According to DOP, the main sources of complexity in Theo's system—and of many traditional OOP systems—are:

1. Code and data are mixed
2. Objects are mutable
3. Data is locked in objects as members
4. Code is locked into classes as methods

This analysis is similar to what Functional Programming (FP) thinks about traditional Object-Oriented Programming. However, as we will see through the book, the data approach that Data-Oriented Programming takes in order to reduce system complexity differs from the FP approach.

In Appendix A, we illustrate how to apply DOP principles both in OOP and FP styles.

**TIP**

**DOP is compatible both with OOP and FP.**

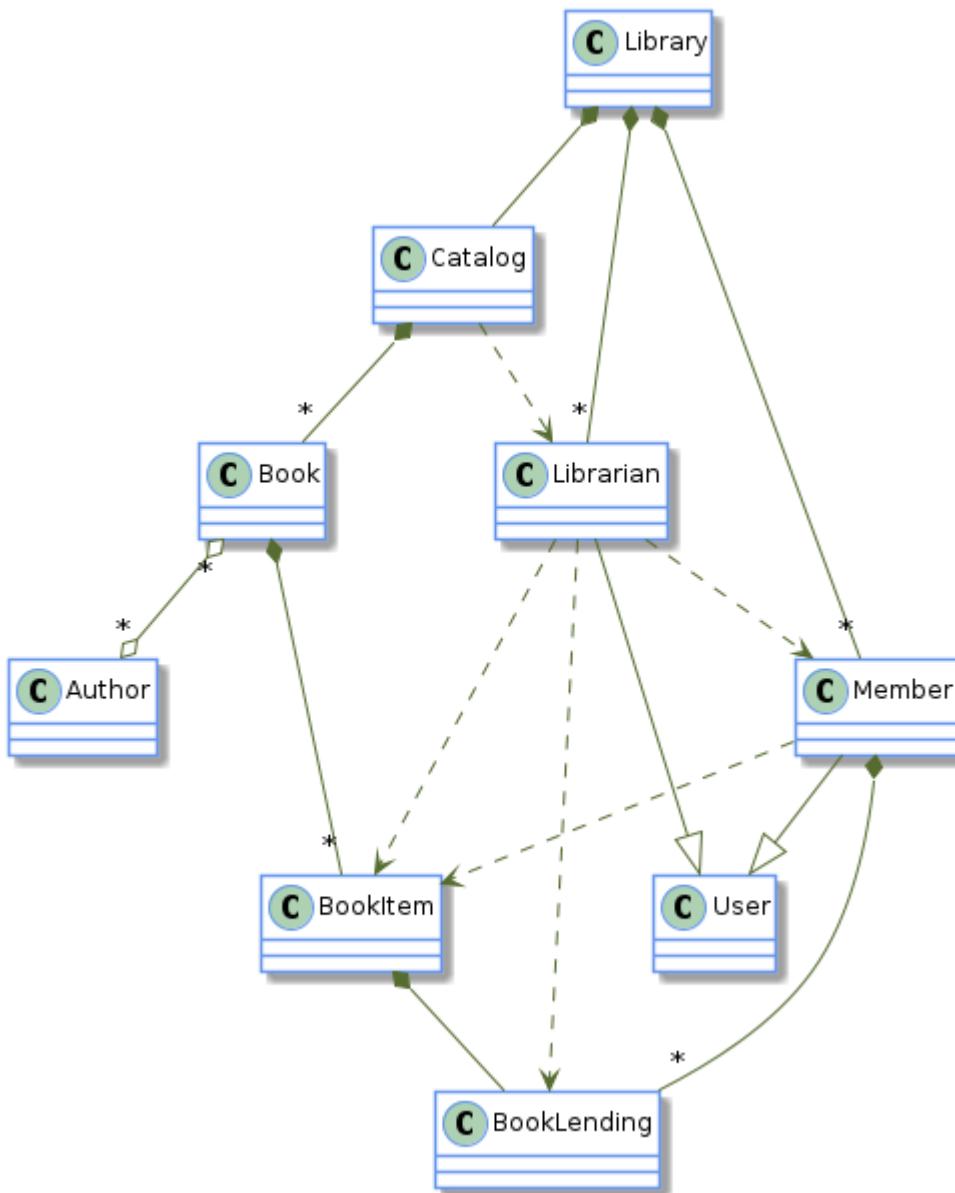
In the remaining sections of this chapter, we are going to illustrate each of the above aspects—summarized in [1.1](#)—in the context of the Klafim project and explain in what sense they are a source of complexity.

**Table 1.1 Aspects of Object Oriented programming and their impact on system complexity**

Aspect	Impact on complexity
Code and data are mixed	Classes tend to be involved in many relations
Objects are mutable	Extra thinking when reading code
Objects are mutable	Explicit synchronization on multi-threaded environments
Data is locked in objects	Data serialization is not trivial
Code is locked in classes	Class hierarchies are complex

### 1.3.1 Many relationships between classes

One way to assess the complexity of a class diagram is to look only at the entities and their relationships (ignoring members and methods) as in [1.10](#).



**Figure 1.10 A class diagram overview for a Library management system**

When we design a system, we have to define the relationships between different pieces of code and data: that's unavoidable.

**TIP**

In OOP, code and data are mixed together in classes, data as members and code as methods.

From a system analysis perspective, the fact that code and data are mixed together makes the

system complex in the sense that entities tend to be involved in many relationships.

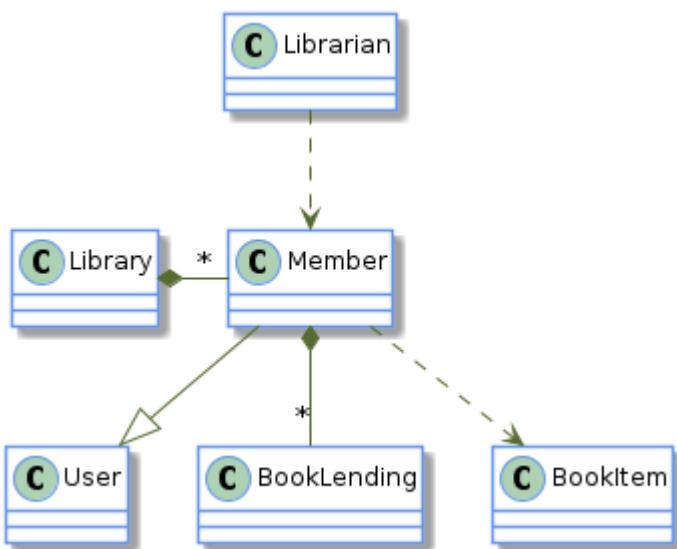
In [1.11](#), we take a closer look at the `Member` class. `Member` is involved 5 relations: 2 data relations and 3 code relations.

Data relations:

1. Library *has* many Members
2. Member *has* many BookLendings

Code relations:

1. Member *extends* User
2. Librarian *uses* Member
3. Member *uses* BookItem



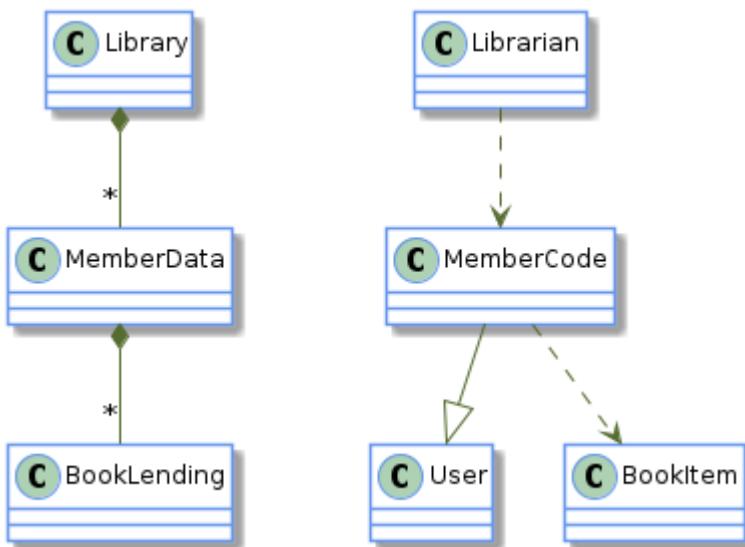
**Figure 1.11** The `Member` is involved in 5 relations

Imagine for a moment that we were able somehow to split the `Member` class into two separate entities:

- `MemberCode` for the code
- `MemberData` for the data

Instead of a `Member` class with 5 relations, we would have the diagram shown in [1.12](#), with:

- A `MemberCode` entity with 3 relations
- A `MemberData` entity with 2 relations

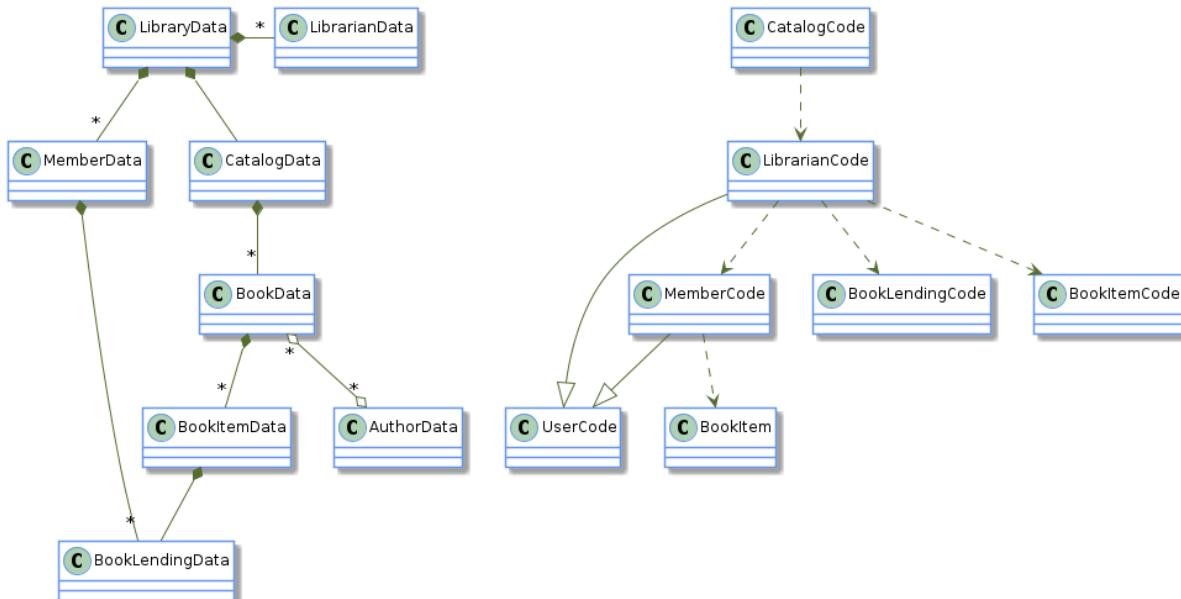


**Figure 1.12** A class diagram where `Member` is split into code and data entities

The class diagram where `Member` is split into `MemberCode` and `MemberData` is made of two independent parts, where each part is easier to understand than the original diagram.

Now, let's split every class of our original class diagram into code and data entities. The resulting diagram is shown in [1.13](#): Now, the system is made of two independent parts:

1. A part that involves only data entities
2. A part that involves only code entities



**Figure 1.13** A class diagram where every class is split into code and data entities

**TIP**

A system where every class is split into two independent parts, code and data, is simpler than a system where code and data are mixed.

The resulting system—made of two independent sub-systems—is easier to understand than the original system. The fact that the two sub-systems are independent means that each sub-system can be understood separately and in any order.

The resulting system not simpler by *accident*, it is a *logical consequence* of separating code from data.

**TIP**

**A system made of multiple simple independent parts is less complex than a system made of a single complex part.**

### 1.3.2 Unpredictable code behavior

You might be a bit tired after the system-level analysis that we presented in the previous section.

Let's get refreshed and look at some code.

Please take a look at the code shown in [1.1](#): we get the blocked status of a member and display it twice.

If I tell you that when I called `displayBlockedStatusTwice`, the program displayed `true` on the first `console.log` call, can you tell me what the program displayed on the second `console.log` call?

#### **Listing 1.1 Really simple code**

```
class Member {
    isBlocked;

    displayBlockedStatusTwice() {
        var isBlocked = this.isBlocked;
        console.log(isBlocked);
        console.log(isBlocked);
    }
}

member.displayBlockedStatusTwice();
```

"Of course, it displayed `true` again", you tell me.

And you are right.

Now, please take a look at a slightly different pseudocode as shown in [1.2](#): here we display twice the blocked status of a member without assigning a variable.

Same question as before: If I tell you that when I called `displayBlockedStatusTwice`, the program displayed `true` on the first `console.log` call, can you tell me what the program displayed on the second `console.log` call?

### **Listing 1.2 Apparently simple code**

```
class Member {
    isBlocked;

    displayBlockedStatusTwice() {
        console.log(this.isBlocked);
        console.log(this.isBlocked);
    }
}

member.displayBlockedStatusTwice();
```

The correct answer is: in a single threaded environment, it displays `true` while on a multi threaded environment it's unpredictable.

Indeed, in a multi threaded environment, between the two `console.log` calls, there could be a context switch and the state of the object could be changed (e.g. a librarian unblocked the member).

In fact, with a slight modification, the same kind of code unpredictability could occur even in a single threaded environment like JavaScript, when a data is modified via asynchronous code (see the section about Principle #3 in Appendix A).

The difference between the two code snippets is that:

- In the first snippet, we access twice a boolean value which is a primitive value
- In the second snippet, we access twice a member of an object

**TIP**

**When data is mutable, code is unpredictable.**

This unpredictable behavior of the second snippet is one of the annoying consequences of the fact that in OOP, unlike primitive types which are usually immutable, object members are mutable.

One way to solve this problem in OOP is to protect sensitive code with concurrency safety mechanism like mutexes, but that introduces issues like a performance hit and a risk of deadlocks.

We will see later in the book that DOP treats every piece of data in the same way: both primitive types and collection types are immutable values. This "value treatment for all citizens" brings serenity to DOP developers' minds, and more brain cells are available to handle the interesting pieces of the applications they build.

**TIP**

**Data immutability brings serenity to DOP developers' minds.**

### 1.3.3 Not trivial data serialization

Theo is really tired and he falls asleep at his desk...

He's having dream. In this dream, Nancy asks him to make the Global Library Management system accessible via a REST API using JSON as a transport layer.

Theo has to implement a `/search` endpoint that receives a query in JSON format and return results in JSON format.

An input example of the `/search` endpoint is shown in [1.3](#).

#### **Listing 1.3 A JSON input of the `/search` endpoint**

```
{
    "searchCriteria": "author",
    "query": "albert"
}
```

An output example of the `/search` endpoint is shown in [1.4](#).

#### **Listing 1.4 A JSON output of the `/search` endpoint**

```
[
{
    "title": "The world as I see it",
    "authors": [
        {
            "fullName": "Albert Einstein"
        }
    ],
{
    "title": "The Stranger",
    "authors": [
        {
            "fullName": "Albert Camus"
        }
    ]
}
]
```

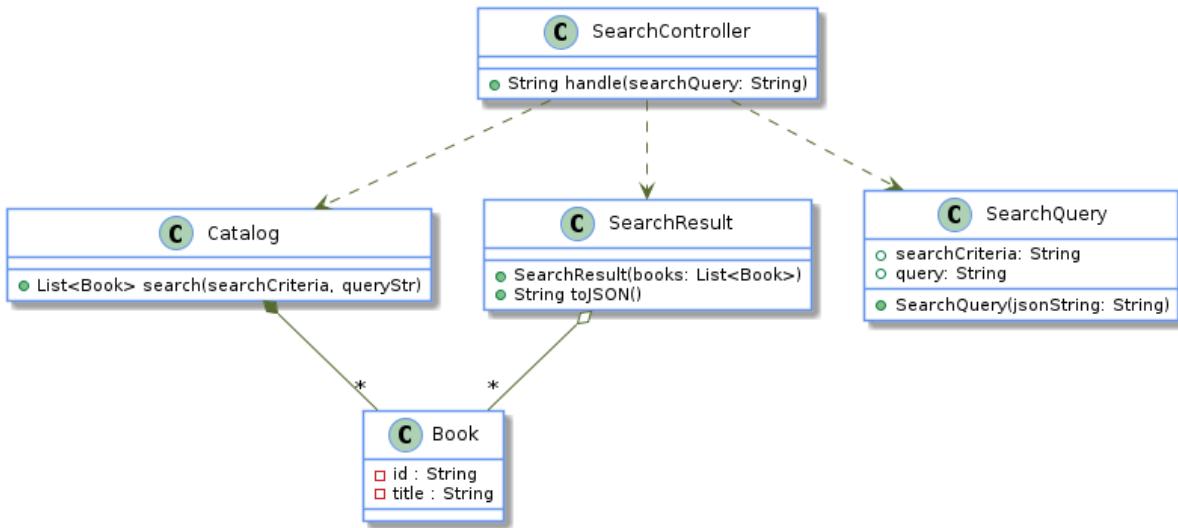
Theo would probably implement the `/search` endpoint by creating three classes similarly to what is shown in [1.14](#) (Not surprising: everything in OOP has to be wrapped in a class. Right?):

1. `SearchController` is responsible for handling the query
2. `SearchQuery` converts the JSON query string into data
3. `SearchResult` converts the search result data into a JSON string

The `SearchController` would have a single `handle` method with the following flow:

1. Create a `SearchQuery` object from the JSON query string
2. Retrieve `searchCriteria` and `queryStr` from the `SearchQuery` object
3. Call the `search` method of the `Catalog` with `searchCriteria` and `queryStr`

- and receives `books : List<Book>`
4. Create a `SearchResult` object with `books`
  5. Convert the `SearchResult` object to a JSON string



**Figure 1.14 The class diagram for `SearchController`**

What about other endpoints, for instance allowing librarians to add book items through `/add-book-item`?

Well, Theo would have to repeat the exact same process and create 3 classes:

1. `AddBookItemController` to handle the query
2. `BookItemQuery` to convert the JSON query string into data
3. `BookItemResult` to convert the search result data into a JSON string

The code that deals with JSON deserialization that Theo wrote previously in `SearchQuery` would have to be rewritten in `BookItemQuery`. Same thing for the code that deals with JSON serialization that he wrote previously in `SearchResult`: it would have to be rewritten in `BookItemResult`.

The bad news is that he would have to repeat the same process for every endpoint of the system. Each time Theo encounters a new kind of JSON input or output, he would have to create a new class and write code.

Suddenly, Theo wakes up and realizes that Nancy never asked for JSON. All of the above was a dream, a really bad dream...

**TIP**

In OOP, data serialization is difficult.

It's quite frustrating that handling JSON serialization and deserialization in OOP requires the addition of so many classes writing so much code - again and again!

The frustration grows when you consider that serializing a search query, a book item query, or *any* query is quite similar.

It comes down to:

1. Go over data fields
2. Concatenate the name of the data fields and the value of the data field (separated by a comma)

Why is such a simple thing so hard to achieve in OOP?

In OOP, data has to follow a rigid shape (defined in classes) which means that data is locked in members. There is no simple way to access data generically.

**TIP**

**In OOP, data is locked in classes as members**

We will refine later what we mean by generic access to the data and we will see how DOP provides a generic way to handle JSON serialization and deserialization. Until then, you will have to continue suffering. But at least you are starting to become aware of this suffering and you know that this suffering is avoidable.

**WARNING**

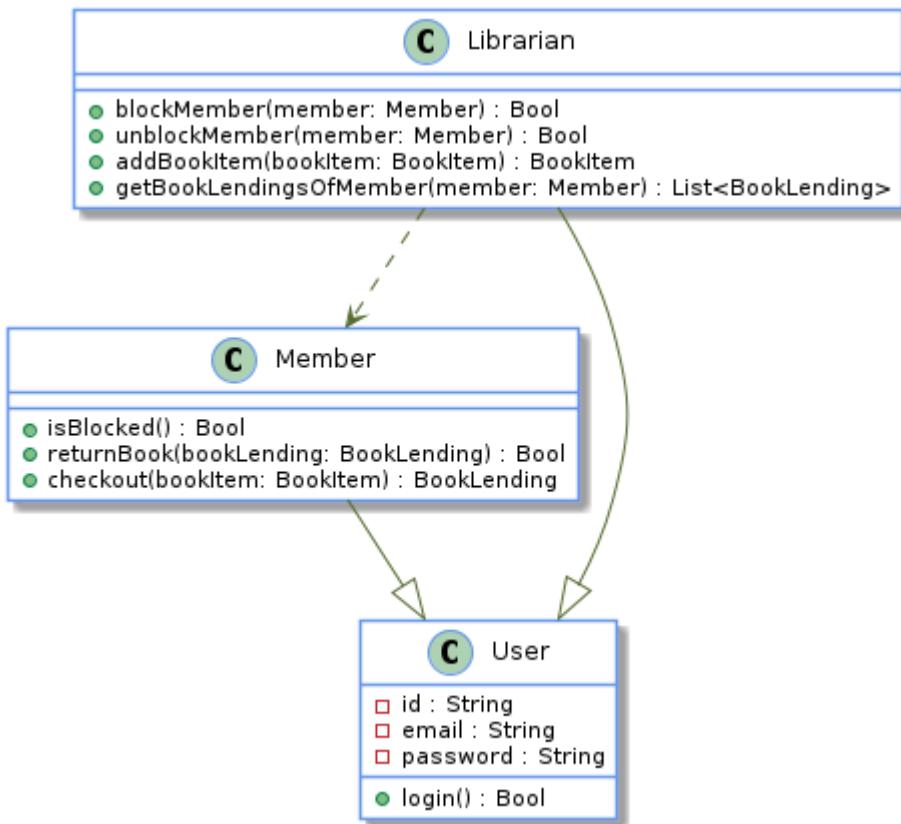
**Most OOP programming languages alleviate a bit the difficulty involved the conversion from and to JSON. It either involves reflection (which is definitely a complex thing) or code verbosity.**

### 1.3.4 Complex class hierarchies

One way to avoid writing the same code twice in OOP involves class inheritance. Indeed, when every requirement of the system is known up front, you design your class hierarchy in such a way that classes with common behavior derive from a base class.

An example of this pattern is shown in [1.15](#), that focuses on the part of our class diagram that deals with members and librarians. Both `Librarians` and `Members` need the ability to login and they inherit this ability from the `User` class.

So far so good.



**Figure 1.15** The part of the class diagram that deals with members and librarians

But when new requirements are introduced after the system is implemented it's a completely different story.

It's Monday March 29th 11:00 AM, two days are left before the deadline (Wednesday at midnight) and Nancy calls Theo with an urgent request.

Theo is not sure if it's dream or reality. He pinches himself and he can feel the jolt. It's definitely reality!

**NANCY:** How is the project doing?

**THEO:** Fine, Nancy. We're on schedule to meet the deadline. We're running our last round of regression tests now.

**NANCY:** Fantastic! It means we have time for adding a tiny feature to the system. Right?

**THEO:** Depends what you mean by *tiny*.

**NANCY:** We need to add VIP members to the system.

**THEO:** What do you mean by VIP members?

**NANCY:** VIP members are members that are allowed to add book items to the library by themselves.

**THEO:** Hmm...

**NANCY:** What?

**THEO:** That's not a tiny change!

**NANCY:** Why?

I am asking you the same question Nancy asked Theo: Why is adding VIP members to your system not a tiny task?

After all, Theo has already written the code that allows librarians to add book items to the library: it's in `Librarian::addBookItem`.

What prevents him from reusing this code for VIP members?

The reason is that in OOP, the code is locked into classes as methods.

**TIP**

**In OOP, code is locked into classes.**

*VIP members are members that are allowed to add book items to the library by themselves.*

Theo decomposes the customer requirements into two pieces:

1. VIP members are members
2. VIP members are allowed to add book items to the library by themselves

He decides that he needs a new class `VIPMember`.

For requirement #1, it sounds reasonable to make `VIPMember` derive from `Member`.

However, handling requirement #2 is more complex. He cannot make `VIPMember` derive from `Librarian` because the relationship between `VIPMember` and `Librarian` is not linear:

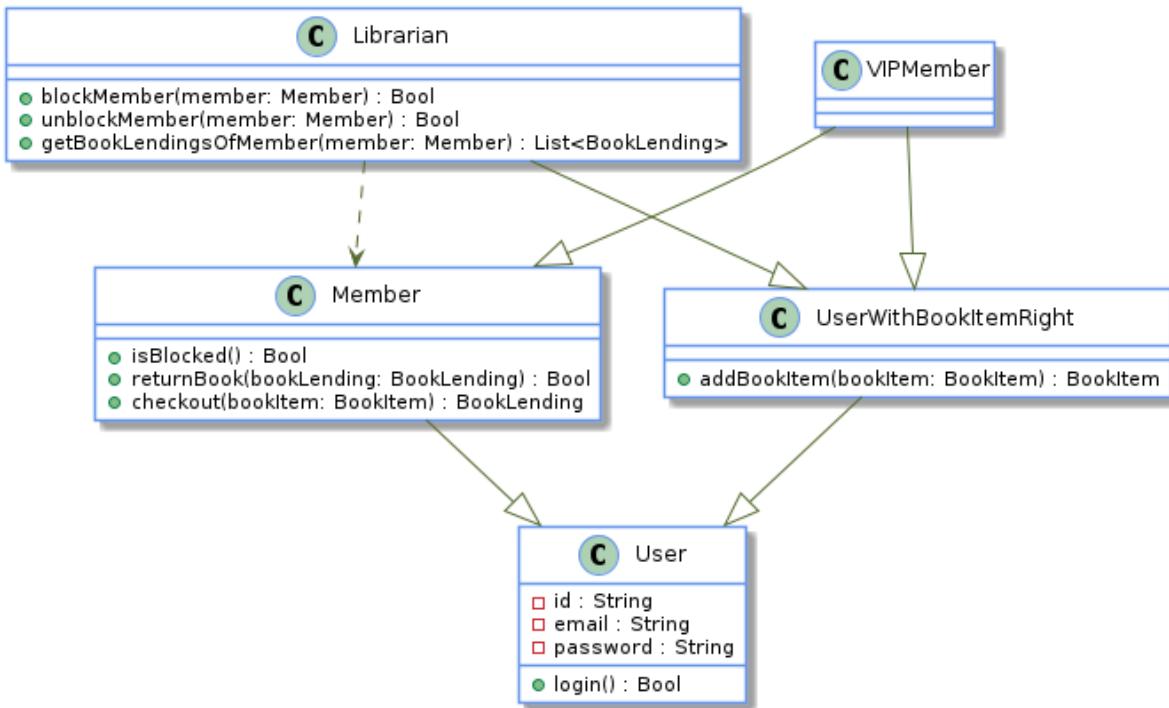
1. On one hand, VIP members are like librarians in that they are allowed to add book items
2. On the other hand, VIP members are *not* like librarians in that they are not allowed to block members or list the book lendings of a member

The problem is that the code that adds book items is locked in the `Librarian` class. There is no way for the `VIPMember` class to use this code.

One possible solution that makes the code of `Librarian::addBookItem` available to both `Librarian` and `VIPMember`, is shown in [1.16](#). Here are the changes to the previous class

diagram:

1. A base class `UserWithBookItemRight` that extends `User`
2. Move `addBookItem` from `Librarian` to `UserWithBookItemRight`
3. Both `VIPMember` and `Librarian` extend `UserWithBookItemRight`



**Figure 1.16 A class diagram for a system with VIP members**

It wasn't easy but Theo just managed to handle it on time, thanks to an all-nighter, coding on his laptop. He was even able to add new tests to the system and run the regression tests again.

However, he was so excited that he didn't pay attention to the diamond problem ([https://en.wikipedia.org/wiki/Multiple\\_inheritance](https://en.wikipedia.org/wiki/Multiple_inheritance)) `VIPMember` introduced in his class diagram due to multiple inheritance, (`VIPMember` extends both `Member` and `UserWithBookItemRight` who both extend `User`)

We are Wednesday March 31 10:00 AM, 14 hours before the deadline and Theo calls Nancy to tell her the good news:

**THEO:** We were able to add VIP members to the system on time, Nancy.

Fantastic! I told you it was a tiny feature.

**THEO:** Hmm...

**NANCY:** Look, I was going to call you anyway. I just finished a meeting with my business partner and we realized that we need another tiny feature before the launch. Will you be able to

handle it before the deadline?

**THEO:** Again, it depends what you mean by *tiny*.

**NANCY:** We need to add Super members to the system.

**THEO:** What do you mean by Super members?

**NANCY:** Super members are members that are allowed list the book lendings of other members.

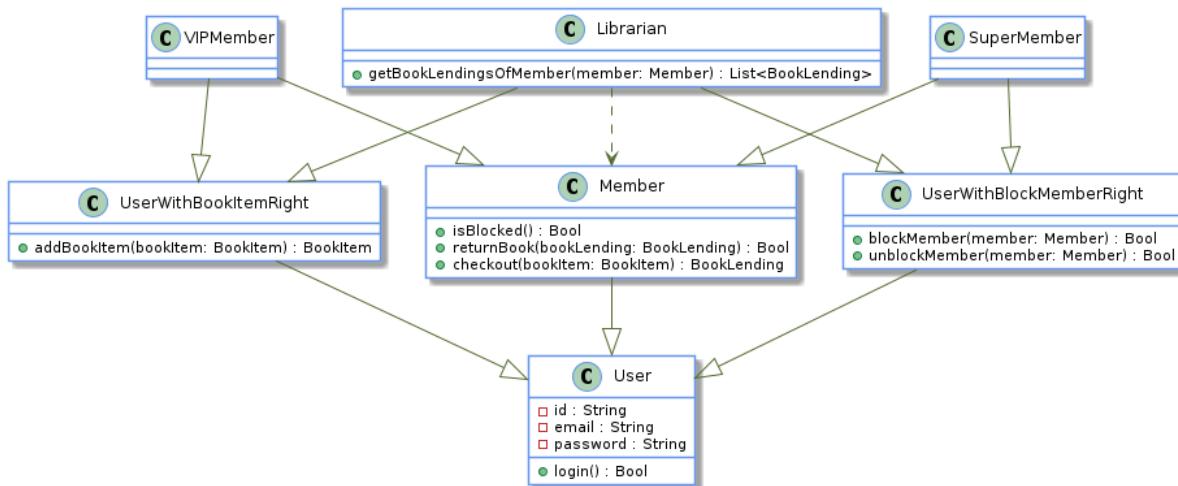
**THEO:** Hmm...

**NANCY:** What?

**THEO:** That's not a tiny change!

**NANCY:** Why?

Like with VIP members, adding Super members to the system requires changes to Theo's class hierarchy. The solution Theo came up with is shown in [1.17](#).



**Figure 1.17 A class diagram for a system with Super members and VIP members**

The addition of Super members has made the system very complex. Theo suddenly notices that he has 3 diamonds in his class diagram: not gemstones but 3 Deadly Diamonds of Death, as OOP developers sometimes name the ambiguity that arises when a class **D** inherits from two classes **B** and **C** that both inherit from **A**!

He tries to avoid the diamonds by transforming the **User** class into an interface and using the "Composition over Inheritance" Design Pattern.

But with the stress of the deadline looming, he isn't able to use all of his brain cells.

In fact, the system has become so complex he's unable to deliver the system by the deadline.

Theo tells himself that he should have used composition instead of class inheritance. But it's too late now.

**TIP**

In OOP, prefer composition over class inheritance.

Theo calls Nancy in order to explain her the situation at 10:00 PM, two hours before the deadline:

**THEO:** Look Nancy, we really did our best, but we won't be able to add Super members to the system before the deadline.

**NANCY:** No worries, my business partner and I decided to get rid of this feature for now. We'll add it later.

With mixed feelings of anger and relief, you say:

**THEO:** I guess that means we're ready for the launch tomorrow morning.

**NANCY:** Yes. We'll offer this new product for a month or so, and if we get good market traction we'll move forward with a bigger project.

**THEO:** Cool. Let's be in touch in a month then. Good luck on the launch!

## 1.4 Summary

- Complex in the context of this book means: hard to understand.
- We use the terms code and behavior interchangeably.
- DOP stands for Data-Oriented Programming
- OOP stands for Object-Oriented Programming
- FP stands for Functional Programming
- In a composition relation, when one object dies, the other one dies also
- A composition relation is represented by a plain diamond at one edge and an optional star at the other edge.
- In an association relation, each object has an independent life cycle
- A many-to-many association relation is represented by an empty diamond and a star at both edges.
- Dashed arrows are for usage relations: for instance, when a class uses a method of another class.
- Plain arrows with empty triangles represent class inheritance, where the arrow points towards the superclass.
- The design presented doesn't pretend to be the smartest OOP design: experienced OOP developers would probably leverage a couple of design patterns and suggest a much better design.
- Traditional OOP systems tend to increase system complexity, in the sense that OOP systems tend to be hard to understand.
- In traditional OOP, code and data are mixed together in classes: data as members and code as methods.
- In traditional OOP, data is mutable.
- The root cause of the increase in complexity is related to the mixing of code and data together into objects.
- When code and data are mixed, classes tend to be involved in many relations.
- When objects are mutable, extra thinking is required in order to understand how the code behaves.
- When objects are mutable, explicit synchronization mechanisms are required on multi-threaded environments.
- When data is locked in objects, data serialization is not trivial.
- When code is locked in classes, class hierarchies tend to be complex.
- A system where every class is split into two independent parts, code and data, is simpler than a system where code and data are mixed.
- A system made of multiple simple independent parts is less complex than a system made of a single complex part.
- When data is mutable, code is unpredictable.
- Strategic use of Design Patterns can help mitigate complexity in traditional OOP to some degree.
- Data immutability brings serenity to DOP developers' minds.
- Most OOP programming languages alleviate a bit the difficulty involved the conversion from and to JSON. It either involves reflection (which is definitely a complex thing) or code verbosity.

- In traditional OOP, data serialization is difficult.
- In traditional OOP, data is locked in classes as members
- In traditional OOP, code is locked into classes.
- DOP reduces complexity by rethinking data.
- DOP is compatible both with OOP and FP.

**Table 1.2 Aspects of Object Oriented programming and their impact on system complexity**

Aspect	Impact on complexity
Code and data are mixed	Classes tend to be involved in many relations
Objects are mutable	Extra thinking when reading code
Objects are mutable	Explicit synchronization on multi-threaded environments
Data is locked in objects	Data serialization is not trivial
Code is locked in classes	Class hierarchies are complex

# *Separation between code and data*



## This chapter covers

- The benefits of separating code from data
- Designing a system where code and data are separate
- Implementing a system that respects the separation between code and data.

## 2.1 A whole new world

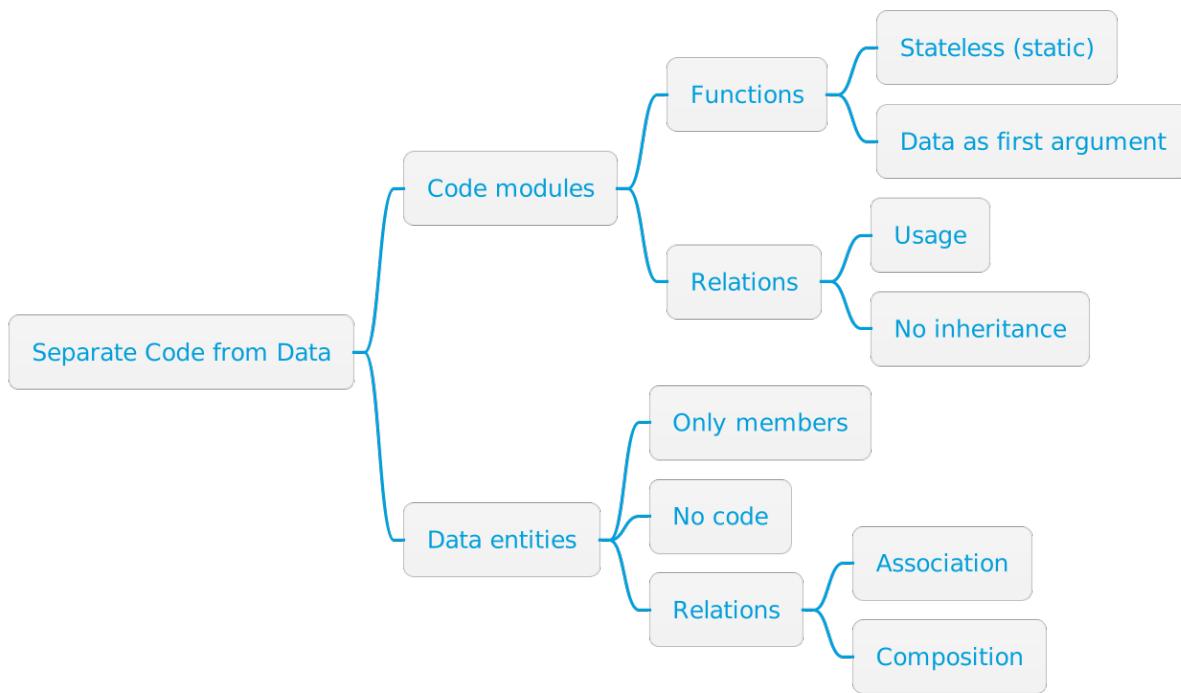
The first insight of Data Oriented Programming (DOP) is that we can decrease the complexity of our systems by separating code from data. Indeed, when code is separated from data, our systems are made of two main pieces that can be thought about separately: Data entities and Code modules.

This chapter is a deep dive in the first principle of Data-Oriented Programming:

**NOTE**

**Principle #1: Separate code from data in a way that the code resides in functions whose behavior doesn't depend on data that is somehow encapsulated in the function's context.**

The details of Principle #1 are summarized in [2.1](#).



**Figure 2.1 The summary of Principle #1: Separate code from data**

We illustrate the separation between code and data in the context of the Library Management system that we introduced in chapter 1, and we unveil the benefits that this separation brings to the system:

1. The system is simple: it is easy to understand
2. The system is flexible and extensible: quite often it requires no design changes to adapt to changing requirements

This chapter focuses on the design of the code in system where code and data are separate. In the next chapter, we'll focus on the design of the data. As we progress in the book, we'll discover other benefits of separating code from data.

## 2.2 The two parts of a DOP system

While Theo is driving back home, he asks himself whether the Klafim project was a success or not. Sure, he was able to satisfy the customer, but it was more luck than brains. He wouldn't have made it on time if Nancy had decided to keep the Super members feature. Why was it so complicated to add tiny features to the system? Why was the system he built so complex? There should be a way to build more flexible systems!

The next morning, Theo asks on Hacker News and on Reddit for ways to reduce system complexity and build flexible systems. Some folks mention using different programming languages, others talk about advanced design patterns. Finally, Theo's attention gets captured by a comment from a user named Joe who mentions "Data-Oriented programming" and claims that

its main goal is to reduce system complexity. Theo has never heard this term before. Out of curiosity he decides to contact Joe by email.

What a coincidence! Joe lives in San Francisco too. Theo invites him to a meeting in his office.

Joe is a 40-year old developer. He'd been a Java developer for nearly decade before adopting Clojure around 7 years ago.

When Theo tells Joe about the Library Management System he designed and built, and about his struggles to adapt to changing requirements, Joe is not surprised.

Joe tells Theo that the systems that he and his team have built in Clojure over the last 7 years are less complex and more flexible than the systems he used to build in Java. According to Joe, the systems they build now tend to be much simpler because they follow the principles of Data-Oriented Programming (DOP).

**THEO:** I've never heard of "Data-Oriented Programming". Is it a new concept?

**JOE:** Yes and no. Most of the foundational ideas of DOP are well-known to programmers as best practices. But the novelty of DOP is that it combines them into a cohesive whole.

**THEO:** That's a bit abstract for me. Can you give me an example?

**JOE:** Sure. Take for instance, the first insight of DOP: it's about the relationships between code and data.

**THEO:** You mean the encapsulation of data in objects?

**JOE:** Actually, DOP is *against* data encapsulation.

**THEO:** Why is that? I thought data encapsulation was a positive programming paradigm.

**JOE:** Data encapsulation has both merits and drawbacks: Think about the way you designed the Library Management System. According to DOP, the main cause of complexity and inflexibility in systems is that code and data are mixed together (in objects).

**TIP**

**DOP is against data encapsulation.**

**THEO:** It sounds similar to what I've heard about Functional Programming (FP). So if I want to adopt DOP, do I need to get rid of OOP and learn FP?

**JOE:** No. DOP principles are language agnostic: they can be applied in both OOP and FP languages.

**THEO:** That's a relief! I was afraid that you were going to teach me about monads, algebraic

data types, and high order functions.

**JOE:** None of that is required in DOP.

**TIP**

**DOP principles are language agnostic.**

**THEO:** What does the separation between code and data look like in DOP, then?

**JOE:** Data is represented by data entities that hold members only. Code is aggregated into modules where all functions are stateless.

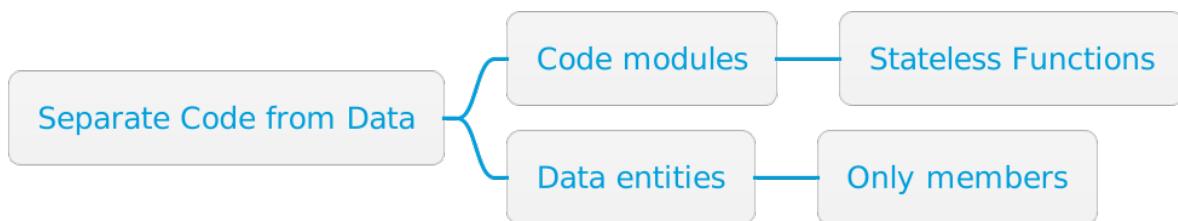
**THEO:** What do you mean by stateless functions?

**JOE:** Instead of having the state encapsulated in the object, the data entity is passed as an argument.

**THEO:** I don't get that.

**JOE:** Here, let's make it visual. [2.2](#)

Joe steps up to a whiteboard and quickly draws a diagram.



**Figure 2.2 The separation between code and data**

**THEO:** It's still not clear

**JOE:** It will become clearer when I show you how it looks in the context of your Library Management System.

**THEO:** OK. Shall we start we code or with data?

**JOE:** Well, it's Data-Oriented Programming. Let's start with Data!

## 2.3 Data entities

In DOP, we start the design process by discovering the data entities of our system.

**JOE:** What are the data entities of your system?

**THEO:** What do you mean by *data entities*?

**JOE:** I mean the parts of your system that hold information.

**NOTE**

**Data entities are the parts of your system that hold information**

**THEO:** Well, it's a library management system. We have books and members.

**JOE:** Of course. But there are more: One way to discover the data entities of a system is to look for nouns and noun phrases in the requirements of the system.

Theo looks at Nancy's requirement napkin and highlights the nouns and noun phrases that seem to represent data entities:

**SIDE BAR**

**Highlighting terms in the requirements that correspond to data entities**

- Two kinds of users: library members and librarians
- Users log in to the system via email and password.
- Members can borrow books
- Members and librarians can search books by title or by author
- Librarians can block and unblock members (e.g. when they are late in returning a book)
- Librarians can list the books currently lent by a member
- There could be several copies of a book

**JOE:** Excellent. Can you see a natural way to group the entities?

**THEO:** Not sure, but it seems to me that *users*, *members* and *librarians* form a group while *books*, *authors* and *book copies* form another group.

**JOE:** Sounds good to me. What would you call each group?

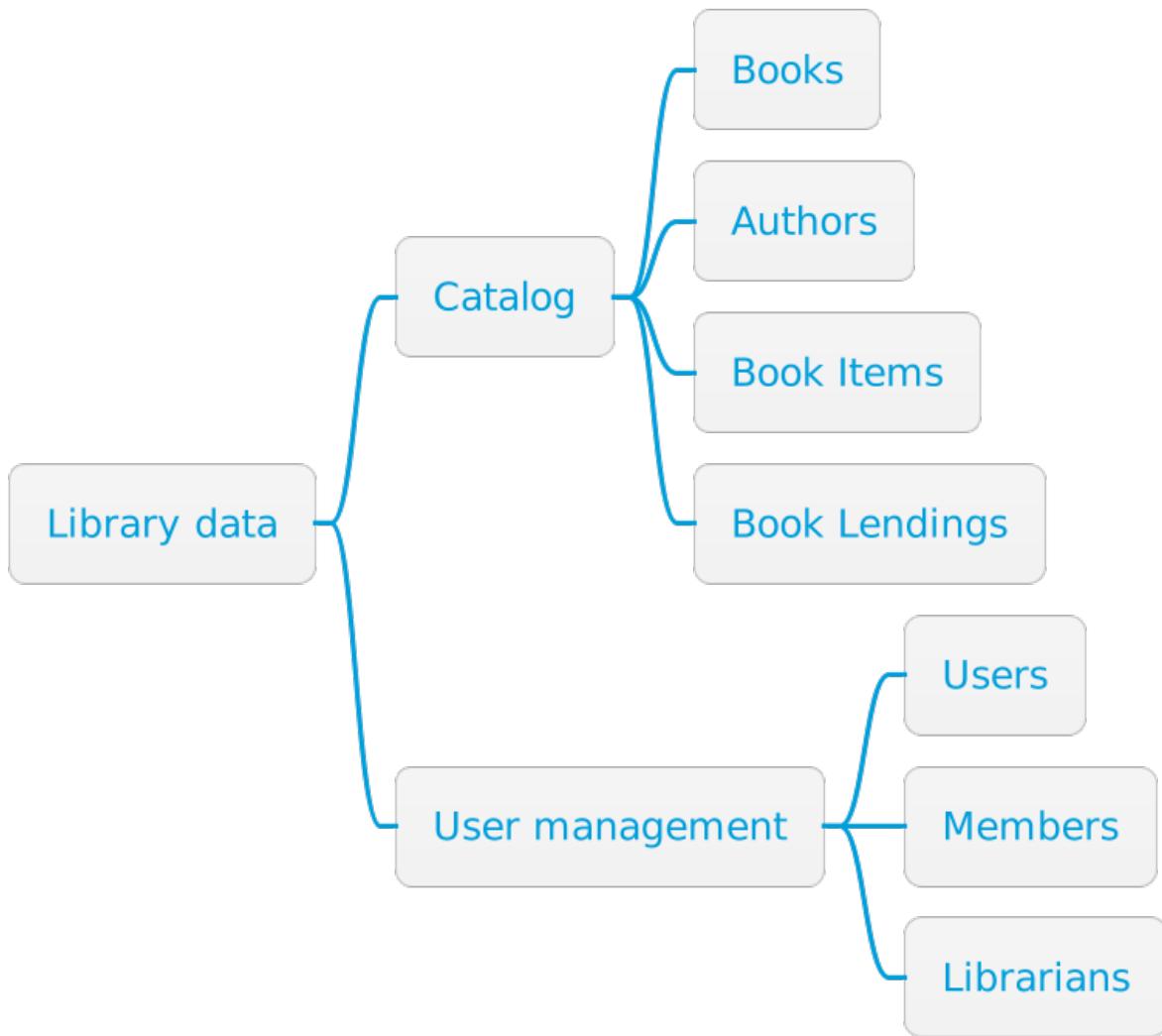
**THEO:** "User management" for the first group and "Catalog" for the second group.

**SIDE BAR****The data entities of the system organized in a nested list**

- The catalog data
  - Data about books
  - Data about authors
  - Data about book items
  - Data about book lendings
- The user management data
  - Data about users
  - Data about members
  - Data about librarians

**THEO:** I'm not sure about the relationships between books and authors: should it be association or composition?

**JOE:** Don't worry too much about the details for the moment. We'll refine our data entities design later. For now, let's visualize the two groups in a mind map.



**Figure 2.3 The data entities of the system organized in a mind map**

The most precise way to visualize the data entities of a DOP system is to draw a data entity diagram with different arrows for association and composition. We will come back to data entity diagrams later.

**TIP**

Discover the data entities of your system and group them into high level groups, either as a nested list or as a mind map.

We will get deeper into the design and representation of data entities in the next chapter. For now, let's simplify and say that the data of our library system is made of two high level groups: User Management and Catalog.

## 2.4 Code modules

The second step of the design process in DOP is to define the code modules.

**JOE:** Now that you have identified the data entities of your system and have grouped them into high level groups, it's time to think about the code part of your system.

**THEO:** What do you mean by *code part*?

**JOE:** One way to think about it is to identity the functionality of your system.

Theo looks again at Nancy's requirement napkin and this time he highlights the verb phrases that represent functionality:

**SIDE BAR**

**Highlighting terms in the requirements that correspond to functionality**

- Two kinds of users: library members and librarians
- Users log into the system via email and password.
- Members can borrow books
- Members and librarians can search books by title or by author
- Librarians can block and unblock members (e.g. when they are late in returning a book)
- Librarians can list the books currently lent by a member
- There could be several copies of a book

In addition, it's obvious that members can also return a book. Moreover, there should be a way to detect whether a user is a librarian or not.

Theo lists the functionality of the system.

**SIDE BAR**

**The functionality of the system**

1. Search a book
2. Add a book item
3. Block a member
4. Unblock a member
5. Login a user into the system
6. List the books currently lent by a member
7. Borrow a book
8. Return a book
9. Check whether a user is a librarian

**JOE:** Excellent! Now, tell me what functionality needs to be exposed to the outside world?

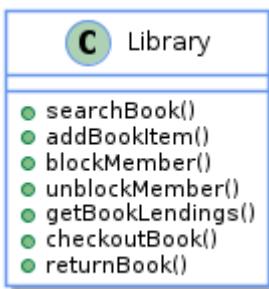
**THEO:** What do you mean by *exposed* to the outside world\_?

**JOE:** Imagine that the library management system exposes an API over HTTP: what functionality would be exposed by the HTTP endpoints?

**THEO:** All system functionality would be exposed except checking to see if a user is a librarian.

**JOE:** OK. Now give each exposed function a short name and gather them together in a module box called `Library`

It takes Theo less than a minute: [2.4](#) shows the module that contains the exposed functions of the Library.



**Figure 2.4** The `Library` module contains the exposed functions of the Library management system

**TIP**

The first step in designing the code part of a DOP system is to aggregate the exposed functions into a single module.

**JOE:** Beautiful. You just created your first code module.

**THEO:** To me it looks like a class: What's the difference between a *module* and a *class*?

**JOE:** A module is an aggregation of functions. In OOP, a module is represented by a class but in other programming languages, it might be a *package* or a *namespace*.

**THEO:** I see.

**JOE:** The important thing about DOP code modules is that they contain only stateless functions.

**THEO:** You mean like static methods in Java?

**JOE:** Yes. And the classes of these static methods should not have any data members!

**THEO:** So how do the functions know what piece of information they operate on?

**JOE:** We pass it as the first argument to the function.

**THEO:** Okay. Can you give me an example?

Joe takes a look at the list of functions of the `Library` module in [2.4](#).

**JOE:** Let's take for example `getBookLendings`: in classic OOP, what would its arguments be?

**THEO:** A librarian ID and a member ID.

**JOE:** In traditional OOP, `getBookLendings` would be a method of a `Library` class that receives two arguments: `librarianId` and `memberId`

**THEO:** Yeap.

**JOE:** Now comes the subtle part: in DOP, `getBookLendings` is part of the library module and it receives the `LibraryData` as an argument.

**THEO:** Could you show me what you mean?

**JOE:** Sure.

Joe gets closer to Theo's keyboard and starts typing.

**JOE:** Here's an example of what a class method looks like in OOP.

```
class Library {
    catalog
    userManagement

    getBookLendings(userId, memberId) {
        // accesses library state via this.catalog and this.userManagement
    }
}
```

**THEO:** Right. The method accesses the state of the object—in our case the library data—via `this`.

**JOE:** Would you say that the object's state is an argument of the object's methods?

**THEO:** I'd say that the object's state is an implicit argument to the object's methods.

**TIP**

In traditional OOP, the state of the object is an implicit argument to the methods of the object.

**JOE:** Well, in DOP we pass data as an explicit argument. The signature of `getBookLendings` would look like this.

Joe shows Theo the code in [2.1](#).

### Listing 2.1 The signature of `getBookLendings`

```
class Library {
    static getBookLendings(libraryData, userId, memberId) {
    }
}
```

**JOE:** The state of the library is stored in `libraryData` and `LibraryData` is passed to the `getBookLendings` static method as an explicit argument.

**THEO:** Is that a general rule?

**JOE:** Absolutely! The same rule applies to the other functions of the library module and to other modules. All of them are stateless: they receive the library data they manipulate as an argument.

**TIP**

In DOP, functions of a code module are stateless: they receive the data they manipulate as an explicit argument, usually the first argument.

**IMPORTANT** A module is an aggregation of functions. In DOP, the module functions are stateless.

**THEO:** It reminds me of Python and the way the `self` argument appears in method signatures as in [2.2](#).

### Listing 2.2 In Python, the object appears as an explicit argument in method signatures

```
class Library:
    catalog = {}
    userManagement = {}

    def getBookLendings(self, userId, memberId):
        # accesses library state via self.catalog and self.userManagement
```

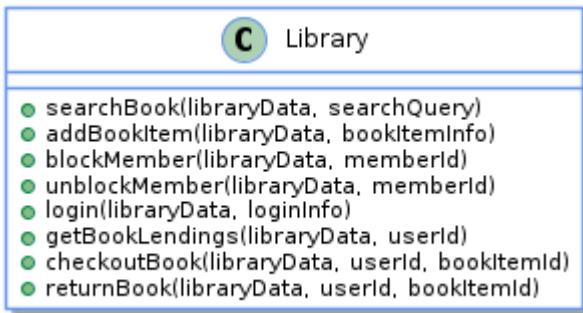
**JOE:** Indeed. But the difference I'm talking about is much deeper than a syntax change. It's about the fact that data lives outside the modules.

**THEO:** I got that. As you said, module functions are stateless.

**JOE:** Exactly. Would you like to try and apply this principle across the whole Library module?

**THEO:** Sure.

Theo refines the design of the library module by including the details about functions' arguments. He gets the diagram in [2.5](#).



**Figure 2.5 The Library module with the function arguments**

**JOE:** Perfect. Now, we're ready to tackle the high level design of our system.

**THEO:** What's a high level design in DOP?

**JOE:** The definition of modules and the interaction between them.

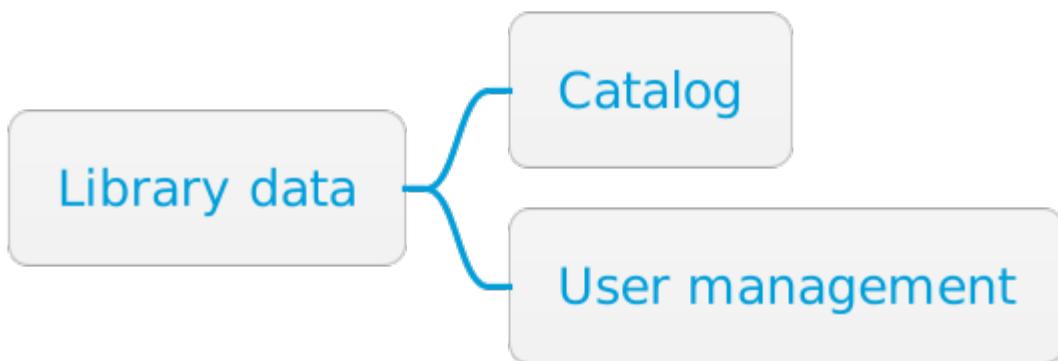
**THEO:** I see. Is there any guideline to help me define the modules?

**JOE:** Definitely. The high level modules of the system correspond to the high level data entities.

**THEO:** You mean the data entities that appear in the data mind map?

**JOE:** Exactly!

Theo looks again at the data mind map in [2.6](#) and he focuses on the high level data entities: Library, Catalog and User management.



**Figure 2.6 A mindmap of high level data entities of the Library management system**

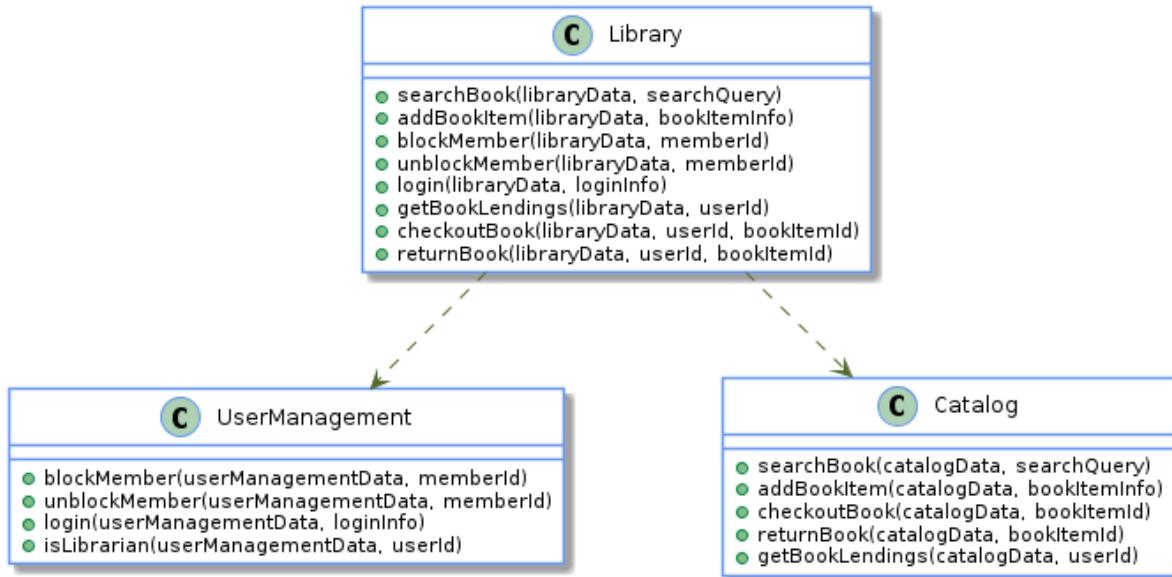
It means that in the system, beside the Library module, we have two high level modules:

1. Catalog module that deals with catalog data
2. UserManagement module that deals with user management data

Then Theo draws the high level design of the library management system, adding Catalog and UserManagement modules:

- Functions of Catalog receive catalogData as first argument
- Functions of UserManagement receive userManagementData as first argument

Here is the diagram:



**Figure 2.7 The modules of the Library management system with the function arguments**

It's not 100% clear for Theo at this point how the data entities get passed between modules. For the moment, he thinks of libraryData as a class with two members:

- catalog that holds the catalog data
- userManagement that holds the user management data

The functions of Library share a common pattern:

1. They receive libraryData as an argument
2. They pass libraryData.catalog to functions of Catalog
3. They pass libraryData.userManagement to functions of UserManagement

Later on in this chapter we'll see the code for some functions of the Library module.

**TIP**

**The high level modules of a DOP system correspond to the high level data entities.**

## 2.5 DOP systems are easy to understand

Theo takes a look at the two diagrams that represent the high level design of his system:

1. The data entities in the data mind map from [2.8](#)
2. The code modules in the module diagram from [2.9](#)

A bit perplexed, Theo asks Joe:

**THEO:** I'm not sure that this system is better than a traditional OOP system, where objects encapsulate data.

**JOE:** The main benefit of a DOP system over a traditional OOP systems is that it's easier to understand.

**THEO:** What makes it easier to understand?

**JOE:** The fact that the system is split cleanly into code modules and data entities.

**THEO:** How does that help?

**JOE:** When you try to understand the data entities of the system, you don't have to think about the details of the code that manipulates the data entities.

**THEO:** So when I look at the data mind map of my library management system, I can understand it on its own?

**JOE:** Exactly. And similarly, when you try to understand the code modules of the system, you don't have to think about the details of the data entities manipulated by the code. There is a clear separation of concerns between the code and the data.

Theo looks again at the data mind map in [2.8](#), and he has kind of an 'Aha' moment:

*Data lives on its own!*

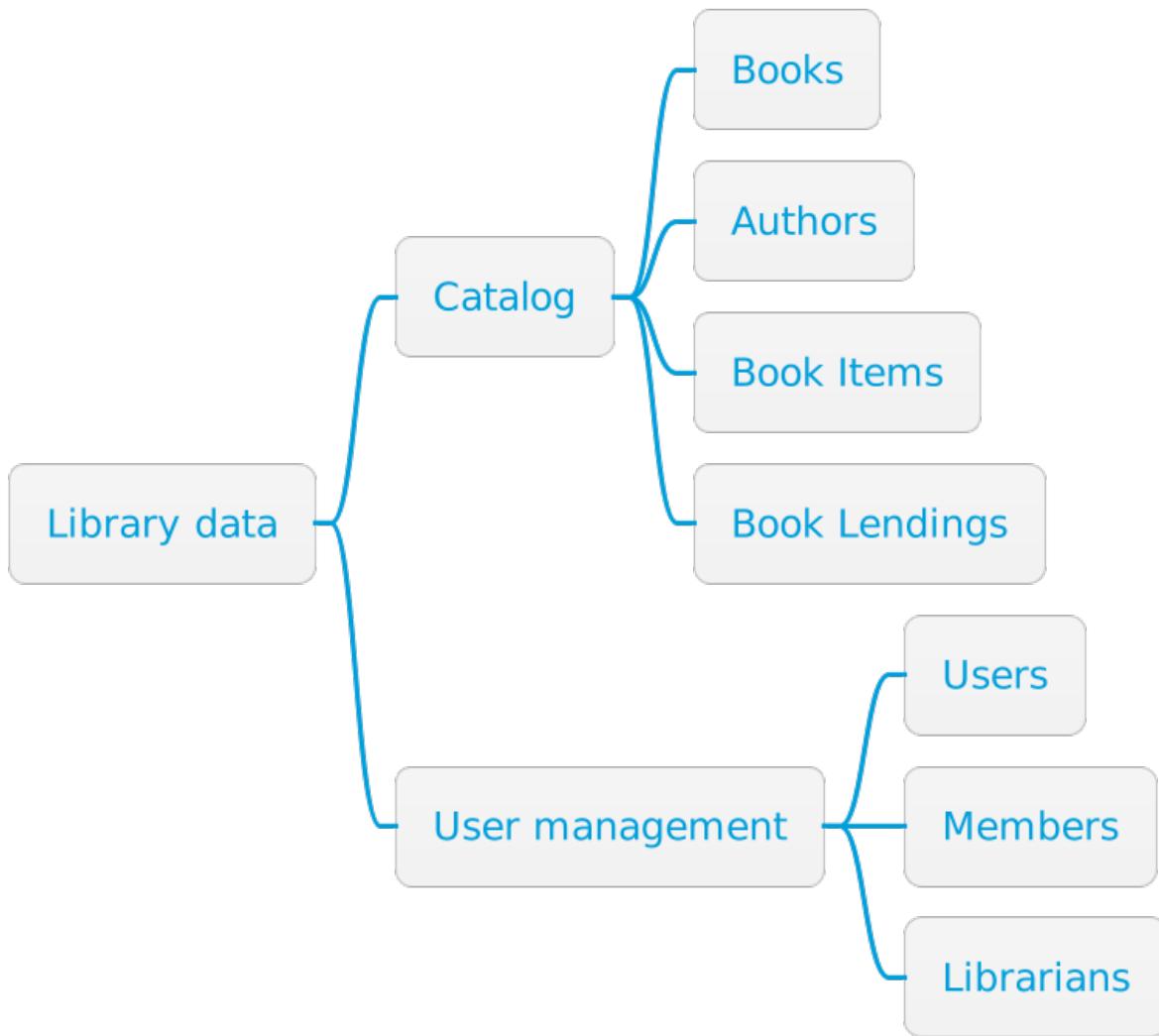


Figure 2.8 A data mindmap of the Library management system

**IMPORTANT** A DOP system is easier to understand because the system is split into two parts: data entities and code modules.

Now Theo looks at the module diagram in [2.9](#) and feels a bit confused:

- On one hand, the module diagram looks similar to the class diagrams from classic OOP: boxes for classes and arrows for relations between classes.
- On the other hand, the code module diagram looks much simpler than the class diagrams from classic OOP, but he cannot explain why.

Theo asks Joe for a clarification.

**THEO:** The module diagram seems much simpler than the class diagrams I am used to in OOP. I feel it, but can't put it into words.

**JOE:** The reason is that module diagrams have constraints.

**THEO:** What kind of constraints?

**JOE:** Constraints on the functions we saw before: All the functions are static (stateless). But also constraints on the relationships between the modules.

**TIP**

All the functions in a DOP module are stateless.

**THEO:** In what way are the relationships between modules constrained?

**JOE:** There is a single kind of relation between DOP modules: the usage relation. A module uses code from another module. No association, no composition and no inheritance between modules. That's what make a DOP module diagram easy to understand.

**THEO:** I understand why there is no association and no composition between DOP modules: after all, association and composition are data relations. But why no inheritance relation? Does it mean that in DOP is against polymorphism?

**JOE:** That's a great question. The quick answer is that in DOP, we achieve polymorphism with a different mechanism than class inheritance. We will talk about it some day!

**NOTE**

For a discussion of polymorphism in DOP, see Chapter 13.

**THEO:** Now, you've piqued my curiosity: I thought inheritance was the only way to achieve polymorphism.

**TIP**

The only kind of relationship between DOP modules is the usage relation.

Theo looks again at the module diagram in 2.9 and now he not only feels that this diagram is simpler than traditional OOP class diagrams, he understands *why* it's simpler: All the functions are static and all the relation between modules are of type *usage*.

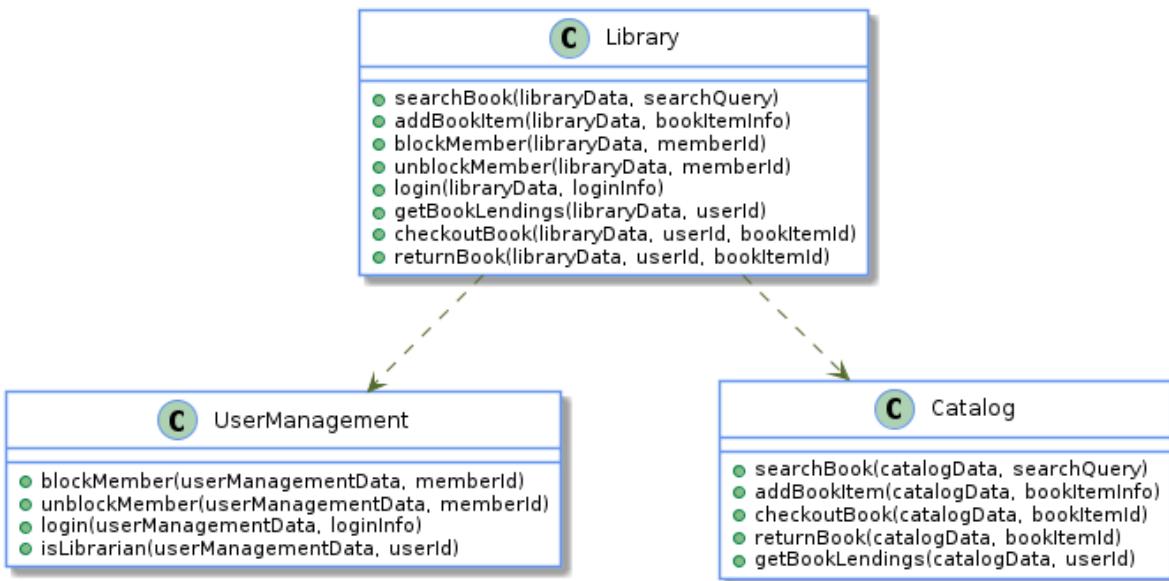


Figure 2.9 The modules of the Library management system with the function arguments

Table 2.1 What makes each part of a DOP system easy to understand

System part	Constraint on entities	Constraints on relations
Data entities	Members only (no code)	Association and Composition Code modules

**TIP** Each part of a DOP system is easy to understand, because it has constraints.

## 2.6 DOP systems are flexible

**THEO:** I see how a sharp separation between code and data makes DOP systems easier to understand than classic OOP systems. But what about adapting to changes in requirements?

**JOE:** Another benefit of DOP systems is that it is easy to extend them and adapt to changing requirements.

**THEO:** I remember that when Nancy asked me to add *Super Members* and *VIP Members* to the system it was hard to adapt my OOP system: I had to introduce a few base classes and the class hierarchy became really complex.

**JOE:** I know exactly what you mean. I've experienced the same kind of struggle so many times. Describe the changes in the requirements for *Super Members* and *VIP Members* and I'm quite sure that you'll see how easy would be to extend your DOP system.

**SIDE BAR****The requirements for Super Members and VIP Members**

- 1. Super Members are members that are allowed to list the book lendings of other members**
- 2. VIP Members are members that are allowed to add book items to the library**

Theo opens his IDE and starts to code the `getBookLendings` function of the Library module, first without addressing the requirements for Super Members. Theo remembers what Joe told him about module functions in DOP:

1. Functions are stateless
2. Functions receive the data they manipulate as their first argument

In terms of functionality, `getBookLendings` has two parts:

1. Check that the user is a librarian
2. Retrieve the book lendings from the catalog

Basically, the code of `getBookLendings` has two parts:

1. Call `isLibrarian` function from the `UserManagement` module and pass it the `UserManagementData`
2. Call `getBookLendings` function from the `Catalog` module and pass it the `CatalogData`

Here is the code for `Library.getBookLendings`:

### **Listing 2.3 Getting the book lendings of a member**

```
class Library {
    static getBookLendings(libraryData, userId, memberId) {
        if(UserManagement.isLibrarian(libraryData.userManagement, userId)) {
            return Catalog.getBookLendings(libraryData.catalog, memberId);
        } else {
            throw "Not allowed to get book lendings"; ①
        }
    }
}

class UserManagement {
    static isLibrarian(userManagementData, userId) {
        // will be implemented later ②
    }
}

class Catalog {
    static getBookLendings(catalogData, memberId) {
        // will be implemented later ③
    }
}
```

- ① There are other ways to manage errors
- ② In Chapter 3, we will see how to manage permissions with generic data collections
- ③ In Chapter 3, we will see how to query data with generic data collections

It's Theo's first piece of DOP code: passing around all those data objects `libraryData`, `libraryData.userManagement` and `libraryData.catalog` feels a bit awkward. But he's done it.

Joe looks at Theo's code and seems satisfied.

**JOE:** How would you adapt your code to adapt to Super Members?

**THEO:** I would add a function `isSuperMember` to the `UserManagement` module and call it from `Library.getBookLendings`

**JOE:** Exactly! It's as simple as that.

Theo types this piece of code on his laptop:

#### Listing 2.4 Allowing Super Members to get the book lendings of a member

```
class Library {
    static getBookLendings(libraryData, userId, memberId) {
        if(Usermanagement.isLibrarian(libraryData.userManagement, userId) ||
           Usermanagement.isSuperMember(libraryData.userManagement, userId)) {
            return Catalog.getBookLendings(libraryData.catalog, memberId);
        } else {
            throw "Not allowed to get book lendings"; ①
        }
    }
}

class UserManagement {
    static isLibrarian(userManagementData, userId) {
        // will be implemented later ②
    }
    static isSuperMember(userManagementData, userId) {
        // will be implemented later ②
    }
}

class Catalog {
    static getBookLendings(catalogData, memberId) {
        // will be implemented later ③
    }
}
```

- ① There are other ways to manage errors
- ② In Chapter 3, we will see how to manage permissions with generic data collections
- ③ In Chapter 3, we will see how to query data with generic data collections

Now, the awkward feeling caused by passing around all those data objects is dominated by a

feeling of relief: Adapting to this change in requirements takes only a few lines of code and requires no changes in the system design.

Once again, Joe seems satisfied.

**TIP**

**DOP systems are flexible. Quite often they adapt to changing requirements without changing the system design.**

Theo starts coding `addBookItem`. He looks at the signature of `Library.addBookItem` in [2.5](#) and the meaning of the third argument `bookItemInfo` isn't clear to him. He asks Joe for a clarification.

**Listing 2.5 The signature of `Library.addBookItem`**

```
class Library {
    static addBookItem(libraryData, userId, bookItemInfo) {
    }
}
```

**THEO:** What is `bookItemInfo`?

**JOE:** Let's call it the book item information and imagine we have a way to represent this information in a data entity named `bookItemInfo`.

**THEO:** You mean an object?

**JOE:** For now, it's OK to think about `bookItemInfo` as an object. Later on, I will show you how to we represent data in DOP.

Beside this subtlety about how the book item info is represented by `bookItemInfo`, the code for `Library.addBookItem` in [2.6](#) is quite similar to the code Theo wrote for `Library.getBookLendings` in [2.4](#). Once again, Theo is amazed by the fact that adding support for VIP Members requires no design change.

## Listing 2.6 Allowing VIP Members to add a book item to the library

```

class Library {
    static addBookItem(libraryData, userId, bookItemInfo) {
        if(UserManagement.isLibrarian(libraryData.userManagement, userId) ||
           UserManagement.isVIPMember(libraryData.userManagement, userId)) {
            return Catalog.addBookItem(libraryData.catalog, bookItemInfo);
        } else {
            throw "Not allowed to add a book item"; ①
        }
    }
}

class UserManagement {
    static isLibrarian(userManagementData, userId) {
        // will be implemented later ②
    }
    static isVIPMember(userManagementData, userId) {
        // will be implemented later ②
    }
}

class Catalog {
    static addBookItem(catalogData, memberId) {
        // will be implemented later ③
    }
}

```

- ① There are other ways to manage errors
- ② In Chapter 3, we will see how to manage permissions with generic data collections
- ③ In Chapter 4, we will see how to manage state of the system with immutable data

**THEO:** It takes a big mindset shift to learn how to separate code from data!

**JOE:** What was the most challenging thing to accept?

**THEO:** The fact that data is not encapsulated in objects.

**JOE:** It was the same for me when I switched from OOP to DOP.

Now it's time to eat! Theo takes Joe for lunch at *Simple*, a nice small restaurant near the office.

## 2.7 Summary

- DOP principles are language agnostic.
- DOP Principle #1: Separate code from data
- The separation between code and data in DOP systems make them simpler (easier to understand) than traditional OOP systems
- Data entities are the parts of your system that hold information.
- DOP is against data encapsulation.
- The more flexible a system is, the easier it is to adapt to changing requirements
- The separation between code and data in DOP systems make them more flexible than traditional OOP systems.
- When code is separated from data, we have the freedom to design code and data in isolation
- We represent data as data entities
- We discover the data entities of our system and group them into high level groups, either as a nested list or as a mind map.
- A DOP system is easier to understand than a traditional OOP system because the system is split into two parts: data entities and code modules.
- A module is an aggregation of functions.
- DOP systems are flexible: Quite often they adapt to changing requirements without changing the system design.
- In traditional OOP, the state of the object is an implicit argument to the methods of the object.
- Stateless functions receive data they manipulate as an explicit argument.
- The high level modules of a DOP system correspond to the high level data entities.
- All the functions in a DOP module are stateless.
- The only kind of relationship between DOP modules is the usage relation.
- For a discussion of polymorphism in DOP, see Chapter 13.

# *Basic data manipulation*

3

## This chapter covers

- Representing records with string maps to improve flexibility
- Manipulating data with generic functions
- Accessing each piece of information via its information path
- Gaining JSON serialization for free

## 3.1 Meditation and programming

After learning why and how to separate code from data in the previous chapter, let's talk about data on its own.

In contrast to traditional OOP where system design tends to involve a rigid class hierarchy, DOP prescribes that we represent our data model as a flexible combination of maps and arrays (or lists) where we can access each piece of information via an information path.

This chapter is a deep dive into the second principle of Data-Oriented Programming.

**NOTE**

**DOP Principle #2: Represent data entities with generic data structures**

We increase system flexibility when we represent records as string maps and not as objects instantiated from classes. This liberates data from the rigidity of a class-based system. Data becomes a first class citizen powered by generic functions to add, remove or rename fields.

**NOTE**

**We refer to maps that have strings as keys, as "string maps".**

The dependency between the code that manipulates data, and the data, is a weak dependency. The code only needs to know the keys of specific fields in the record it wants to manipulate. The code doesn't even need to know about all the keys in the record, only the ones relevant to it.

In this chapter, we'll deal only with data query. We'll discuss managing changes in system state in the next chapter.

## 3.2 Design a data model

During lunch at *Simple*, Theo and Joe don't talk about programming. Instead, they start getting to know each other on a personal level. Theo discovers that Joe is married to Kay, who has just opened her creative therapy practice after many years of studying various fields related to well-being. Neriah, their 14-year-old son, is passionate about drones while Aurelia, their 12-year-old daughter, plays the transverse flute.

Joe tells Theo that he's been practicing meditation for 10 years. Meditation, he says, has taught him how to break away from being continually lost in a storm thought (especially negative thoughts, which can be the source of great suffering) to achieve a more direct relationship with reality. The more he learns to experience reality as it is, the calmer his mind. When he first started to practice meditation it was sometimes difficult - and even weird - but by persevering he has increased his feeling of well-being with each passing year.

When they're back at the office, Joe tells Theo that his next step in his DOP journey will be about data model and data representation...

**JOE:** When we design of the data part of our system, we're free to do it in isolation.

**THEO:** What do you mean by *isolation*?

**JOE:** I mean that you don't have to bother with code. Only data.

**THEO:** Oh, right, I remember you telling me how that makes a DOP system simpler than OOP. Separation of concerns is a design principle I'm used to in OOP.

**JOE:** Indeed.

**THEO:** And when we think about data, the only relations we have to think about are association and composition.

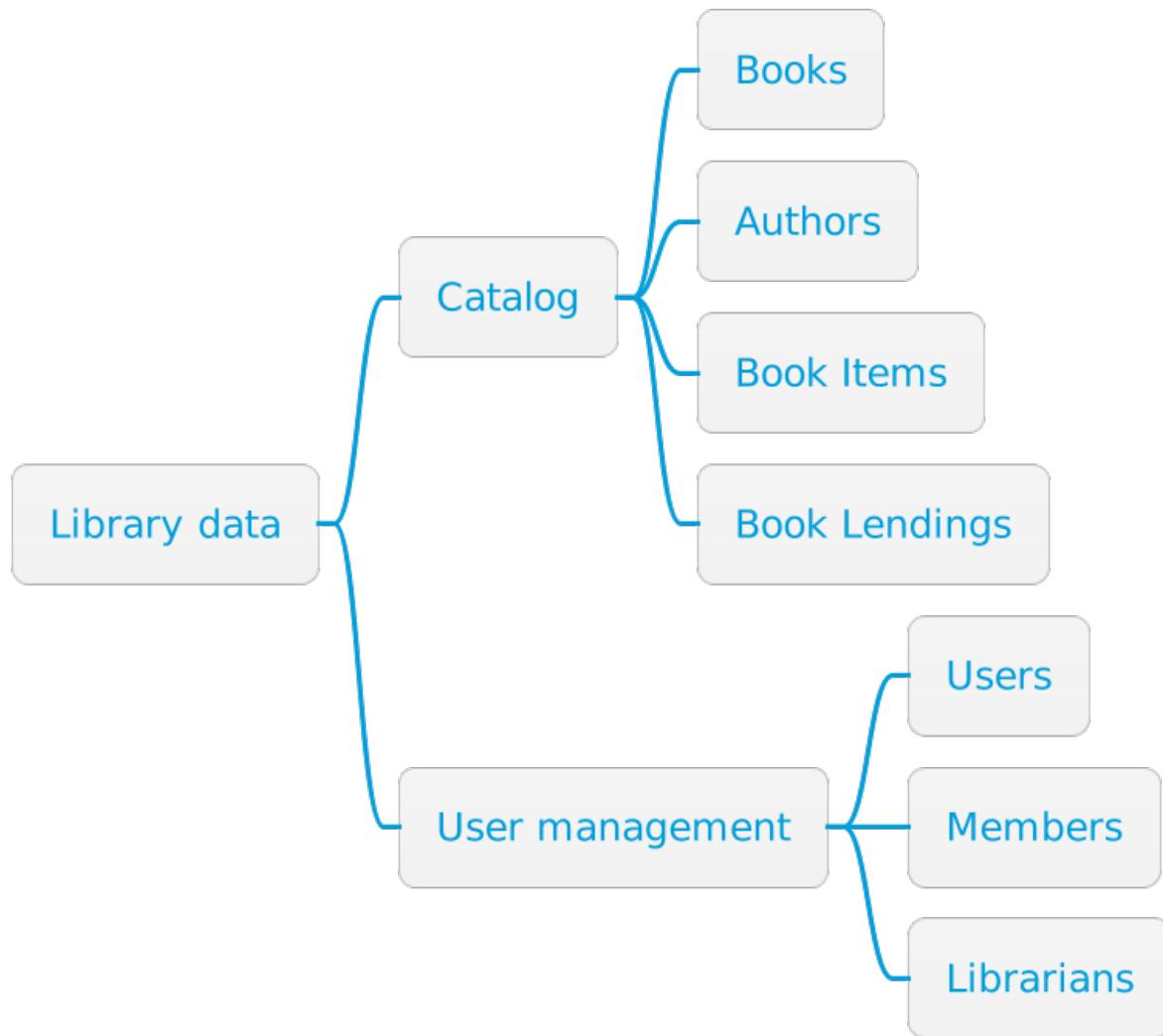
**JOE:** Correct.

**THEO:** Will the data model design be significantly different than the data model I'm used to designing as an OOP developer?

**JOE:** Not so much.

**THEO:** OK. Let me see if I can draw a DOP-style data entity diagram.

Theo takes a look at the data mind map that he drew earlier in the morning:



**Figure 3.1** A data mindmap of the Library management system

He refines the details of the fields of each data entity and the kind of relationships between entities, and the result is this data entity diagram:

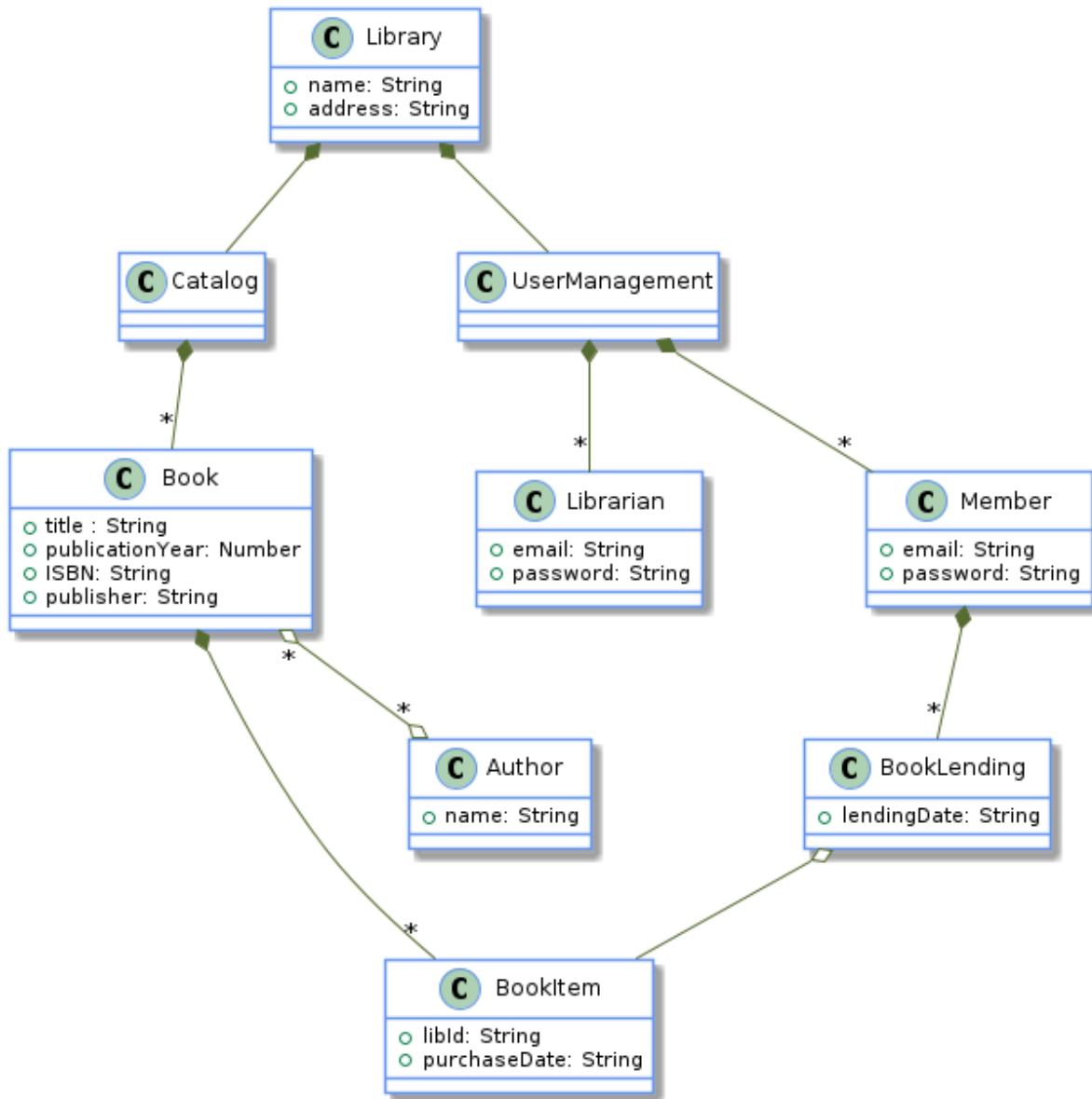


Figure 3.2 A data model of the Library management system

**JOE:** The next step is to be more explicit about the relations between entities.

**THEO:** What do you mean?

**JOE:** For example, in your entity diagram, **Book** and **Author** are connected by a many-to-many association relation. How is this relation going to be represented in your program?

**THEO:** In the **Book** entity, there will be a collection of author IDs, and in the **Author** entity, there will be a collection of book IDs.

**JOE:** Sounds good. And what will the book ID be?

**THEO:** The book ISBN.<sup>1</sup>

**JOE:** And where will you hold the index that will enable you to retrieve a Book from its ISBN?

**THEO:** In the Catalog. The catalog holds a bookByISBN index.

**JOE:** What about author ID?

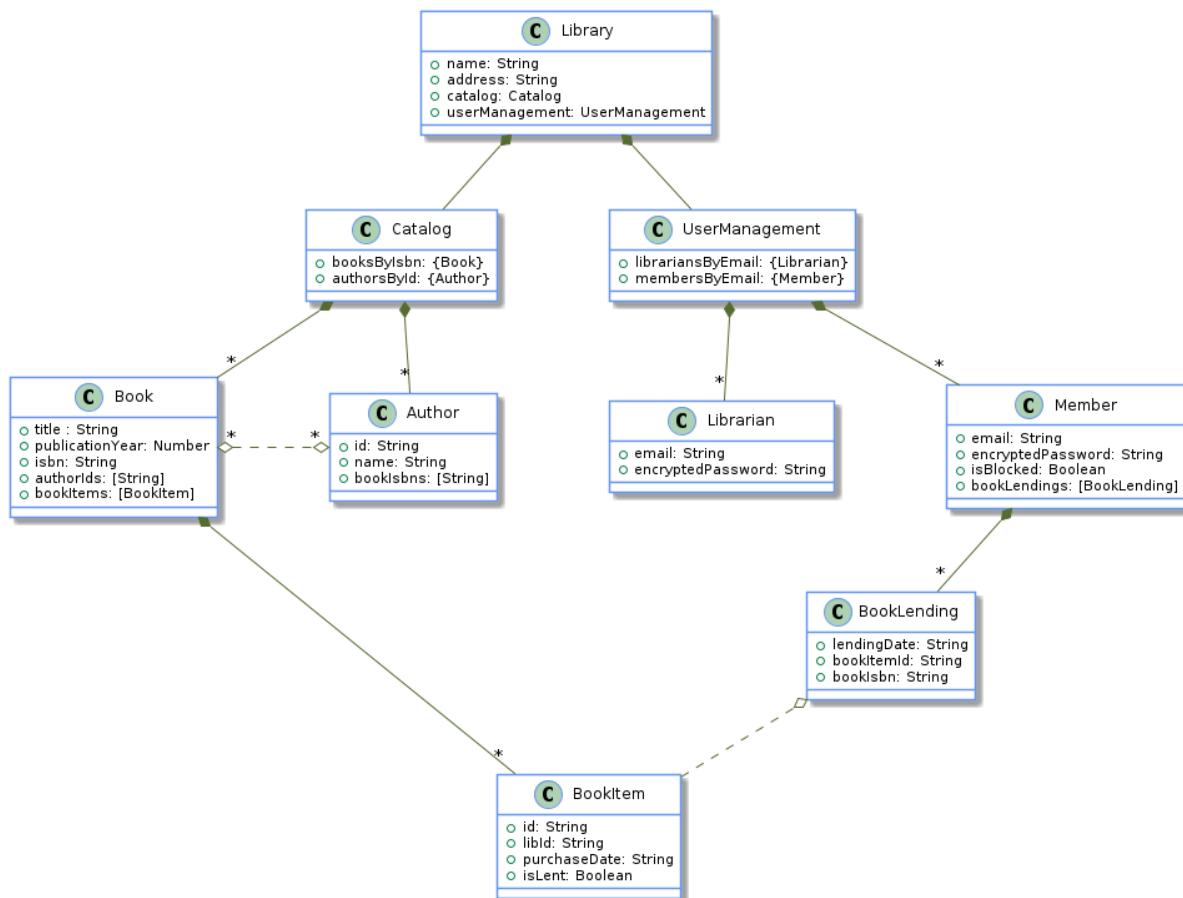
**THEO:** Author ID is the author name, in lower case, and with dashes instead of white spaces (assuming that we don't have two authors with the same name).

**JOE:** And I guess that you also hold the author index in the Catalog?

**THEO:** Exactly!

**JOE:** Excellent. You've been 100% explicit about the relation between Book and Author. I'll ask you to do the same with the other relations of the system.

It's quite easy for Theo as he has done that so many times as an OOP developer. Here's the detailed entity diagram of Theo's system:



**Figure 3.3 Library management relations model.** Dashed lines (e.g., between Book and Author) denotes indirect relations. `[String]` denotes a positional collection of strings. `{Book}` denotes an index of Books.

**NOTE** By positional collection, we mean a collection where the elements are in order (like a list or an array). By index, we mean a collection where the elements are accessible via a key (like a hash map, or a dictionary).

The `Catalog` entity contains two indexes:

1. `booksByIsbn`: The keys are book ISBNs and the values are `Book` entities. Its type is noted as `{Book}`
2. `authorsById`: The keys are author IDs and the values are `Author` entities. Its type is noted as `{Author}`

Inside a `Book` entity, we have `authors`, which is a positional collection of author IDs of type `[String]`. Inside an `Author` entity, we have `books`, which is a collection of book IDs of type `[String]`.

**NOTE** Notation for collection and index types: A positional collection of `String`s is noted as `[String]`. An index of `Books` is noted as `{Book}`. In the context of a data model, the index keys are always strings.

There is a dashed line between `Book` and `Author`, which means that the relation between `Book` and `Author` is indirect. To access the collection of `Author` entities from a `Book` entity, we'll use the `authorById` index defined in the `Catalog` entity.

**JOE:** I like your data entity diagram.

**THEO:** Thank you.

**JOE:** Can you tell me what the three kinds of data aggregations are in your diagram (and in fact in any data entity diagram)?

**THEO:** Let's see... We have positional collections, like `authors` in `Book`. We have indexes, like `booksByIsbn` in `Catalog`. I can't find the third one.

**JOE:** The third kind of data aggregation is what we've called until now an "entity" (like `Library`, `Catalog`, `Book`, etc...). The common term for "entity" in computer science is record.

**NOTE** A record is a data structure that groups together related data items. It's a collection of fields, possibly of different data types.

**THEO:** Is it correct to say that a data entity diagram consists only of records, positional collections and indexes?

**JOE:** That's correct. Can you make a similar statement about the relations between entities?

**THEO:** The relations in a data entity diagram are either composition (solid line with full diamond) or association (dashed line with empty diamond). Both types of relations can be either 1-to-1, 1-to-many or many-to-many.

**JOE:** Excellent!

**TIP**

A data entity diagram consists of records whose values are either primitives, positional collections or indexes. The relation between records is either composition or association.

### 3.3 Represent records as maps

So far, we've illustrated the benefits we gain from the separation between Code and Data at a high system level. There's a separation of concerns between code and data, and each part has clear constraints:

1. Code consists of static functions that receive data as an explicit argument
2. Data entities are modeled as records, and the relations between records are represented by positional collections and indexes

Now comes the question of the representation of the data.

DOP has nothing special to say about collections and indexes. However, it's strongly opinionated about the representation of records: records should be represented by generic data structures such as maps.

It applies both to Object-Oriented and to Functional programming languages. In dynamically-typed languages like JavaScript, Python and Ruby it feels very natural. While in statically-typed languages like Java and C#, it is a bit more cumbersome.

**THEO:** I'm really curious to know how we represent positional collections, indexes and records in DOP.

**JOE:** Let's start with positional collections. DOP has nothing special to say about the representation of collections. They can be linked lists, arrays, vectors, sets or other collections best suited for the use case.

**THEO:** It's like in OOP.

**JOE:** Right. For now, to keep things simple, we'll use arrays to represent positional collections.

**THEO:** What about indexes?

**JOE:** Indexes are represented as homogeneous string maps.

**THEO:** What do you mean by an *homogeneous* map?

**JOE:** I mean that all the values of the map are of the same kind. For example, in a Book index, all the values are Book, in an author index, all the values are Author, etc...

**THEO:** Again, it's like in OOP.

**NOTE**

A homogeneous map is a map where all the values are of the same type. A heterogeneous map is a map where the values are of different types.

**JOE:** Now, here's the big surprise. In DOP, records are represented as maps; more precisely, heterogeneous string maps.

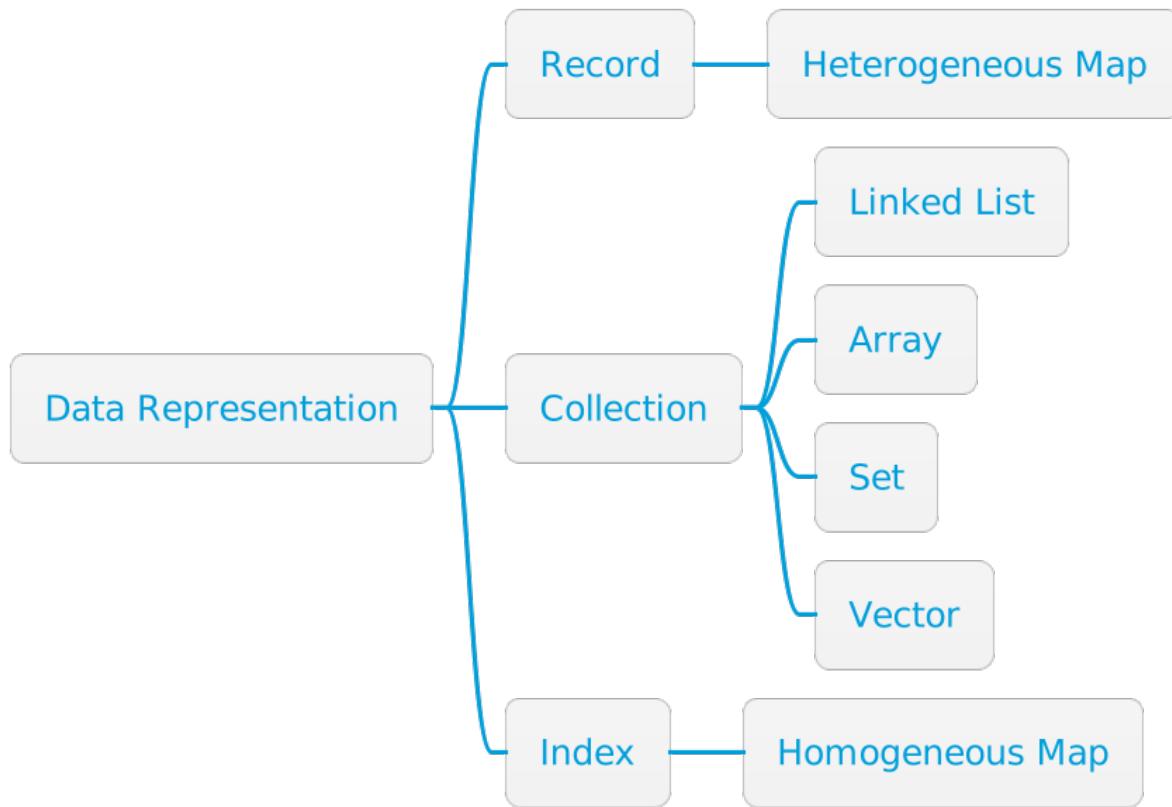


Figure 3.4 The building blocks of data representation

Theo stays silent for a while. He is shocked to hear that one can represent the data entities of a system as a generic data structure, where the field names and value types are not specified in a class.

Then Theo asks Joe:

**THEO:** What are the benefits of this folly?

**JOE:** Flexibility and genericity.

**THEO:** Could you explain, please?

**JOE:** I'll explain in a moment, but before that, I'd like to show you what an instance of a record in a DOP system looks like.

**THEO:** OK.

**JOE:** Let's take as an example, "Watchmen" by Alan Moore and Dave Gibbons, which is my favorite graphic novel. This masterpiece was published in 1987. I'm going to assume that there are two copies of this book in the library, both located in a physical library whose ID is nyc-central-lib, and that one of the two copies is currently out. Here's how I'd represent the Book record for "Watchmen" in DOP.

Joe comes closer to Theo's laptop, opens a text editor (not an IDE!) and starts typing...

### Listing 3.1 An instance of a Book record represented as a map

```
{
  "isbn": "978-1779501127",
  "title": "Watchmen",
  "publicationYear": 1987,
  "authors": [ "alan-moore", "dave-gibbons" ],
  "bookItems": [
    {
      "id": "book-item-1",
      "libId": "nyc-central-lib",
      "isLent": true
    },
    {
      "id": "book-item-2",
      "libId": "nyc-central-lib",
      "isLent": false
    }
  ]
}
```

Theo looks at the laptop screen and ask Joe:

**THEO:** How am I supposed to instantiate the Book record for "Watchmen" programmatically?

**JOE:** It depends on the facilities that your programming language offers to instantiate maps. With dynamic languages like JavaScript, Ruby or Python, it's straightforward because we can leverage literals for maps and arrays.

### Listing 3.2 Creating an instance of a Book record represented as a map in JavaScript

```
var watchmenBook = {
  "isbn": "978-1779501127",
  "title": "Watchmen",
  "publicationYear": 1987,
  "authors": ["alan-moore", "dave-gibbons"],
  "bookItems": [
    {
      "id": "book-item-1",
      "libId": "nyc-central-lib",
      "isLent": true
    },
    {
      "id": "book-item-2",
      "libId": "nyc-central-lib",
      "isLent": false
    }
  ]
}
```

**THEO:** And if I'm in Java?

**JOE:** It's a bit more tedious, but still doable with the immutable `Map` and `List` static factory methods.<sup>2</sup>:

### Listing 3.3 Creating an instance of a Book record represented as a map in Java

```
Map watchmen = Map.of(
  "isbn", "978-1779501127",
  "title", "Watchmen",
  "publicationYear", 1987,
  "authors", List.of("alan-moore", "dave-gibbons"),
  "bookItems", List.of(
    Map.of(
      "id", "book-item-1",
      "libId", "nyc-central-lib",
      "isLent", true
    ),
    Map.of(
      "id", "book-item-2",
      "libId", "nyc-central-lib",
      "isLent", false
    )
  )
);
```

**TIP**

In DOP, we represent a record as a heterogeneous string map.

**THEO:** I'd definitely prefer to create a Book record using a `Book` class and a `BookItem` class.

Theo opens his IDE and he starts typing...

### Listing 3.4 Representing a Book record as an instance of a Book class in JavaScript

```

class Book {
    isbn;
    title;
    publicationYear;
    authors;
    bookItems;
    constructor(isbn, title, publicationYear, authors, bookItems) {
        this.isbn = isbn;
        this.title = title;
        this.publicationYear = publicationYear;
        this.authors = authors;
        this.bookItems = bookItems;
    }
}

class BookItem {
    id;
    libId;
    isLent;
    constructor(id, libId, isLent) {
        this.id = id;
        this.libId = libId;
        this.isLent = isLent;
    }
}

var watchmenBook = new Book("978-1779501127",
                            "Watchmen",
                            1987,
                            ["alan-moore", "dave-gibbons"],
                            [new BookItem("book-item-1", "nyc-central-lib", true),
                             new BookItem("book-item-2", "nyc-central-lib", false)]);

```

**JOE:** Why do you prefer classes over maps for representing records?

**THEO:** It makes the data shape of the record part of my program. As a result, the IDE can auto-complete field names, and errors are caught at compile time.

**JOE:** Fair enough. Can I show you some drawbacks of this approach?

**THEO:** Sure.

**JOE:** Imagine that you want to display the information about a book in the context of search results. In that case, instead of author IDs, you want to display author names and you don't need the book item information. How would you handle that?

**THEO:** I'd create a class `BookInSearchResults` without a `bookItems` member, and with an `authorNames` member instead of the `authorIds` member of the `Book` class. Also, I would need to write a copy constructor that receives a `Book` object.

**JOE:** The fact that in classic OOP data is instantiated only via classes brings safety. But this safety comes at the cost of flexibility.

**THEO:** How can it be different?

**TIP**

**There's a trade-off between flexibility and safety in a data model.**

**JOE:** In the DOP approach, where records are represented as maps, we don't need to create a class for each variation of the data. We're free to add, remove and rename record fields dynamically. Our data model is flexible.

**THEO:** Interesting!

**TIP**

**In DOP, the data model is flexible. We're free to add, remove and rename record fields dynamically, at runtime.**

**JOE:** Now, let me talk about genericity: How would you serialize the content of a `Book` object to JSON?

**TIP**

**In DOP, records are manipulated with generic functions.**

**THEO:** Oh no! I remember that while working on Klfm prototype I had a nightmare about JSON serialization when I was developing the first version of the Library Management system.

**JOE:** Well, in DOP, serializing a record to JSON is super easy.

**THEO:** Does it involve *reflection* to go over the fields of the record, like the gson Java library (<https://github.com/google/gson>) does?

**JOE:** Not at all! Remember that in DOP, a record is nothing more than data. We can write a generic JSON serialization function that works with any record. It can be a `Book`, an `Author`, a `BookItem`, or anything else.

**THEO:** Amazing!

**TIP**

**In DOP, you get JSON serialization for free.**

**JOE:** Actually, as I'll show you in a moment, lots of data manipulation stuff can be done using generic functions.

**THEO:** Are the generic functions part of the language?

**JOE:** It depends on the functions and on the language. For example, JavaScript provides a JSON serialization function called `JSON.stringify` out of the box, but none for omitting multiple keys or for renaming keys.

**THEO:** That's annoying.

**JOE:** Not so much. There are third-party libraries that provide data-manipulation facilities. A popular data-manipulation library in the JavaScript ecosystem is Lodash (<https://lodash.com/>).

**THEO:** And in other languages?

**JOE:** Lodash has been ported to Java (<https://javalibs.com/artifact/com.github.javadev/underscore-lodash>), to C# (<https://www.nuget.org/packages/lodash/>), to Python (<https://github.com/dgilland/pydash>) and to Ruby (<https://rudash-website.now.sh/>).

**WARNING** Throughout the book, we use Lodash to show how to manipulate data with generic functions. But there is nothing special about Lodash. The exact same approach could be implemented via other data manipulation libraries or custom code.

**THEO:** Cool!

**JOE:** Actually, Lodash and its rich set of data manipulation functions can be ported to any language! That's why it's so beneficial to represent records as maps!

**TIP** DOP compromises on data safety to gain flexibility and genericity.

**Table 3.1 Trade-off between safety, flexibility and genericity**

	OOP	DOP
Safety	high	low
Flexibility	low	high
Genericity	low	high

## 3.4 Manipulate data with generic functions

**JOE:** Now let me show you how to manipulate data in DOP with generic functions.

**THEO:** Yes, I'm quite curious to see how you'll implement the search functionality of the Library Management system.

**JOE:** OK. First, let's instantiate, according to your data model from [3.3](#), a Catalog record for the catalog data of a library, where we have a single book, "Watchmen".

Joe instantiates a Catalog record in [3.5](#).

### Listing 3.5 A Catalog record

```
var catalogData = {
  "booksByIsbn": {
    "978-1779501127": {
      "isbn": "978-1779501127",
      "title": "Watchmen",
      "publicationYear": 1987,
      "authorIds": ["alan-moore", "dave-gibbons"],
      "bookItems": [
        {
          "id": "book-item-1",
          "libId": "nyc-central-lib",
          "isLent": true
        },
        {
          "id": "book-item-2",
          "libId": "nyc-central-lib",
          "isLent": false
        }
      ]
    },
    "authorsById": {
      "alan-moore": {
        "name": "Alan Moore",
        "bookIsbns": ["978-1779501127"]
      },
      "dave-gibbons": {
        "name": "Dave Gibbons",
        "bookIsbns": ["978-1779501127"]
      }
    }
}
```

**THEO:** I see the two indexes we talked about, `booksByIsbn` and `authorsById`. How do you differentiate a record from an index in DOP?

**JOE:** In an entity diagram there's a clear distinction between records and indexes. But in our code, both are plain data.

**THEO:** I guess that's why this approach is called Data-Oriented Programming.

**JOE:** See how straightforward it is to visualize any part of the system data inside a program? The reason is that data is represented as data!

**THEO:** That sounds like a lapalissade!<sup>3</sup>

**TIP**

In DOP, data is represented as data.

**JOE:** Oh, does it? I'm no so sure! In OOP, data is usually represented by *objects*, which makes it more challenging to visualize data inside a program.

**TIP**

In DOP, we can visualize any part of the system data.

**THEO:** How would you retrieve the title of a specific book from the catalog data?

**JOE:** Great question. In fact, in a DOP system, every piece of information has an *information path* from which we can retrieve the information.

**THEO:** Information path?

**JOE:** For example, the information path to the title of the "Watchmen" book in the catalog is:  
`[ "booksByIsbn", "978-1779501127", "title" ].`

**THEO:** Ah, I see. So is an "information path" sort of like a file path? But that names in an information path correspond to nested entities?

**JOE:** Exactly right. And once we have the path of a piece of information, we can retrieve the information with Lodash's `_.get` function.

Joe types a few characters on Theo's laptop as in [3.6](#)

### Listing 3.6 Retrieving the title of a book from its information path

```
_.get(catalogData, [ "booksByIsbn", "978-1779501127", "title" ])
// "Watchmen"
```

**THEO:** Neat. I wonder how hard it would be to implement a function like `_.get` myself.

After a few minutes of trial and error, Theo is able to produce the code in [3.7](#).

### Listing 3.7 Custom implementation of `get`

```
function get(m, path) {
  var res = m;
  for(var i = 0; i < path.length; i++) {    ①
    var key = path[i];
    res = res[key];
  }
  return res;
}
```

① We could use `forEach` instead of a `for` loop

After testing Theo's implementation of `get` as in [3.8](#), Joe compliments Theo.

### Listing 3.8 Testing the custom implementation of `get`

```
get(catalogData, [ "booksByIsbn", "978-1779501127", "title" ]);
// "Watchmen"
```

**JOE:** Well done!

**THEO:** I wonder if a function like `_.get` works smoothly in a statically-typed language like

Java?

**JOE:** It depends whether you need only to pass the value around or to concretely access the value.

**THEO:** I don't follow.

**JOE:** Imagine that once you get the title of a book, you want to convert the string into an upper-case string. So you need to do a static cast to `String`.

### Listing 3.9 Casting a field value to a string, in order to manipulate it as a string

```
((String)watchmen.get("title")).toUpperCase()
```

**THEO:** That makes sense. The values of the map are of different types so the compiler declares it as a `Map<String, Object>`. The information of the type of the field is lost.

**JOE:** It's a bit annoying, but quite often our code just passes the data around - in which case we don't have to deal with static casting. Moreover, in a language like C#, using the `dynamic` data type,<sup>4</sup> type casting can be avoided.<sup>5</sup>

**TIP**

In statically-typed languages, we sometimes need to statically cast the field values.

**THEO:** What about performance?

**JOE:** In most programming languages, maps are quite efficient. Accessing a field in a map is slightly slower than accessing a class member. Usually it's not significant.

**TIP**

No significant performance hit for accessing a field in a map instead of a class member.

**THEO:** Let's get back to this idea of information path. It works in OOP too. I could access the title of the "Watchmen" book with `catalogData.booksByIsbn["978-1779501127"].title`. I'd use Class members for record fields and strings for index keys.

**JOE:** There's a fundamental difference. When records are represented as maps, the information can be retrieved via its information path using a generic function like `_get`. But when records are represented as objects, you need to write specific code for each type of information path.

**THEO:** What do you mean by *specific* code? What's specific in `catalogData.booksByIsbn["978-1779501127"].title`?

**JOE:** In a statically-typed language like Java you'd need to import the class definitions for

Catalog and Book.

**THEO:** And in a dynamically-typed language like JavaScript?

**JOE:** Even in JavaScript, when you represent records with objects instantiated from classes, you can't easily write a function that receives a path as an argument and display the information that corresponds to this path. You would have to write specific code for each kind of path. You'd access class members with dot notation and map fields with bracket notation.

**THEO:** Would you say that in DOP, the information path is a first-class citizen?

**JOE:** Absolutely! The information path can be stored in a variable and passed as an argument to a function.

### TIP

In DOP, you can retrieve every piece of information via a path and a generic function.

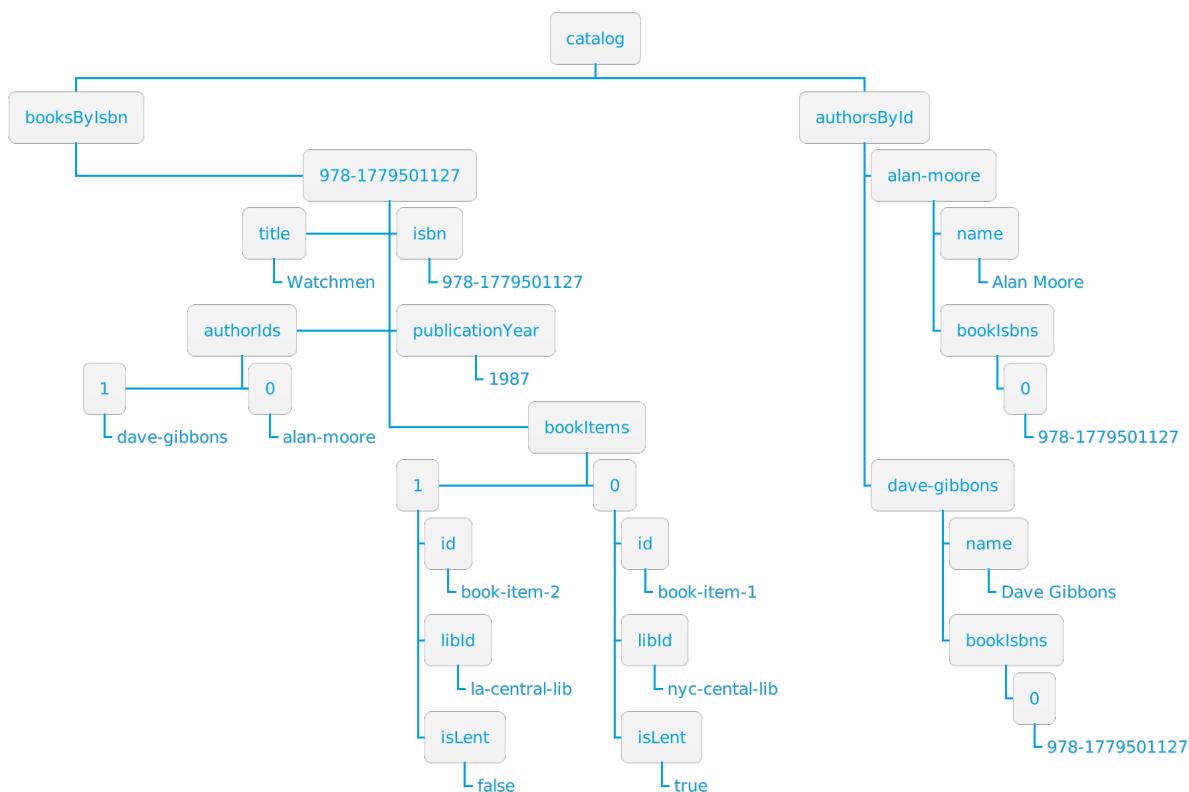


Figure 3.5 The catalog data as a tree. Each piece of information is accessible via a path made of strings and integers. For example, the path of Alan Moore's first book of is `["catalog", "authorsById", "alan-moore", "bookIsbns", 0]`.

## 3.5 Calculate search results

**THEO:** Interesting. I'm starting to feel the power of expression of DOP!

**JOE:** Wait. That's just the beginning. Let me show you how simple it is to write code that retrieves book information and displays it in search results. Can you tell me exactly what information has to appear in the search results?

**THEO:** Searching for book information should return `isbn`, `title` and `authorNames`.

**JOE:** And what would a `BookInfo` record look like for "Watchmen"?

**THEO:** Here you go...

#### Listing 3.10 A `BookInfo` record for *Watchmen* in the context of search result

```
{
  "title": "Watchmen",
  "isbn": "978-1779501127",
  "authorNames": [
    "Alan Moore",
    "Dave Gibbons",
  ]
}
```

**JOE:** Now, I'm going to show you, step by step, how to write a function that returns search results matching a title in JSON format, using generic data manipulation functions from Lodash.

**THEO:** I'm ready!

**JOE:** Let's start with an `authorNames` function that calculates the author names of a `Book` record by looking at the `authorsById` index. Could you tell me what's the information path for the name of an author whose id is `authorId`?

**THEO:** It's `["authorsById", "authorId", "name"]`.

**JOE:** Now, let me show you how to retrieve the name of several authors using `_.map`.

Joe types the code in [3.11](#).

#### Listing 3.11 Mapping author ids to author names

```
_.map(["alan-moore", "dave-gibbons"],
      function(authorId) {
        return _.get(catalogData, ["authorsById", authorId, "name"]);
      });
//[
//  "Alan Moore",
//  "Dave Gibbons",
//]
```

**THEO:** What's this `_.map` function? It smells like Functional Programming! You said I wouldn't have to learn FP to implement DOP!

**JOE:** No need to learn FP in order to use `_.map`: it's a function that transforms the values of a

collection. You can implement it with a simple for loop.

Theo spends a couple of minutes in front of his computer figuring out how to implement `_.map`. The result of his endeavour is in [3.12](#).

### **Listing 3.12 Custom implementation of `map`**

```
function map(coll, f) {
  var res = [];
  for(var i = 0; i < coll.length; i++) { ①
    res[i] = f(coll[i]);
  }
  return res;
}
```

- ① We could use `forEach` instead of a `for` loop

After testing Theo's implementation of `map` with the code snippet in [3.13](#), Joe compliments Theo.

### **Listing 3.13 Testing the custom implementation of `map`**

```
map(["alan-moore", "dave-gibbons"],
  function(authorId) {
    return _.get(catalogData, ["authorsById", authorId, "name"]);
});
//[
//  "Alan Moore",
//  "Dave Gibbons",
//]
```

**JOE:** Well done!

**THEO:** You were right. It wasn't hard.

**JOE:** Now, let's implement `authorNames` using `_.map`!

It takes a few minute to Joe to come up with the implementation of `authorNames` in [3.14](#).

### **Listing 3.14 Calculating the author names of a book**

```
function authorNames(catalogData, book) {
  var authorIds = _.get(book, "authorIds");
  var names = _.map(authorIds, function(authorId) {
    return _.get(catalogData, ["authorsById", authorId, "name"]);
  });
  return names;
}
```

**JOE:** We also need a `bookInfo` function that converts a `Book` record into a `BookInfo` record.

Joe implements `bookInfo` as in [3.15](#).

### Listing 3.15 Converting a Book record into a BookInfo record

```
function bookInfo(catalogData, book) {
  var bookInfo = {
    "title": _.get(book, "title"),
    "isbn": _.get(book, "isbn"),
    "authorNames": authorNames(catalogData, book)
  };
  return bookInfo; ①
}
```

- ① No need to create a class for bookInfo

**THEO:** Looking at the code, I see that a BookInfo record has three fields: title, isbn and authorNames. Is there a way to get this information without looking at the code?

**JOE:** You can either add it to the data entity diagram or write it in the documentation of the bookInfo function, or both.

**THEO:** I have to get used to the idea that in DOP, the record field information is not part of the program.

**JOE:** Indeed, it's not part of the program, but it gives us a lot of flexibility.

**THEO:** Is there any way for me to have my cake and eat it, too?!

**JOE:** Yes. Some day, I'll show you how to make record field information part of a DOP program (See chapters 7 and 12).

**THEO:** Sounds intriguing!

**JOE:** Now, we have all the pieces in place to write our searchBooksByTitle function that returns book information about the books that match the query. First, we find the Book records that match the query (with `_.filter`), and then we transform each Book record into a BookInfo record (with `_.map` and `bookInfo`).

Joe assembles all the pieces and implement `searchBooksByTitle` in [3.16](#).

### Listing 3.16 Searching books that match a query

```
function searchBooksByTitle(catalogData, query) {
  var allBooks = _.values(_.get(catalogData, "booksByIsbn"));
  var matchingBooks = _.filter(allBooks, function(book) {
    return _.get(book, "title").includes(query); ①
  });

  var bookInfos = _.map(matchingBooks, function(book) {
    return bookInfo(catalogData, book);
  });
  return bookInfos;
}
```

- ① `includes` is a JavaScript function that checks whether a string includes a string as a substring.

**THEO:** You're using Lodash functions without any explanation - again!

**JOE:** Sorry about that. I am so used to basic data manipulation functions that I consider them as part of the language. What functions are new to you?

**THEO:** `_.values` and `_.filter`.

**JOE:** `_.values` return a collection made of the values of a map and `_.filter` return a collection made of values that satisfy a predicate.

**THEO:** `_.values` seems trivial. Let me try to implement `_.filter`.

The implementation of `_.filter` takes a bit more time but eventually Theo manages to get it right in [3.17](#) and to test it in [3.18](#)

### Listing 3.17 Custom implementation of filter

```
function filter(coll, f) {
  var res = [];
  for(var i = 0; i < coll.length; i++) { ①
    if(f(coll[i])) {
      res.push(coll[i]);
    }
  }
  return res;
}
```

- ① We could use `forEach` instead of a `for` loop

### Listing 3.18 Testing the custom implementation of filter

```
filter(["Watchmen", "Batman"], function (title) {
  return title.includes("Watch");
});
//["Watchmen"]
```

**THEO:** It's a bit weird to me that to access a the title of a book record, you write `_.get(book, "title")`. I'd expect it to be `book.title` in dot notation, or `book["title"]` in bracket notation!

**JOE:** Remember that `book` is a record that's not represented as an object. It's a map. Indeed, in JavaScript, you can write `_.get(book, "title")`, `book.title` or `book["title"]`. But I prefer to use Lodash's `_.get`. In some languages, the dot and the bracket notations might not work on maps.

**THEO:** Being language-agnostic has a price!

**JOE:** Would you like to test `searchBooksByTitle`?

**THEO:** Absolutely! Let me call it to search the books whose title contain the string `watch`.

Theo executes on his laptop the code in [3.19](#).

### Listing 3.19 Testing `searchBooksByTitle`

```
searchBooksByTitle(catalogData, "Wat");
//[[
//  {
//    "authorNames": [
//      "Alan Moore",
//      "Dave Gibbons"
//    ],
//    "isbn": "978-1779501127",
//    "title": "Watchmen"
//  }
//]]
```

**THEO:** It seems to work! Are we done with the search implementation?

**JOE:** Almost. The `searchBooksByTitle` function we wrote is going to be part of the `Catalog` module, and it returns a collection of records. We have to write a function that's part of the `Library` module and that returns a JSON string.

**THEO:** You told me earlier that JSON serialization was straightforward in DOP.

**JOE:** Correct. The code for `searchBooksByTitleJSON` retrieves the `Catalog` record, passes it to `searchBooksByTitle`, and converts the results to JSON with `JSON.stringify` (that's part of JavaScript).

Joe quickly implements `searchBooksByTitleJSON` in [3.20](#).

### Listing 3.20 Implementation of searching books in a library as JSON

```
function searchBooksByTitleJSON(libraryData, query) {
  var results = searchBooksByTitle(_.get(libraryData, "catalog"), query);
  var resultsJSON = JSON.stringify(results);
  return resultsJSON;
}
```

**JOE:** In order to test our code, we need to create a `Library` record that contains our `Catalog` record. Could you do that for me please?

**THEO:** Should the `Library` record contain all the `Library` fields (`name`, `address`, `UserManagement`)?

**JOE:** That's not necessary. For now, we only need the `catalog` field.

Theo creates a `Library` record in [3.21](#) and tests `searchBooksByTitleJSON` in [3.22](#).

### Listing 3.21 A Library record

```

var libraryData = {
  "catalog": {
    "booksByIsbn": {
      "978-1779501127": {
        "isbn": "978-1779501127",
        "title": "Watchmen",
        "publicationYear": 1987,
        "authorIds": ["alan-moore",
                      "dave-gibbons"],
        "bookItems": [
          {
            "id": "book-item-1",
            "libId": "nyc-central-lib",
            "isLent": true
          },
          {
            "id": "book-item-2",
            "libId": "nyc-central-lib",
            "isLent": false
          }
        ]
      }
    },
    "authorsById": {
      "alan-moore": {
        "name": "Alan Moore",
        "bookIsbns": ["978-1779501127"]
      },
      "dave-gibbons": {
        "name": "Dave Gibbons",
        "bookIsbns": ["978-1779501127"]
      }
    }
  };
};

```

### Listing 3.22 Test for searching books in a library as JSON

```
searchBooksByTitleJSON(libraryData, "Wat");
```

**THEO:** How are we going to combine the four functions that we've written so far?

**JOE:** The functions `authorNames`, `bookInfo` and `searchBooksByTitle` go into the `Catalog` module, and `searchBooksByTitleJSON` goes into the `Library` module.

Theo looks at the resulting code of the two modules in [3.23](#), quite amazed by the conciseness of the code.

### Listing 3.23 Calculating search results. The code is split in two modules: Library and Catalog.

```

class Catalog {
    static authorNames(catalogData, book) {
        var authorIds = _.get(book, "authorIds");
        var names = _.map(authorIds, function(authorId) {
            return _.get(catalogData, ["authorsById", authorId, "name"]);
        });
        return names;
    }

    static bookInfo(catalogData, book) {
        var bookInfo = {
            "title": _.get(book, "title"),
            "isbn": _.get(book, "isbn"),
            "authorNames": Catalog.authorNames(catalogData, book)
        }; ①
        return bookInfo;
    }

    static searchBooksByTitle(catalogData, query) {
        var allBooks = _.get(catalogData, "booksByIsbn");
        var matchingBooks = _.filter(allBooks, function(book) { ②
            return _.get(book, "title").includes(query);
        });
        var bookInfos = _.map(matchingBooks, function(book) {
            return Catalog.bookInfo(catalogData, book);
        });
        return bookInfos;
    }
}

class Library {
    static searchBooksByTitleJSON(libraryData, query) {
        var catalogData = _.get(libraryData, "catalog");
        var results = Catalog.searchBooksByTitle(catalogData, query);
        var resultsJSON = JSON.stringify(results); ③
        return resultsJSON;
    }
}

```

- ① No need to create a class for bookInfo
- ② When `_.filter` is passed a map, it goes over the values of the map
- ③ Converts data to JSON (part of JavaScript)

After testing the final code in [3.24](#), Theo looks again at the source code from [3.23](#)... After a few seconds he feels like he's having an Aha! moment.

### Listing 3.24 Search results in JSON

```

Library.searchBooksByTitleJSON(libraryData, "Watchmen");
// "[{"title": "Watchmen", "isbn": "978-1779501127",
// "authorNames": ["Alan Moore", "Dave Gibbons"]}]"

```

**THEO:** The important thing is not that the code is concise, but that the code contains no abstractions. It's just data manipulation!

Joe responds with a smile that says, "You got it, my friend!"

**JOE:** It reminds me of what my first meditation teacher told me 10 years ago: Meditation guides the mind to grasp the reality as it is, without the abstractions created by our thoughts.

**TIP**

In DOP, many parts of our code base tend to be just about data manipulation with no abstractions.

### 3.6 Handle records of different types

We've seen how DOP enables us to treat records as first class citizens that can be manipulated in a flexible way using generic functions. But if a record is nothing more than an aggregation of fields, how do we know what the type of the record is?

DOP has a surprising answer to this question.

**THEO:** I have a question. If a record is nothing more than a *map*, how do you know the type of the record?

**JOE:** That's a great question with a surprising answer.

**THEO:** I'm curious.

**JOE:** Most of the time, there's no need to know the type of the record.

**THEO:** What do you mean?

**JOE:** I mean that what matters most are the values of the fields. For example, take a look at `Catalog.authorNames` source code. It operates on a `Book` record, but the only thing that matters is the value of the `authorIds` field.

Doubtful, Theo looks at the source code of `Catalog.authorNames` in [3.25](#).

#### Listing 3.25 Calculating the author names of a book

```
function authorNames(catalogData, book) {
  var authorIds = _.get(book, "authorIds");
  var names = _.map(authorIds, function(authorId) {
    return _.get(catalogData, ["authorsById", authorId, "name"]);
  });
  return names;
}
```

**THEO:** What about differentiating between various user types like Member vs Librarian? I mean, they both have `email` and `encryptedPassword`. How do you know if a record represents a Member or a Librarian?

**JOE:** You check to see if the record is found in the `librariansByEmail` index or in the `membersByEmail` index of the Catalog.

**THEO:** Could you be more specific?

**JOE:** Sure. Let me write down what the user management data of our tiny library might look like, assuming we have one librarian and one member. To keep things simple, I'm 'encrypting' passwords through naive base-64 encoding.

Joe creates a small `UserManagement` record in [3.26](#).

### Listing 3.26 A `UserManagement` record

```
var userManagementData = {
  "librariansByEmail": {
    "franck@gmail.com": {
      "email": "franck@gmail.com",
      "encryptedPassword": "bXlwYXNzd29yZA==" ①
    }
  },
  "membersByEmail": {
    "samantha@gmail.com": {
      "email": "samantha@gmail.com",
      "encryptedPassword": "c2VjcmV0", ②
      "isBlocked": false,
      "bookLendings": [
        {
          "bookItemId": "book-item-1",
          "bookIsbn": "978-1779501127",
          "lendingDate": "2020-04-23"
        }
      ]
    }
  }
}
```

① base-64 encoding of "mypassword"

② base-64 encoding of "secret"

#### TIP

Most of the time, there's no need to know what the type of a record is.

**THEO:** This morning, you told me you'd show me the code for `UserManagement.isLibrarian` function in the afternoon.

**JOE:** So here we are in the afternoon, and I'm going to fulfill my promise.

Joe implements `isLibrarian` as in [3.27](#) and tests it in [3.28](#).

### Listing 3.27 Checking if a user is a librarian

```
function isLibrarian(userManagement, email) {
  return _.has(_.get(userManagement, "librariansByEmail"), email);
}
```

**Listing 3.28 Testing isLibrarian**

```
isLibrarian(userManagementData, "franck@gmail.com");
//true
```

**THEO:** I'm assuming that `_.has` is a function that checks whether a key exists in a map. Right?

**JOE:** Correct.

**THEO:** OK. You simply check whether the `librariansByEmail` map contains the `email` field.

**JOE:** Yep.

**THEO:** Would you use the same pattern to check whether a member is a Super member or a VIP member?

**JOE:** Sure, we could have `SuperMembersByEmail` and `VIPMembersByEmail` indexes. But there's a better way.

**THEO:** How?

**JOE:** When a member is a VIP member, we add a field, `isVIP` with the value `true`, to its record. To check if a member is a VIP member, check whether the `isVIP` field is set to `true` in the member record.

Joe codes `isVIPMember` as in [3.29](#).

**Listing 3.29 Checking whether a member is a VIP member**

```
function isVIPMember(userManagement, email) {
    return _.get(userManagement, ["membersByEmail", email, "isVIP"]) == true;
}
```

**THEO:** I see that you access the `isVIP` field via its information path: `["membersByEmail", email, "isVIP"]`.

**JOE:** Yes. I think it makes the code crystal clear.

**THEO:** Agree. And I guess that we can do the same and have an `isSuper` field set to `true` when a member is a Super member?

**JOE:** Yes. Just like this.

Joe assembles all the pieces in a `UserManagement` class as in [3.30](#)

### Listing 3.30 The code of UserManagement module

```

class UserManagement {
    isLibrarian(userManagement, email) {
        return _.has(_.get(userManagement, "librariansByEmail"), email);
    }

    isVIPMember(userManagement, email) {
        return _.get(userManagement, ["membersByEmail", email, "isVIP"]) == true;
    }

    isSuperMember(userManagement, email) {
        return _.get(userManagement, ["membersByEmail", email, "isSuper"]) == true;
    }
}

```

Theo looks at the `UserManagement` module code for a couple of seconds, and suddenly an idea comes to his mind...

**THEO:** Why not have a `type` field in member record, whose value would be either `VIP` or `Super`?

**JOE:** I assume that, according to the product requirements, a member can be both a `VIP` and a `Super` member.

**THEO:** Hmm... Then `types` field could be a collection containing `VIP` or `Super` or both.

**JOE:** In some situations, having a `types` field is helpful, but I find it simpler to have a *boolean* field for each feature that the record supports.

**THEO:** Is there a name for fields like `isVIP` and `isSuper`?

**JOE:** I call them feature fields.

**TIP**

Instead of maintaining type information about a record, use a feature field (e.g., `isVIP`).

**THEO:** Can we use feature fields to differentiate between librarians and members?

**JOE:** You mean having an `isLibrarian` and an `isMember` field?

**THEO:** Yes, and having a common `User` record type for both librarians and members.

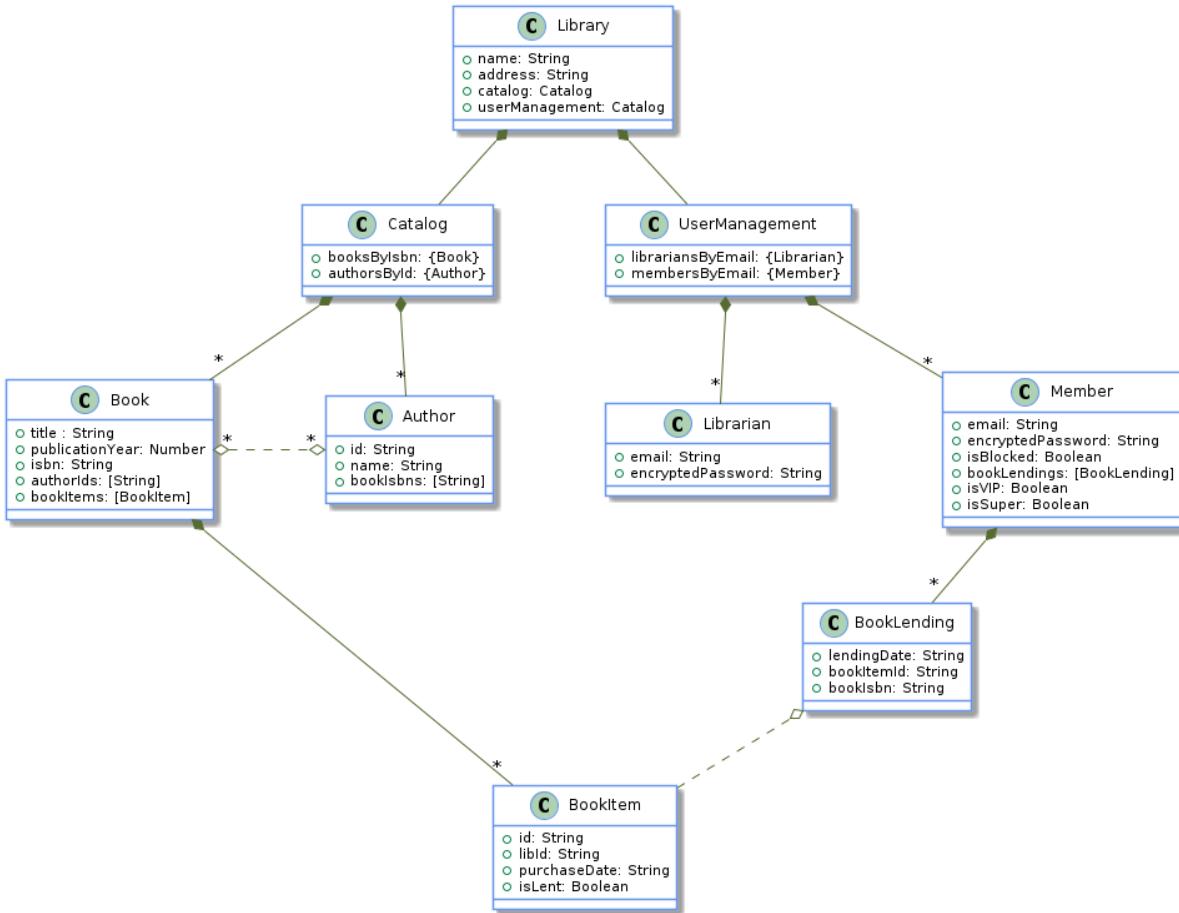
**JOE:** We can, but I think it's simpler to have different record types for librarians and members: `Librarian` for librarians, and `Member` for members.

**THEO:** Why?

**JOE:** Because there's a clear distinction between librarians and members in terms of data. For

example, members have book lendings but librarians don't.

**THEO:** I agree. Now, we need to mention the two `Member` feature fields in our entity diagram:



**Figure 3.6 Library management data model, with Member feature fields `isVIP` and `isSuper`.**

**JOE:** Do you like the data model that we have designed together?

**THEO:** I find it quite simple and clear.

**JOE:** That's the main goal of DOP.

**THEO:** Also, I'm pleasantly surprised how easy it is to adapt to changing requirements both in terms of code and data model.

**JOE:** I suppose you're also happy to get rid of complex class hierarchy diagrams.

**THEO:** Absolutely! Also I think I've found an interesting connection between DOP and meditation.

**JOE:** Really?

**THEO:** When we were eating at Simple, you told me that meditation helped you experience

reality as it is, without the filter of your thoughts.

**JOE:** Right.

**THEO:** From what you taught me today, I understand that in DOP we are encouraged to treat data as data, without the filter of our classes.

**JOE:** Clever! I never noticed that connection between those two disciplines that are so important for me. I guess you'd like to continue your journey in the realm of DOP.

**THEO:** Definitely. Let's meet again tomorrow.

**JOE:** Unfortunately, tomorrow we are going to the beach to celebrate the 12th birthday of my elder daughter, Aurelia.

**THEO:** Happy birthday Aurelia!

**JOE:** We could meet again next Monday, if that's OK for you.

**THEO:** With pleasure!

## 3.7 Summary

- DOP Principle #2: Represent data entities with generic data structures.
- We refer to maps that have strings as keys, as "string maps".
- Representing data as data means representing records with string maps.
- By *positional collection*, we mean a collection where the elements are in order (like a list or an array).
- A positional collection of `Strings` is noted as `[String]`.
- By *index*, we mean a collection where the elements are accessible via a key (like a hash map, or a dictionary).
- An index of `Books` is noted as `{Book}`.
- In the context of a data model, the index keys are always strings.
- A *record* is a data structure that groups together related data items. It's a collection of fields, possibly of different data types.
- A *homogeneous* map is a map where all the values are of the same type.
- A *heterogeneous* map is a map where the values are of different types.
- In DOP, we represent a record as a heterogeneous string map.
- A data entity diagram consists of records whose values are either primitives, positional collections or indexes.
- The relation between records in a data entity diagram is either composition or association.
- The data part of a DOP system is flexible, and each piece of information is accessible via its information path.
- There is a trade-off between flexibility and safety in a data model.
- DOP compromises on data safety to gain flexibility and genericity.
- In DOP, the data model is flexible. We're free to add, remove and rename record fields dynamically, at runtime.
- We manipulate data with generic functions
- Generic functions are provided either by the language itself or by third-party libraries like Lodash.
- JSON serialization is implemented in terms of generic function.
- On the one hand, we've lost the safety of accessing record fields via members defined at compile time. On the other hand, we've liberated data from the limitation of classes and objects. Data is represented as data!
- The weak dependency between code and data makes it is easier to adapt to changing requirements.
- When data is represented as data, it is straightforward to visualize system data.
- Usually, we do not need to maintain type information about a record.
- We can visualize any part of the system data.
- In statically-typed languages, we sometimes need to statically cast the field values.
- Instead of maintaining type information about a record, use a feature field.
- There is no significant performance hit for accessing a field in a map instead of a class member.
- In DOP, you can retrieve every piece of information via a path and a generic function.
- In DOP, many parts of our code base tend to be just about data manipulation with no

abstractions.

**Table 3.2 Trade-off between safety, flexibility and genericity**

	OOP	DOP
Safety	high	low
Flexibility	low	high
Genericity	low	high

**Table 3.3 Lodash functions introduced in this chapter**

Function	Description
<code>get(map, path)</code>	Gets the value at <code>path</code> of <code>map</code>
<code>has(map, path)</code>	Checks if <code>map</code> has a field at <code>path</code>
<code>merge(mapA, mapB)</code>	Creates a map resulting from the recursive merges between <code>mapA</code> and <code>mapB</code>
<code>values(map)</code>	Creates an array of values of <code>map</code>
<code>filter(coll, pred)</code>	Iterates over elements of <code>coll</code> , returning an array of all elements for which <code>pred</code> returns <code>true</code>
<code>map(coll, f)</code>	Creates an array of values by running each element in <code>coll</code> through <code>f</code>

# *State management*



## This chapter covers

- Multi-version approach to state management
- The calculation phase of a mutation
- The commit phase of a mutation
- Keeping a history of previous state versions

## 4.1 Time travel

So far we have seen how DOP handles queries via generic functions that access system data, which is represented as a hash map.

In this chapter, we illustrate how DOP deals with mutations, i.e. requests that change the system state. Instead of updating the state in place, we maintain multiple versions of the system data. At a specific point in time, the system state refers to a specific version of the system data.

This chapter is a deep dive in the third principle of Data-Oriented Programming:

**NOTE**

**Principle #3: Data is immutable.**

The maintenance of multiple versions of the system data requires the data to be immutable. This is made efficient both in terms of computation and memory via a technique called "structural sharing", where parts of the data that are common between two versions are shared instead of being copied.

In DOP, a mutation is split into two distinct phases:

1. In the calculation phase, we compute the next version of the system data.
2. In the commit phase, we move the system state forward so that it refers to the version of the system data computed by the calculation phase.

This distinction between calculation and commit phases allows us to reduce the part of our system that is stateful to its bare minimum. Only the code of the commit phase is stateful, while the code in the calculation phase of a mutation is stateless and made of generic functions similar to the code of a query.

The implementation of the commit phase is common to all mutations. As a consequence, inside the commit phase, we have the ability to ensure that the state always refers to a valid version of the system data.

Another benefit of this state management approach is that we can keep track of the history of previous versions of the system data. Restoring the system to a previous state (if needed) becomes straightforward.

**Table 4.1 The two phases of a mutation**

Phase	Responsibility	State	Implementation
calculation	Compute next version of system data	Stateless	Specific
commit	Move forward the system state	Stateful	Common

In this chapter, we assume that no mutations occur concurrently in our system. In the next chapter, we will deal with concurrency control.

## 4.2 Multiple versions of the system data

When Joe comes in to the office, he tells Theo that he needs to exercise before starting to work with his mind. Theo and Joe go for a walk around the block and the discussion turns around version control systems. They discuss how git keeps track of the whole commit history and how easy and fast it is to restore the code to a previous state. When Theo tells Joe that git's ability to "time travel" reminds him one of his favorite movies: "Back to the Future", Joe shares that a month ago he watched "Back to the future" trilogy with Neriah, his 14-year old son.

**JOE:** So far we have seen how in DOP, how we manage queries that retrieve information from the system. Now I'm going to show you how we manage mutations. By a mutation, I mean an operation that changes the state of the system.

**NOTE**

A mutation is an operation that changes the state of the system.

**THEO:** Is there a fundamental difference between queries and mutations in DOP? After all, the

whole state of the system is represented as a hash map. I could easily write code that modifies part of the hash map. It would be similar to the code that retrieves information from the hash map.

**JOE:** You could mutate the data in place, but then it would be challenging to make sure that the code of a mutation doesn't put the system into an invalid state. And you would lose the ability to track previous versions of the system state.

**THEO:** I see. So how do you handle mutations in DOP?

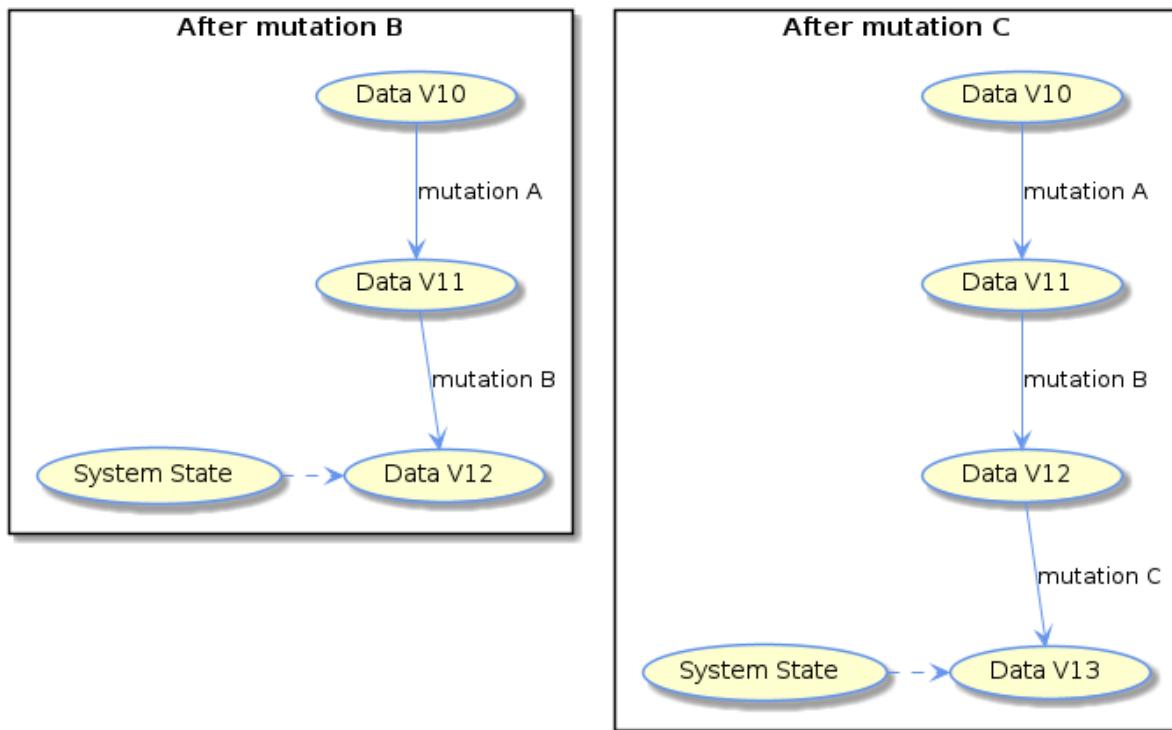
**JOE:** We adopt a multi-version state approach, similar to what a version control system like git does. We manage different versions of the system data. At a specific point in time, the state of the system refers to a version of the system data. After a mutation is executed, we move the reference forward.

**THEO:** I'm confused: is the system state mutable or immutable?

**JOE:** The data is immutable but the state reference is mutable.

**TIP**

The data is immutable but the state reference is mutable.



**Figure 4.1** After `mutation B` is executed, the system state refers to `Data v12`. After `mutation C` is executed, the system state refers to `Data v13`.

**THEO:** Does that mean that before the code of a mutation runs, we make a copy of the system

data?

**JOE:** No. That would be very inefficient, as we would have to do a deep copy of the data.

**THEO:** So how does it work?

**JOE:** It works by using a technique called *structural sharing*, where most of the data between subsequent versions of the state is shared instead of being copied. This technique efficiently creates new versions of the system data, both in terms of memory and computation.

**THEO:** I'm intrigued.

**TIP**

With structural sharing, it's efficient (in terms of memory and computation) to create new versions of data.

**JOE:** I'll explain in detail how structural sharing works in a moment.

Theo takes another look at the diagram in [4.1](#) that illustrates how the system state refers to a version of the system data and suddenly a question emerges in his mind.

**THEO:** Are the previous versions of the system data kept?

**JOE:** In a simple application, previous versions are automatically removed by the garbage collector. But in some cases, we maintain historical references to previous versions of the data.

**THEO:** What kind of cases?

**JOE:** For example, if we want to support time travel in our system. Like in git, we can move the system back to a previous version of the state very easily.

**THEO:** Now, I understand what you meant by: *The data is immutable but the state reference is mutable.*

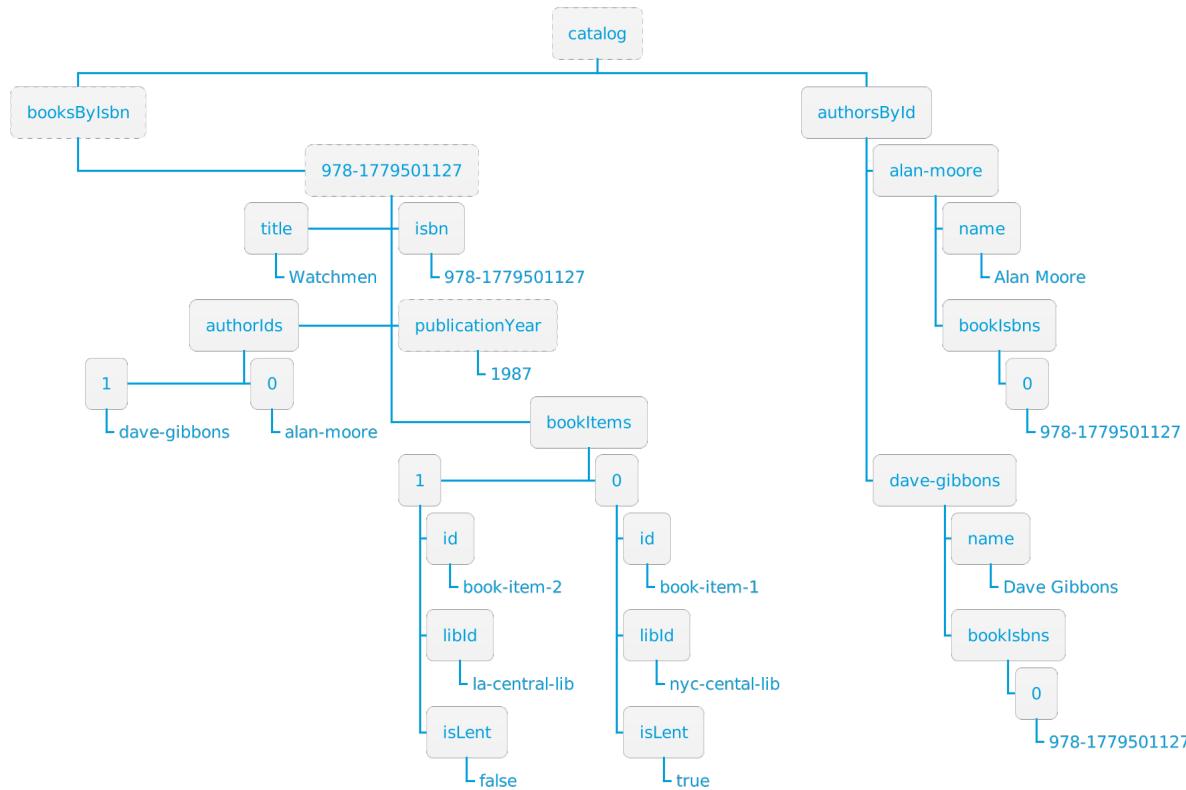
## 4.3 Structural sharing

As mentioned in the previous section, structural sharing enables the efficient creation of new versions of immutable data. In DOP, we leverage structural sharing in the calculation phase of a mutation to compute the next state of the system - based on the current state of the system. Inside the calculation phase, we don't have to deal with state management: that is delayed to the commit phase. As a consequence, the code involved in the calculation phase of a mutation is stateless and is as simple as the code of a query.

**THEO:** I'm really intrigued by this efficient way to create new version of data. How does it work?

**JOE:** Let's take a simple example from our library system. Imagine that you want to modify the value of a field in a book in the catalog, for instance the publication year of Watchmen. Can you tell me the information path for Watchmen's publication year?

After a quick look at the catalog data in [4.2](#), Theo answers:



**Figure 4.2 Visualization of the catalog data. The nodes in the information path to Watchmen's publication year are marked with a dotted border.**

**THEO:** The information path for Watchmen's publication year is: `["catalog", "booksByisbn", "978-1779501127", "publicationYear"]`.

**JOE:** Now, let me show how you to use the immutable function `_.set` provided by Lodash.

**THEO:** What do you mean by an immutable function? When I look at Lodash documentation for `_.set` in their website (<https://lodash.com/>), it says that it mutates the object.

**JOE:** You are right. By default Lodash functions are not immutable. In order to use a immutable version of the functions, we need to use Lodash FP module (Functional Programming), as it is explained in the Lodash FP guide (<https://github.com/lodash/lodash/wiki/FP-Guide>).

**THEO:** Do the immutable functions have the same signature as the mutable functions?

**JOE:** By default, the order of the arguments in immutable functions is shuffled. In the Lodash FP guide, they explain how to resolve it: with this piece of code in [4.1](#) the signature of the

immutable functions is exactly the same as the mutable functions.

### **Listing 4.1 Configuring Lodash so that the immutable functions have the same signature as the mutable functions**

```
_ = fp.convert({
  "cap": false,
  "curry": false,
  "fixed": false,
  "immutable": true,
  "rearg": false
});
```

**TIP** In order to use Lodash immutable functions, we use Lodash's FP module and we configure it so that the signature of the immutable functions is the same as in the Lodash documentation web site.

**THEO:** So basically, I can still rely on Lodash documentation when using immutable versions of the functions.

**JOE:** Except for the piece in the documentation that says the function mutates the object.

**THEO:** Of course!

**JOE:** Now, let me show you how to write code that creates a version of the library data with the immutable function `_.set` provided by Lodash.

### **Listing 4.2 Creating a version of the library where Watchmen publication year is 1986**

```
var nextLibraryData = _.set(libraryData,
  ["catalog", "booksByIsbn",
   "978-1779501127", "publicationYear"],
  1986);
```

**NOTE** A function is said to be immutable when instead of mutating the data, it creates a new version of the data without changing the data it receives.

**THEO:** You told me earlier that structural sharing allowed immutable functions to be efficient in terms of memory and computation. Could you tell me what makes them efficient?

**JOE:** With pleasure. But before that you have to answer a series of questions. Are you ready?

**THEO:** Yes.

**JOE:** What part of the library data is impacted by updating Watchmen publication year: the `UserManagement` or the `Catalog`?

**THEO:** Only the `Catalog`.

## **JOE: What part of the Catalog?**

**THEO:** Only the booksByIsbn index.

**JOE:** What part of the booksByIsbn index?

**THEO:** Only the Book record that holds the information about Watchmen.

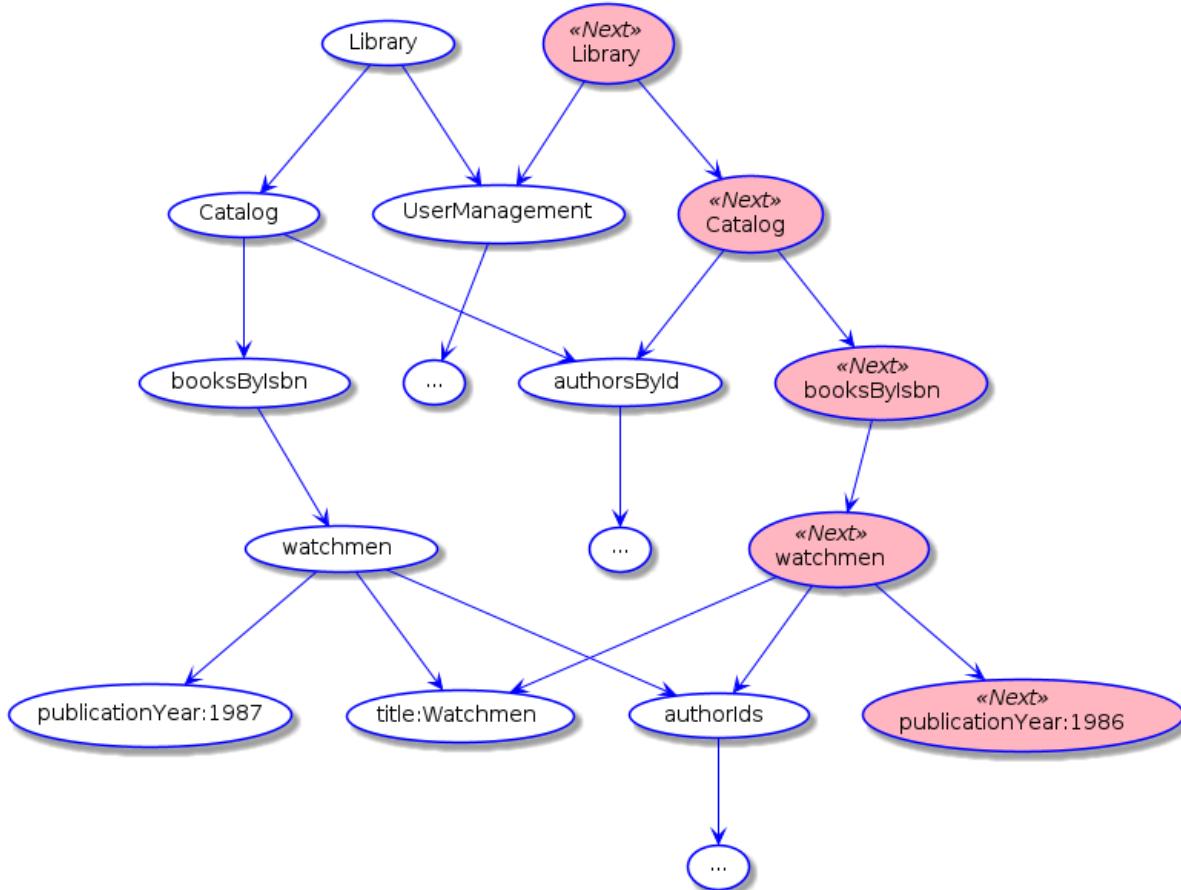
**JOE:** What part of the Book record?

**THEO:** Only the publicationYear field.

**JOE:** When you use an immutable function to create a new version of the `Library` where the publication year of `Watchmen` is set to 1986 (instead of 1987), it creates a fresh `Library` hash map that recursively uses the parts of the current `Library` that are common between the two versions instead of deeply copying them. This technique is called: structural sharing.

**THEO:** Could you describe me how structural sharing works step by step?

Joe grabs a piece of paper and draws the diagram in 4.3 that illustrates structural sharing.



**Figure 4.3** Structural sharing provides an efficient way to create a new version of the data: `Next Library` is recursively made of nodes that use the parts of `Library` that are common between the two.

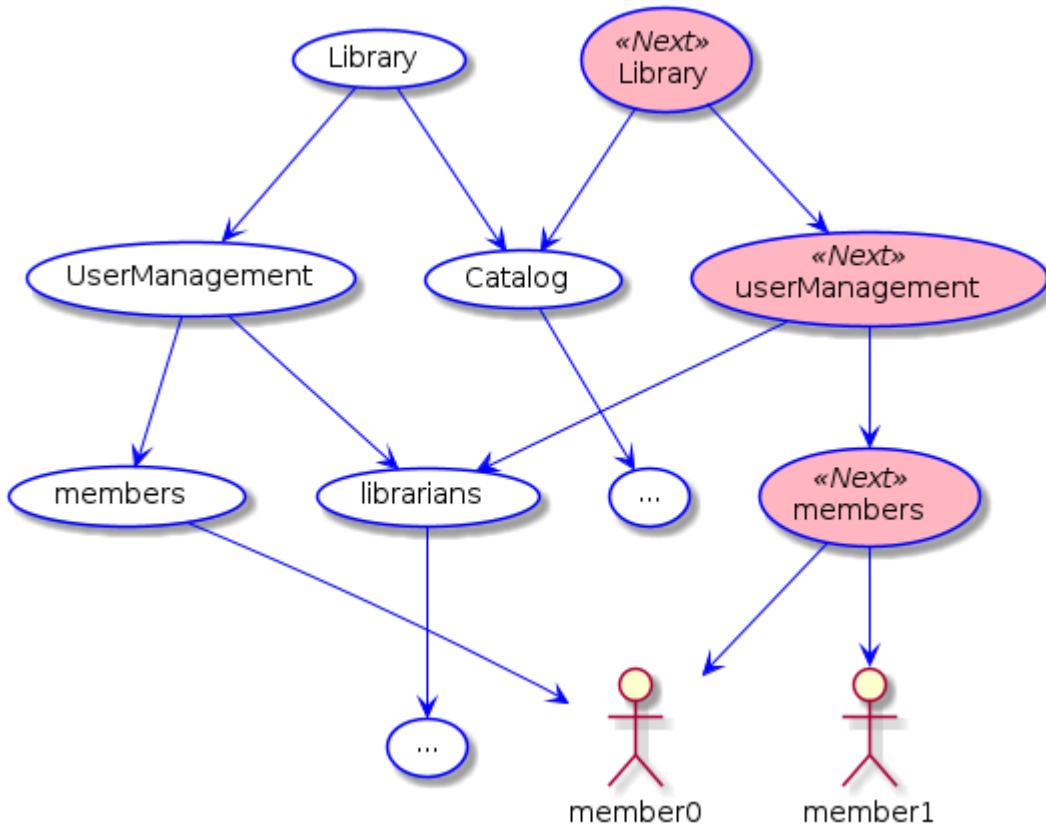
**JOE:** The next version of `Library`, uses the same `UserManagement` hash map as the old one. The `Catalog` inside the next `Library` uses the same `authorsById` as the current `Catalog`. The `Watchmen` Book record inside the next `Catalog` uses all the fields of the current `Book` except for the `publicationYear` field.

**TIP**

Structural sharing provides an efficient way (both memory and computation) to create a new version of the data by recursively sharing the parts that don't need to change.

**THEO:** That's very cool!

**JOE:** Indeed. Now let me show you how to write a mutation for adding a member using immutable functions. [4.4](#) shows a diagram that illustrates how structural sharing looks when we add a member.



**Figure 4.4 Adding a member with structural sharing: most of the data is shared between the two versions**

**THEO:** Awesome! The `Catalog` and the `librarians` hash maps don't have to be copied!

**JOE:** In terms of code, we have to write a `Library.addMember` function that delegates to `UserManagement.addMember`.

**THEO:** I guess it's going to be similar to the code we wrote earlier to implement the search books query, where `Library.searchBooksByTitleJSON` delegates to `Catalog.searchBooksByTitle`.

**JOE:** Similar in the sense that all the functions are static and they receive the data they manipulate as an argument. But there are two differences: First, A mutation could fail, for instance if the member to be added already exists. Secondly, the code for `Library.addMember` is a bit more elaborate than the code for `Library.searchBooksByTitleJSON` as we have to create a new version of the `Library` that refers to the new version of the `UserManagement`. [4.3](#) shows the code for the mutation that adds a member.

### **Listing 4.3 The code for the mutation that adds a member**

```
UserManagement.addMember = function(userManagement, member) {
  var email = _.get(member, "email");
  var infoPath = ["membersByEmail", email];
  if(_.has(userManagement, infoPath)) { ①
    throw "Member already exists.";
  }
  var nextUserManagement = _.set(userManagement, ②
    infoPath,
    member);
  return nextUserManagement;
};

Library.addMember = function(library, member) {
  var currentUserManagement = _.get(library, "userManagement");
  var nextUserManagement = UserManagement.addMember(currentUserManagement, member);
  var nextLibrary = _.set(library, "userManagement", nextUserManagement); ③
  return nextLibrary;
};
```

- ① Check if a member already exists with the same email address
- ② Create a new version of `userManagement` that includes the member
- ③ Create a new version of `library` that contains the new version of `userManagement`

**THEO:** It's a bit weird to me that immutable functions return an updated version of the data instead of changing it in place.

**JOE:** It was also weird for me when I first encountered immutable data in Clojure 7 years ago.

**THEO:** How long did it take you to get used to it?

**JOE:** A couple of weeks.

## **4.4 Implementation of structural sharing**

When Joe leaves the office, Theo meets Dave near the coffee machine.

**DAVE:** Who is this guy that just left the office?

**THEO:** It's Joe. My DOP coach.

**DAVE:** What's DOP?

**THEO:** Data-Oriented programming.

**DAVE:** I never heard that term before.

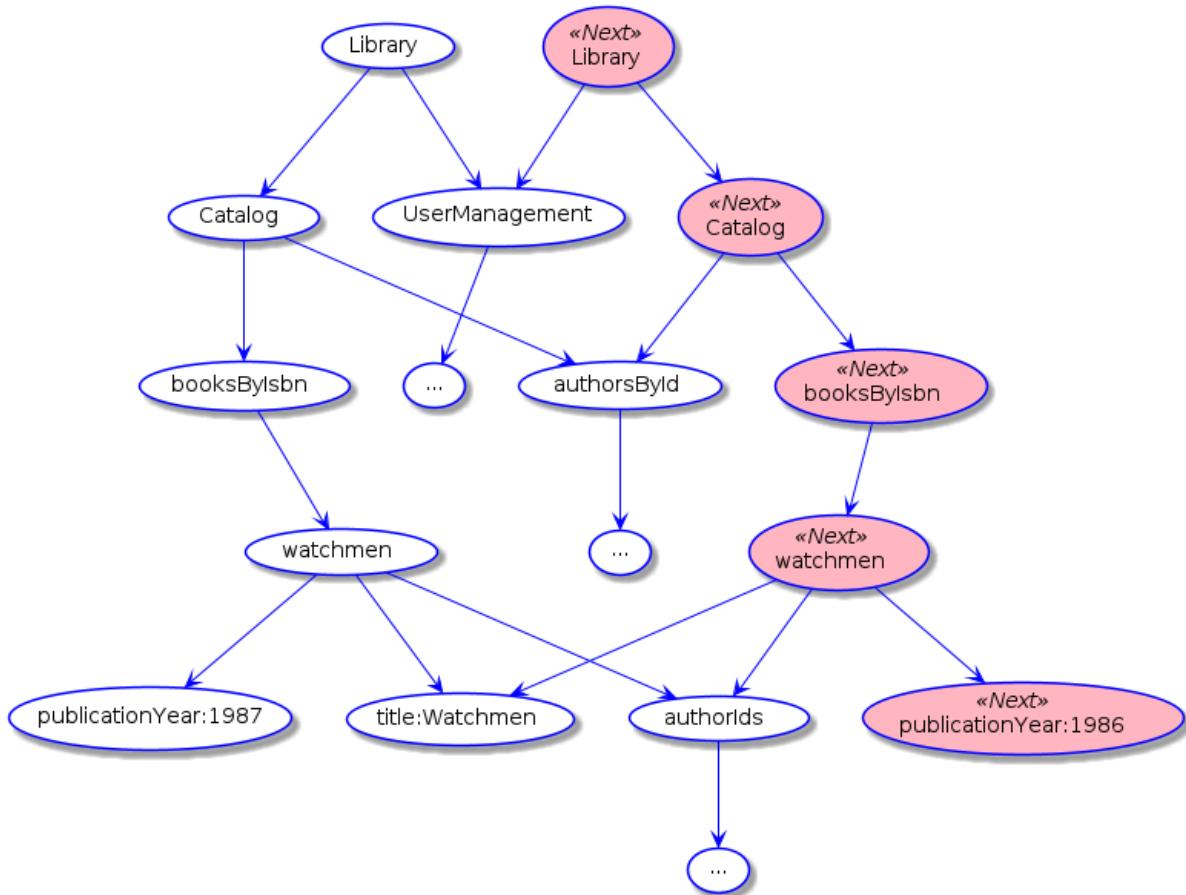
**THEO:** It's not very well known by programmers yet, but it's quite a powerful programming paradigm. From what I've seen so far, it makes programming simpler.

**DAVE:** Can you give me an example?

**THEO:** I just learned about structural sharing and how it makes it possible to create new versions of data without copying in an effective way.

**DAVE:** How does that work?

Theo shows Dave the structural sharing diagram that Joe drew on the whiteboard a couple of minutes ago, as in [4.5](#). It takes Theo a few minutes to explain to Dave what it does exactly but at the end he gets it.



**Figure 4.5 Structural sharing in action**

**DAVE:** What does the implementation of structural sharing look like?

**THEO:** I don't know, I use `_.set` from Lodash.

**DAVE:** It sounds like an interesting challenge.

**THEO:** Take the challenge if you want. I'm too tired for this recursive algorithmic stuff.

The next day, Dave comes to Theo's room and he shows him with a touch of pride, his implementation of structural sharing, as in [4.4](#). Theo is amazed by the fact that it's only 11 lines of JavaScript code!

#### Listing 4.4 The implementation of structural sharing

```
function setImmutable(map, path, v) {
  var modifiedNode = v;
  var k = path[0];
  var restOfPath = path.slice(1);
  if (restOfPath.length > 0) {
    modifiedNode = setImmutable(map[k], restOfPath, v);
  }
  var res = Object.assign({}, map); ①
  res[k] = modifiedNode;
  return res;
}
```

- ① That's how we shallow clone a map in JavaScript

**THEO:** Dave, you're brilliant!

**DAVE:** Aww, shucks. <smiling>

**THEO:** I have to go. I'm already late for my session with Joe!

## 4.5 Data safety

When Joe is about to start the day's lesson, Theo asks him a question about yesterday's material.

**THEO:** Something isn't clear to me regarding this structural sharing stuff. What happens if we write code that modifies the data part that's shared between the two versions of the data? Does the change affect both versions?

**JOE:** Could you please write a code snippet that illustrates your question?

Theo starts typing on his laptop, and he comes up with the code snippet in [4.5](#) that illustrates his point.

## Listing 4.5 A piece of code that modifies a piece of data that is shared between two versions

```

var books = {
  "978-1779501127": {
    "isbn": "978-1779501127",
    "title": "Watchmen",
    "publicationYear": 1987,
    "authorIds": ["alan-moore",
      "dave-gibbons"]
  }
};

var nextBooks = _.set(books, ["978-1779501127", "publicationYear"], 1986)

console.log("Before:", nextBooks["978-1779501127"]["authorIds"][1]);

books["978-1779501127"]["authorIds"][1] = "dave-chester-gibbons";

console.log("After:", nextBooks["978-1779501127"]["authorIds"][1]);
//Before: dave-gibbons
//After: dave-chester-gibbons

```

**THEO:** My question is: what is the value of `isBlocked` in `updatedMember`?

**JOE:** The answer is that mutating data via the native hash map setter is forbidden. All the data manipulation must be via immutable functions.

**WARNING** All data manipulation must be done via immutable functions: It is forbidden to use the native hash map setter.

**THEO:** When you say *forbidden* you mean that it's up to the developer to make sure it doesn't happen. Right?

**JOE:** Exactly.

**THEO:** Is there a way to protect our system from a developer's mistake?

**JOE:** Yes, there is a way to ensure the immutability of the data at the level of the data structure. It's called persistent data structures.

**THEO:** Are persistent data structures also efficient in terms of memory and computation?

**JOE:** Actually, the way data is organized inside persistent data structures make them even more efficient than immutable functions.

**TIP** Persistent data structures are immutable at the level of the data: There is no way to mutate them (even by mistake).

**THEO:** Are there libraries providing persistent data structures?

**JOE:** Definitely. For example, we have Immutable.js in JavaScript (<https://immutable-js.com/>), Paguro in Java (<https://github.com/GlenKPeterson/Paguro>), Immutable Collections in C#,<sup>6</sup> Pyrsistent in Python (<https://github.com/tobgu/pyrsistent>) and Hamster in Ruby (<https://github.com/hamstergem/hamster>).

**THEO:** So why not using persistent data structures instead of immutable functions?

**JOE:** The drawback of persistent data structures is that they are not native which means that working with them requires conversion from native to persistent and from persistent to native.

**THEO:** What approach would you recommend then?

**JOE:** If you want to play around a bit then start with immutable functions. But for a production application I'd recommend using persistent data structures.

**THEO:** Too bad the native data structures aren't persistent!

**JOE:** That's one of the reasons why I love Clojure: the native data structures of the language are immutable!

## 4.6 The commit phase of a mutation

So far we've seen how to implement the calculation phase of a mutation. The calculation phase is stateless, in the sense that it doesn't make any change to the system. Now, we're going to see how to update the state of the system inside the commit phase.

Theo takes another look at the code for `Library.addMember` in [4.6](#) and something bothers him: this function returns a new state of the library that contains an additional member but it doesn't affect the current state of the library!

### Listing 4.6 The calculation phase of a mutation doesn't make any change to the system

```
Library.addMember = function(library, member) {
  var currentUserManagement = _.get(library, "userManagement");
  var nextUserManagement = UserManagement.addMember(currentUserManagement, member);
  var nextLibrary = _.set(library, "userManagement", nextUserManagement);
  return nextLibrary;
};
```

**THEO:** I see that `Library.addMember` doesn't change the state of the library. How does the library state get updated then?

**JOE:** That's an excellent question. `Library.addMember` deals only with data calculation and is stateless. The state is updated in the commit phase by moving forward the version of the state that the system state refers to.

**THEO:** What do you mean?

**JOE:** Here's what happens when we add a member to the system. The calculation phase creates a version of the state that has two members. Before the commit phase, the system state refers to the version of the state with one member. The responsibility of the commit phase is to move the system state forward so that it refers to the version of the state with two members.

**TIP**

The responsibility of the commit phase is to move the system state forward to the version of the state returned by the calculation phase.

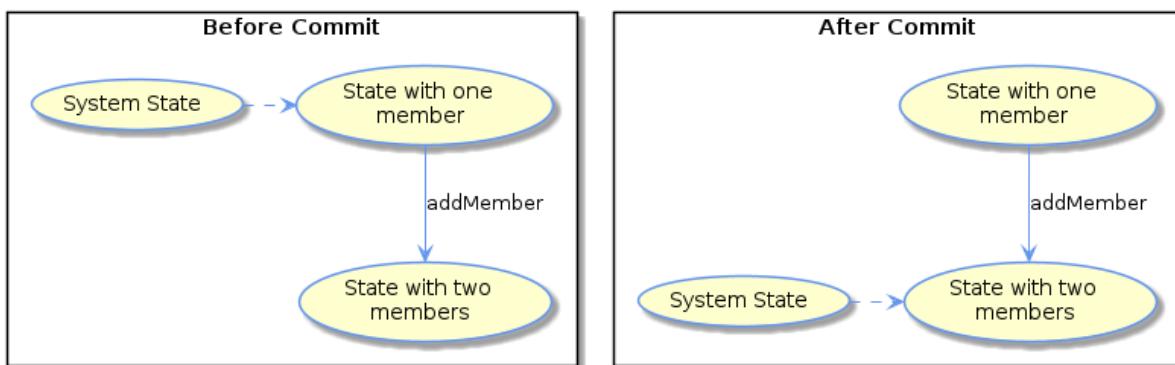


Figure 4.6 The commit phase moves the system state forward

**THEO:** How is that implemented?

**JOE:** The code is made of 2 classes: `System`, a singleton stateful class that implements the mutations. `SystemState` a singleton stateful class that manages the system state.

**THEO:** It sounds to me like classic OOP.

**JOE:** Right. This part of the system being stateful is very OOP-like.

**THEO:** I'm happy to see that you still find some utility in OOP.

**JOE:** Meditation taught me that *Every part of our universe has a role to play*.

**THEO:** Nice! Could you show me some code?

**JOE:** Sure. Here's the `System` class: [4.7](#) and its implementation of the `addMember` mutation.

**Listing 4.7 The `System` class**

```

class System {
    addMember(member) {
        var previous = SystemState.get();
        var next = Library.addMember(previous, member);
        SystemState.commit(previous, next); ①
    }
}

```

- ① `SystemState` is covered below

**THEO:** What does `SystemState` look like?

**JOE:** Here's the code for the `SystemState` class: [4.8](#) It's a stateful class!

#### **Listing 4.8 The `SystemState` class**

```
class SystemState {
    systemState;

    get() {
        return this.systemState;
    }

    commit(previous, next) {
        this.systemState = next;
    }
}
```

**THEO:** I don't get the point of the `SystemState`. It's a simple class with a getter and a commit function!

**JOE:** In a moment, we are going to enrich the code of the `SystemState.commit` method so that it provides data validation and history tracking. For now, the important thing to notice is that the code of the calculation phase is stateless and is decoupled from the code of the commit phase which is stateful.

**TIP**

The calculation phase is stateless. The commit phase is stateful.

## 4.7 Ensure system state integrity

**THEO:** Something still bothers me with the way functions manipulate immutable data in the calculation phase: How do we preserve *data integrity*?

**JOE:** What do you mean?

**THEO:** In OOP, data is manipulated only by methods that belong to the same class as the data. It prevents other classes from corrupting the inner state of the class.

**JOE:** Could you give me an example of an invalid state of the library?

**THEO:** For example, imagine that the code of a mutation adds a book item to the book lendings of a member without marking the book item as lent in the catalog. Then the system data would be corrupted.

**JOE:** In DOP, we have the privilege of ensuring data integrity at the level of the whole system

instead of scattering the validation among many classes.

**THEO:** How does that work?

**JOE:** The fact that the code for the commit phase is common to all the mutations allows us to validate the system data in a central place: At the beginning of the commit phase there is a step that checks (see [4.9](#)) whether the version of the system state to be committed is valid. If the data is invalid, the commit is rejected.

#### **Listing 4.9 Data validation inside the commit phase**

```
SystemState.commit = function(previous, next) {
    if(!SystemValidity.validate(previous, next)) { // not implemented for now
        throw "The system data to be committed is not valid!";
    }
    this.systemData = next;
};
```

**THEO:** It sounds similar to a commit hook in git.

**JOE:** I like your analogy!

**THEO:** Why are you passing the previous state in `previous` and the next state in `next`to`SystemValidity.validate?`

**JOE:** Because it allows `SystemValidity.validate` to optimize the validation in terms of computation. For example, we could validate just the data that has changed.

**TIP**

**In DOP, we validate the system data as a whole. Data validation is decoupled from data manipulation.**

**THEO:** What does the code of `SystemValidity.validate` look like?

**JOE:** Some day, I will show you how to define a data schema and to validate that a piece of data conforms to a schema (See Chapters 7 and 12).

## 4.8 Time travel

Another advantage of the multi-version state approach with immutable data that is manipulated via structural sharing is that we can keep track of the history of all the versions of the data without exploding the memory of our program. It allows us, for instance, to restore the system back to an earlier state very easily.

**THEO:** You told me earlier that it was easy to restore the system to a previous state. Could you show me how?

**JOE:** Happily. But before that, I'd like to make sure you understand why keeping track of all the versions of the data is efficient in terms of memory.

**THEO:** I think it's related to the fact that immutable functions use structural sharing. And most of the data between subsequent versions of the state is shared.

**TIP**

**Structural sharing allows us to keep many versions of the system state without exploding memory use.**

**JOE:** Perfect. Now, I'm going to show you how simple it is to *undo* a mutation. In order to implement *undo*, our `SystemState` class needs to have two references to the system data: `systemData` references the current state of the system and `previousSystemData` references the previous state of the system.

**THEO:** That makes sense.

**JOE:** In the commit phase, we update both `previousSystemData` and `systemData`.

**THEO:** And what does it take to implement *undo*?

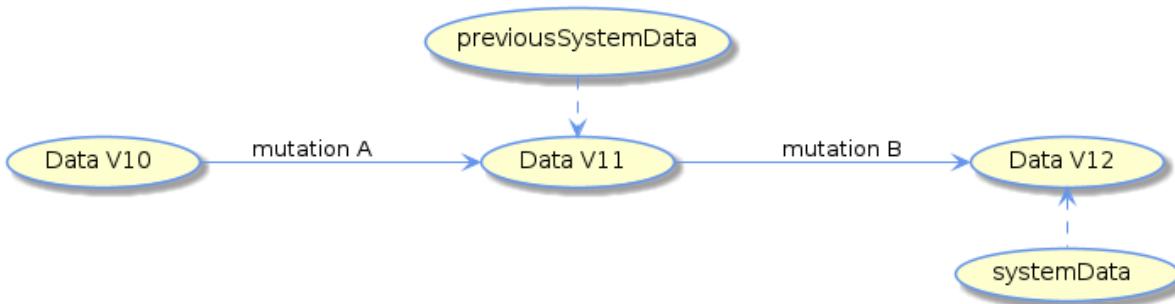
**JOE:** Undo is achieved by having `systemData` referencing the same version of the system data as `previousSystemData`.

**THEO:** Could you walk me through an example?

**JOE:** To make things simple, I am going to give a number to each version of the system state. It starts at `v0` and each time a mutation is committed the version is incremented: `v1`, `v2`, `v3` etc...

**THEO:** OK.

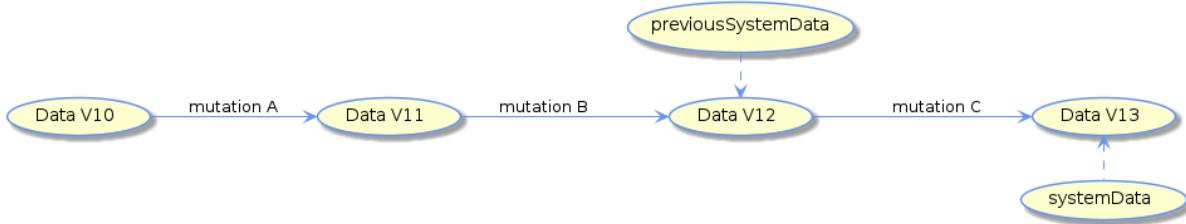
**JOE:** Let's say that currently our system state is at `v12` (see [4.7](#)). In the `SystemState` object, `systemData` refers to `v12` and `previousSystemData` refers to `v11`.



**Figure 4.7 When the system state is at v12, systemData refers to v12 and previousSystemData refers to v11**

**THEO:** So far so good.

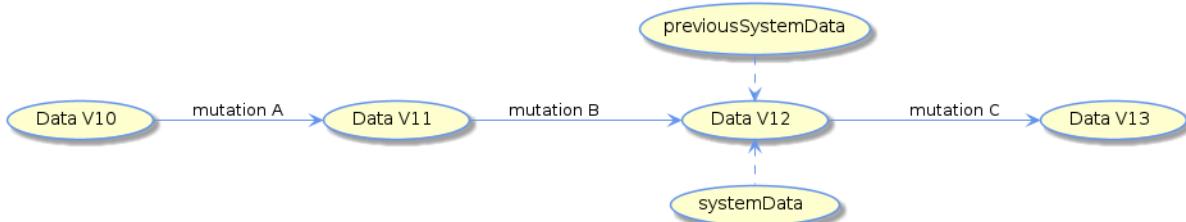
**JOE:** Now when a mutation is committed (for instance adding a member), both references move forward: `systemData` refers to v13 and `previousSystemData` refers to v12



**Figure 4.8** When a mutation is committed, `systemData` refers to v13 and `previousSystemData` refers to v12

**THEO:** And I suppose that when we undo the mutation, both references move backward.

**JOE:** In theory, yes. But in practice, it would be necessary to maintain a stack of all the state references. For now, to simplify things, we maintain only a reference to the previous version. As a consequence, when we undo the mutation, both references refer to v12 as shown in [4.9](#).



**Figure 4.9** When a mutation is undone, both `systemData` and `previousSystemData` refer to v12

**THEO:** Could you show me how to implement this undo mechanism?

**JOE:** Actually, it takes only a couple of changes to the `SystemState` class. The result is in [4.10](#). Pay attention to the changes in the `commit` function: Inside `systemDataBeforeUpdate` we keep a reference to the current state of the system. If the validation and the conflict resolution succeed, we update both `previousSystemData` and `systemData`.

### Listing 4.10 The `SystemState` class with undo capability

```

class SystemState {
    systemData;
    previousSystemData;

    get() {
        return this.systemData;
    }

    commit(previous, next) {
        var systemDataBeforeUpdate = this.systemData;
        if(!Consistency.validate(previous, next)) {
            throw "The system data to be committed is not valid!";
        }
        this.systemData = next;
        this.previousSystemData = systemDataBeforeUpdate;
    }

    undoLastMutation() {
        this.systemData = this.previousSystemData;
    }
}

```

**THEO:** And I see that implementing `System.undoLastMutation` is simply a matter of having `systemData` refer the same value as `previousSystemData`.

**JOE:** As I told you, if we need to allow multiple undos, the code would be a bit more complicated. But you get the idea.

**THEO:** I think so. "Back to the future" belongs to the realm of science fiction, but in DOP time travel is real.

## 4.9 Summary

- DOP Principle #3: Data is immutable.
- A *mutation* is an operation that changes the state of the system.
- In a multi-version approach to state management, mutations are split into calculation and commit phases.
- All data manipulation must be done via immutable functions: It is forbidden to use the native hash map setter.
- Structural sharing allows us to efficiently (memory and computation) create new versions of data where data that is common between the two versions is shared instead of being copied.
- Structural sharing creates a new version of the data by recursively sharing the parts that don't need to change.
- A mutation is split in two phases: calculation and commit.
- A function is said to be immutable when instead of mutating the data, it creates a new version of the data without changing the data it receives.
- During the calculation phase, data is manipulated with immutable functions that leverage structural sharing.
- The calculation phase is stateless.
- The responsibility of the commit phase is to move the system state forward to the version of the state returned by the calculation phase.
- During the commit phase, we update the system state.
- The data is immutable but the state reference is mutable.
- The commit phase is stateful.
- We validate the system data as a whole. Data validation is decoupled from data manipulation.
- The fact that the code for the commit phase is common to all the mutations, allows us to validate the system state in a central place before we update the state.
- Keeping the history of the versions of the system data is memory efficient due to structural sharing.
- Restoring the system to one of its previous states is straightforward due to the clear separation between the calculation phase and the commit phase.
- In order to use Lodash immutable functions, we use Lodash's FP module.

**Table 4.2 The two phases of a mutation**

Phase	Responsibility	State	Implementation
calculation	Compute next version of system data	Stateless	Specific
commit	Move forward the system state	Stateful	Common

**Table 4.3 Lodash functions introduced in this chapter**

Function	Description
<code>set(map, path, value)</code>	Creates a map with the same fields as <code>map</code> with the addition of a field <code>&lt;path, value&gt;</code>



# Basic concurrency control

## This chapter covers

- Managing concurrent mutations with a lock-free optimistic concurrency control strategy
- Supporting high throughput of reads and writes
- Reconciliation between concurrent mutations

## 5.1 Conflicts at home

The changes required to let our system manage concurrency are only in the commit phase. They involve a reconciliation algorithm that is universal in the sense that it can be used in any system where system data is represented as an immutable hash map.

The implementation of the reconciliation algorithm is efficient as it leverages the fact that subsequent versions of the system state are created via structural sharing.

In the previous chapter, we illustrated the multi-version approach to state management where a mutation is split into two distinct phases: the calculation phase that deals only with computation and the commit phase that moves the state reference forward.

Usually, in a production system, mutations occur concurrently. Moving the state forward naively like we did in the previous chapter is not appropriate. In the present chapter, we are going to learn how to handle concurrent mutations.

In DOP, the fact that only the code of the commit phase is stateful allows us to leverage an *Optimistic Concurrency Control* strategy that doesn't involve any locking mechanism. As a consequence, the throughput of reads and writes is high.

The modifications to the code are not trivial, as we have to implement an algorithm that reconciles concurrent mutations. But the modifications impact only the commit phase. The code for the calculation phase stays the same as in the previous chapter.

**WARNING** This chapter requires more of an effort to grasp: The flow of the reconciliation algorithm is definitely not trivial and the implementation involves a non-trivial recursion.

## 5.2 Optimistic Concurrency Control

This morning, before getting to work, Theo takes Joe to the fitness room of the office and while running on the step machine, the two men talk about their personal lives. Joe talks about a fight he had last night with Kay, who thinks that he pays more attention to his work than to his family. Theo recounts the painful conflict he had with Jane, his wife, about house budget management. They went to see a therapist, an expert in Imago Therapy. Imago allowed them to transform their conflict into an opportunity to grow and heal.

Joe's ears perk up when he hears the word *conflict*, as today's lesson is going to be about resolving conflicts between concurrent mutations. A different kind of conflict, though...

After a shower and a healthy breakfast, Theo and Joe get down to work.

**JOE:** Last week, I showed you how to manage state with immutable data, assuming that no mutations occur concurrently. Today, I am going to show you how to deal with concurrency control in DOP.

**THEO:** I'm curious to discover what kind of lock mechanisms you use in DOP to synchronize between concurrent mutations.

**JOE:** In fact, we don't use any lock mechanism!

**THEO:** Why not?

**JOE:** Locks hit performance and if you're not careful, your system could get into a deadlock.

**THEO:** So how do you handle possible conflicts between concurrent mutations in DOP?

**JOE:** In DOP, we use a lock-free strategy called "Optimistic Concurrency Control". It's a strategy that allows databases like Elasticsearch (<https://www.elastic.co/elasticsearch/>) to be highly scalable.

**THEO:** You sound like my couples therapist and her anger-free optimistic conflict resolution strategy...

**JOE:** Optimistic Concurrency Control and DOP fit very well. As you will see in a moment, Optimistic Concurrency Control is super efficient when the system data is immutable.

**TIP**      **Optimistic Concurrency Control with immutable data is super efficient.**

**THEO:** How does it work?

**JOE:** Optimistic concurrency control is when we let mutations ask forgiveness instead of permission.

**THEO:** What do yo mean?

**TIP**      **Optimistic concurrency control is when we let mutations ask forgiveness instead of permission.**

**JOE:** The calculation phase does its calculation as if it were the only mutation running. The commit phase is responsible for reconciling concurrent mutations when they don't conflict, or aborting the mutation.

**TIP**      **The calculation phase does its calculation as if it were the only mutation running.**

**TIP**      **The commit phase is responsible for trying to reconcile concurrent mutations.**

**THEO:** That sounds quite challenging to implement!

**JOE:** Dealing with state is never trivial. But the good news is that the code for the reconciliation logic in the commit phase is universal.

**THEO:** Does that mean that the same code for the commit phase can be used in any DOP system?

**JOE:** Definitely. The code that implements the commit phase assumes nothing about the details of the system except that the system data is represented as an immutable map.

**TIP**      **The implementation of the commit phase in Optimistic Concurrency Control is universal: It can be used in any system where the data is represented by an immutable hash map.**

**THEO:** That's awesome!

**JOE:** Another cool thing is that handling concurrency doesn't require any changes to the code in the calculation phase. From the calculation phase perspective, the next version of the system data is computed in isolation as if no other mutations were running concurrently (see [5.1](#) and [5.1](#)).

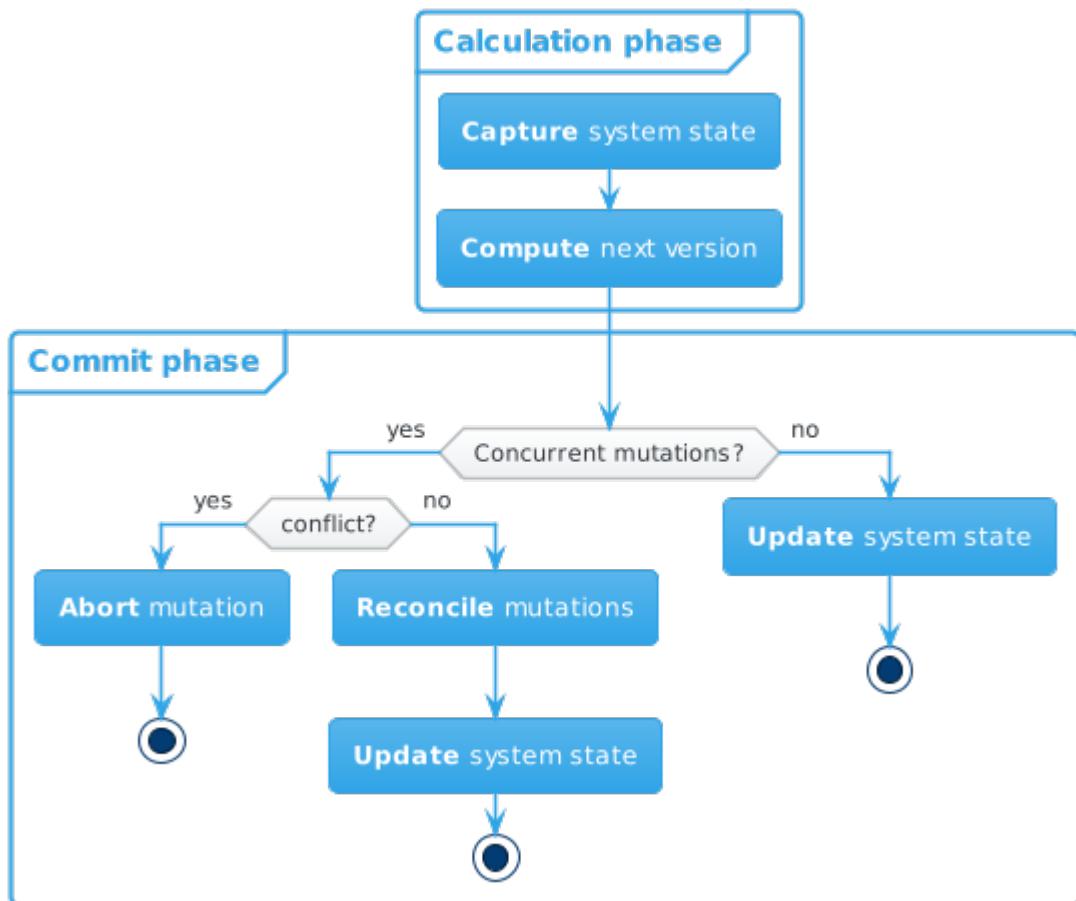


Figure 5.1 The logic flow of Optimistic Concurrency Control

Table 5.1 The two phases of a mutation with Optimistic Concurrency Control

Phase	Responsibility	State	Implementation
Calculation	Compute next state in isolation	Stateless	Specific
Commit	Reconcile and update system state	Stateful	Common

### 5.3 Reconciliation between concurrent mutations

**THEO:** Could you give me some examples of conflicting concurrent mutations?

**JOE:** Two members trying to borrow the same book copy. Or two librarians updating the publication year of the same book.

**THEO:** And what do you mean exactly by *reconciliation* between possible concurrent mutations?

**JOE:** It's quite similar to what could happen in git when you merge a branch back into the main branch.

**THEO:** I love it when the main branch stays the same.

**JOE:** Yes, it's nice when the merge has no conflicts and can be done automatically. Do you remember how git handles the merge in that case?

**THEO:** Git does a *fast-forward*: it updates the main branch to be the same as the merge branch.

**JOE:** Right. And what happens when you discover that meanwhile another developer has committed their code to the main branch?

**THEO:** Then git does a *3-way merge*, trying to combine all the changes from the 2 merge branches with the main branch.

**JOE:** Does it always go smoothly?

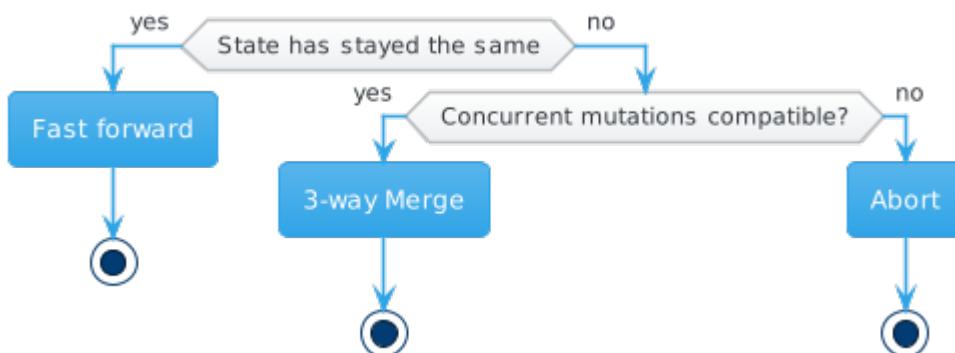
**THEO:** Usually, yes. But it's possible that two developers have modified the same line in the same file. Then I have to manually resolve the conflict. I hate when it happens!

**TIP**

In a production system, multiple mutations run concurrently. Before updating the state, we need to reconcile the conflicts between possible concurrent mutations.

**JOE:** In DOP, the reconciliation algorithm in the commit phase is quite similar to a merge in git, except that instead of a manual conflict resolution, we abort the mutation. There are three possibilities to reconcile between possible concurrent mutations: fast-forward, 3-way merge or abort.

Joe goes to the whiteboard and draws a diagram, shown in [5.2](#).



**Figure 5.2** The reconciliation flow

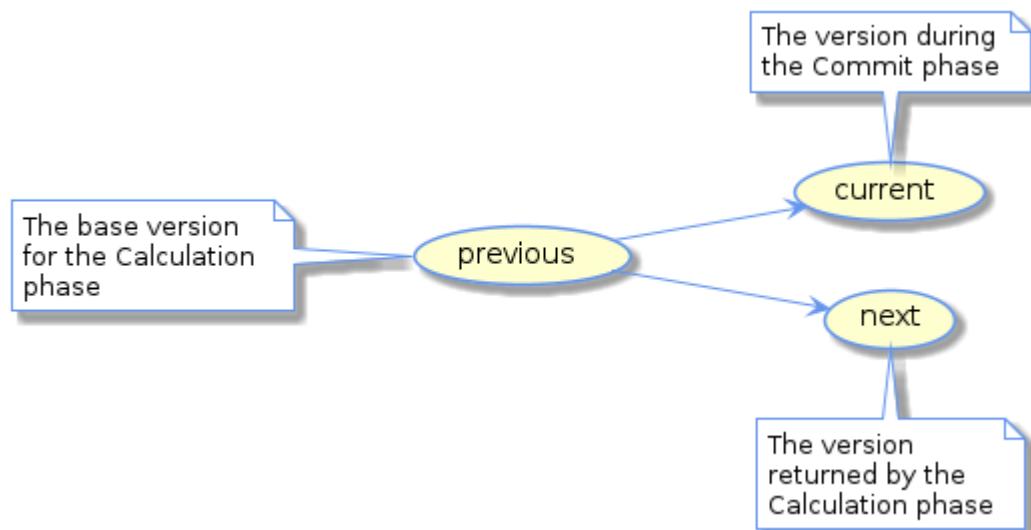
**THEO:** Could you explain in more detail?

**JOE:** When the commit phase of a mutation starts, we have 3 versions of the system state (see [5.3](#)): `previous` - the version on which the calculation phase based its computation, `current` - the current version during the commit phase and `next` - the version returned by the calculation phase.

**THEO:** Why would `current` be different than `previous`?

**JOE:** It happens when other mutations have run concurrently with our mutation.

**THEO:** I see.



**Figure 5.3 When the commit phase starts, there are three versions of the system state.**

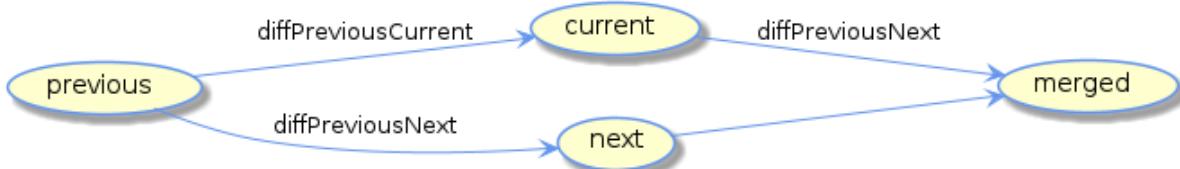
**JOE:** If we are in a situation where the current state is the same as the previous state, it means that no mutations run concurrently. Therefore, as in git, we can safely fast-forward and update the state of the system with the next version.

**THEO:** And if the state has *not* stayed the same?

**JOE:** Then it means that mutations have run concurrently. We have to check for conflicts in a way similar to the 3-way merge used by git. The difference is that instead of comparing lines, we compare fields of the system hash map.

**THEO:** Could you explain that?

**JOE:** We calculate the diff between `previous` and `next` and between `previous` and `current`. If the two diffs have no fields in common, then there is no conflict between the mutations that have run concurrently. We can safely apply the changes from `previous` to `next` into `current`. (See [5.4](#).)



**Figure 5.4** In a 3-way merge, we calculate the diff between `previous` and `next` and we apply it to `current`

**THEO:** And if there is a conflict?

**JOE:** Then we abort the mutation.

**THEO:** Aborting a user request seems unacceptable.

**JOE:** In fact, in a user facing system, conflicting concurrent mutations are fairly rare: that's why it's OK to abort and let the user run the mutation again.

**Table 5.2** The analogy between git and Data-Oriented Programming

Data-Oriented Programming	git
concurrent mutations	different branches
a version of the system data	a commit
state	a reference
calculation phase	branching
validation	pre-commit hook
reconciliation	merge
fast-forward	fast-forward
3-way merge	3-way merge
abort	manual conflict resolution
hash map	tree (folder)
leaf node	blob (file)
data field	line of code

**THEO:** Well in cases where two mutations update the same field of the same entity, I think it's fair enough to let the user know that the request couldn't be processed.

**TIP**

In a user facing system, conflicting concurrent mutations are fairly rare.

## 5.4 Reducing collections

**JOE:** Are you ready to challenge your mind with the implementation of the diff algorithm?

**THEO:** Let's make a short coffee break before, if you don't mind.

After enjoying large mug of hot coffee and a few butter cookies, Theo and Joe are back to work...

**JOE:** In the implementation of the diff algorithm, we're going to reduce collections.

**THEO:** I have heard about reduce in a talk about FP. But I don't remember the details. Could you remind me how reduce works?

**JOE:** Imagine you want to calculate the sum of the elements in a collection of numbers. With `reduce`, it would look like this:

Joe writes the code in [5.1](#) on the white board.

### Listing 5.1 Summing numbers with `reduce`

```
_.reduce([1, 2, 3], function(res, elem) {
  return res + elem;
}, 0);
// 6
```

**THEO:** I don't understand.

Joe comes to the whiteboard and starts writing the description of `_.reduce`.

#### SIDE BAR

#### Description of `reduce`

`reduce` receives 3 arguments:

1. `coll` - a collection of elements
2. `f` - a function that receives 2 arguments
3. `initVal` - a value

Logic flow:

1. Initialize `currentRes` with `initVal`
2. For each element `x` of `coll`, update `currentRes` with `f(currentRes, x)`
3. Return `currentRes`

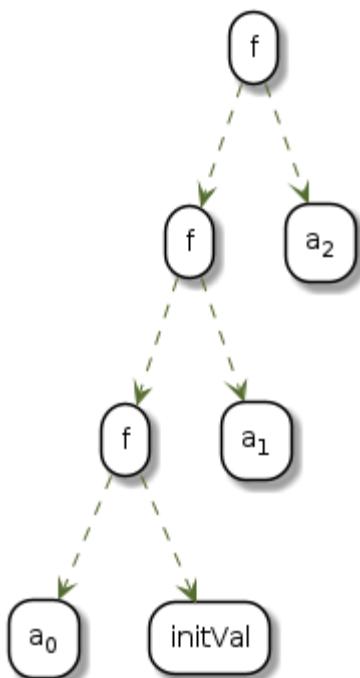
**THEO:** Would you mind if I expand manually the logic flow of that code we just wrote using `reduce`? ([5.1](#))

**JOE:** I think it's a great idea!

**THEO:** In our case, `initVal` is 0. It means that the first call to `f` will be `f(0, 1)`. Then, we'll have `f(f(0, 1), 2)` and finally `f(f(f(0, 1), 2), 3)`.

**JOE:** I like your manual expansion Theo! Let me make it visual.

Theo comes to the whiteboard and draw a diagram like the one in [5.5](#).



**Figure 5.5 Visualization of reduce**

**THEO:** It's much clearer now. I think that by implementing my custom version of `_.reduce`, it will make things 100% clear.

It takes Theo much less time than he expected to implement `reduce` as in [5.2](#).

### Listing 5.2 Custom implementation of `reduce`

```

function reduce(coll, f, initVal) {
  var currentRes = initVal;
  for (var i = 0; i < coll.length; i++) { ①
    currentRes = f(currentRes, coll[i])
  }
  return currentRes;
}
  
```

- ① We could use `forEach` instead of a `for` loop

After checking that Theo's code works as expected with the code in [5.3](#), Joe is proud of Theo.

### Listing 5.3 Testing the custom implementation of `reduce`

```

reduce([1, 2, 3], function(res, elem) {
  return res + elem;
}, 0);
// 6
  
```

**JOE:** Well done!

## 5.5 Structural diff

**WARNING** This section deals with the implementation of a structural diff algorithm. Feel free to skip this section if you don't want to challenge your mind right now with the details of a sophisticated use of recursion. It won't prevent you from enjoying the rest of the book. You can come back to this section later.

**THEO:** How do you calculate the diff between various versions of the system state?

**JOE:** That's the most challenging part of the reconciliation algorithm. We need to implement a structural diff algorithm for hash maps.

**THEO:** In what sense is the diff *structural*?

**JOE:** The structural diff algorithm looks at the structure of the hash maps and ignores the order of the fields.

**THEO:** Could you give me an example?

**JOE:** Let's start with maps without nested fields. Basically, there are three kinds of diffs: field replacement, field addition and field deletion. In order to make things not too complicated, we are going to deal only with replacement and addition.

Joe comes to the whiteboard and draws a table like the one in [5.3](#).

**Table 5.3 Kinds of structural differences between maps without nested fields**

Kind	First map	Second map	Diff
Replacement	{ "a": 1 }	{ "a": 2 }	{ "a": 2 }
Addition	{ "a": 1 }	{ "a": 1, "b": 2 }	{ "a": 2 }
Deletion	{ "a": 1, "b": 2 }	{ "a": 1 }	Not supported

**THEO:** I notice that the order of the maps matter a lot. What about nested fields?

**JOE:** It's the same idea but the nesting makes it a bit more difficult to grasp.

Joe adds more rows to the table (see [5.4](#)).

**Table 5.4** Kinds of structural differences between maps with nested fields

Kind	First map	Second map	Diff
Replacement	{ "a": { "x": 1 } }	{ "a": { "x": 2 } }	{ "a": { "x": 2 } }
Addition	{ "a": { "x": 1 } }	{ "a": { "x": 1, "y": 2, } }	{ "a": { "y": 2 } }
Deletion	{ "a": { "x": 1, "y": 2, } }	{ "a": { "y": 2 } }	Not supported

**WARNING** This version of the structural diff algorithm does not deal with deletions. Dealing with deletions is definitely possible but it requires a more complicated algorithm.

**THEO:** As you said. It's harder to grasp. What about arrays?

**JOE:** We compare the elements of the arrays in order: if they are equal the diff is `null` and if they differ, the diff has the value of the second array (see [5.5](#)).

**Table 5.5** Kinds of structural differences between arrays without nested elements

Kind	First array	Second array	Diff
Replacement	[1]	[2]	[2]
Addition	[1]	[1, 2]	[null, 2]
Deletion	[1, 2]	[1]	Not supported

**THEO:** This usage of `null` is a bit weird but OK. Is it complicated to implement the structural diff algorithm?

**JOE:** Definitely. It took a good dose of mental gymnastics to come up with these 30 lines of code (see [5.4](#)).

### Listing 5.4 The implementation of structural diff

```

function diffObjects(data1, data2) {
    var emptyObject = _.isArray(data1) ? [] : {};
    if(data1 == data2) {
        return emptyObject;
    }
    var keys = _.union(_.keys(data1), _.keys(data2)); ②
    return _.reduce(keys,
        function (acc, k) {
            var res = diff(
                _.get(data1, k),
                _.get(data2, k));
            if(_.isObject(res) && _.isEmpty(res)) || ③
                (res == "no-diff")) { ⑤
                return acc;
            }
            return _.set(acc, [k], res);
        },
        emptyObject);
}

function diff(data1, data2) {
    if(_.isObject(data1) && _.isObject(data2)) { ④
        return diffObjects(data1, data2);
    }
    if(data1 !== data2) {
        return data2;
    }
    return "no-diff"; ⑤
}

```

- ① `_.isArray` checks whether its argument is an array
- ② `_.union` creates an array of unique values from two arrays (like union of two sets in Maths)
- ③ `_.isObject` checks whether its argument is a collection (either a map or an array). `_.isEmpty` checks whether its argument is an empty collection
- ④ `_.isEmpty` checks whether its argument is an empty collection
- ⑤ "no-diff" is how we mark that two values are the same.

**THEO:** Wow! It involves a recursion inside a reduce! I'm sure Dave will love it, but I'm too tired to understand this code right now. Let's focus on *what* it does instead of *how* it does it.

In order familiarize himself with the structural diff algorithm, Theo runs the algorithm with examples from the table that Joe drew on the whiteboard (see [5.5](#)). While Theo occupies his fingers with more and more complicated examples, his mind wanders in the realm of performance...

### Listing 5.5 An example of usage of structural diff

```

var data1 = {
  "a": {
    "x": 1,
    "y": [2, 3],
    "z": 4
  }
};

var data2 = {
  "a": {
    "x": 2,
    "y": [2, 4],
    "z": 4
  }
}

diff(data1, data2);
//{
//  "a": {
//    "x": 2,
//    "y": [
//      undefined,
//      4
//    ]
//  }
//}

```

**THEO:** What about the performance of the structural diff algorithm? It seems that the algorithm goes over the leaves of both pieces of data?

**JOE:** In the general case, that's true. But in the case of system data that's manipulated with structural sharing, the code is much more efficient.

**THEO:** What do you mean?

**JOE:** With structural sharing, most of the nested objects are shared between two versions of the system state. Therefore most of the time, when the code enters `diffObjects`, it will immediately return because `data1` and `data2` are the same.

**TIP**

**Calculating the diff between two versions of the state is efficient because two hash maps created via structural sharing from the same hash map have most of their nodes in common.**

**THEO:** Another benefit of immutable data. Let me see how the diff algorithm behaves with concurrent mutations. I think I'll start with a library with no users and a catalog with a single book "Watchmen" (see [5.6](#)).

### Listing 5.6 The data for a tiny library

```

var library = {
  "catalog": {
    "booksByIsbn": {
      "978-1779501127": {
        "isbn": "978-1779501127",
        "title": "Watchmen",
        "publicationYear": 1987,
        "authorIds": ["alan-moore", "dave-gibbons"]
      }
    },
    "authorsById": {
      "alan-moore": {
        "name": "Alan Moore",
        "bookIsbns": ["978-1779501127"]
      },
      "dave-gibbons": {
        "name": "Dave Gibbons",
        "bookIsbns": ["978-1779501127"]
      }
    }
  };
};

```

**JOE:** I suggest that we start with non-conflicting mutations. What do you suggest?

**THEO:** A mutation that updates the publication year of "Watchmen" and a mutation that updates both the title of "Watchmen" and the name of the author of "Watchmen" (see [5.7](#)).

### Listing 5.7 Two non-conflicting mutations: one mutation updates the publication year of "Watchmen" and the other one updates the title of "Watchmen" and the name of the author of "Watchmen"

```

var previous = library;
var next = _.set(
  library,
  ["catalog", "booksByIsbn", "978-1779501127", "publicationYear"],
  1986);
var libraryWithUpdatedTitle = _.set(
  library,
  ["catalog", "booksByIsbn", "978-1779501127", "title"],
  "The Watchmen");
var current = _.set(
  libraryWithUpdatedTitle,
  ["catalog", "authorsById", "dave-gibbons", "name"],
  "David Chester Gibbons");

```

**THEO:** I'm curious to see what the diff between `previous` and `current` looks like.

**JOE:** Run the code and you'll see.

Theo runs the code snippets in [5.8](#) and [5.9](#).

### Listing 5.8 The structural diff between `previous` and `next` is a nested hash map with a single field

```
diff(previous, next);
//{
//  "catalog": {
//    "booksByIsbn": {
//      "978-1779501127": {
//        "publicationYear": 1986
//      }
//    }
//  }
//}
```

### Listing 5.9 The structural diff between `previous` and `current` is a nested hash map with two fields

```
diff(previous, current);
//{
//  "authorsById": {
//    "dave-gibbons": {
//      "name": "David Chester Gibbons",
//    }
//  },
//  "catalog": {
//    "booksByIsbn": {
//      "978-1779501127": {
//        "title": "The Watchmen"
//      }
//    }
//  }
//}
```

**JOE:** Could you give me the information path of the single field in the structural diff between `previous` and `next`?

**THEO:** It's `[ "catalog", "booksByIsbn", "978-1779501127", "publicationYear" ]`.

**JOE:** Right. And what are the information paths of the fields in the structural diff between `previous` and `current`?

**THEO:** It's `[ "catalog", "booksByIsbn", "978-1779501127", "title" ]` for the book title and `[ "authorsById", "dave-gibbons", "name" ]` for the author name.

**JOE:** Perfect!

**JOE:** Can you figure out how to detect conflicting mutations by inspecting the information paths of the structural diffs?

**THEO:** We need to check if they have an information path in common or not.

**JOE:** Exactly! If they have, it means the mutation are conflicting.

**THEO:** But I have no idea how to write code that retrieves the information paths of a nested map.

**JOE:** Once again, it's a non-trivial piece of code that involves a recursion inside a `reduce` (see [5.10](#)).

### Listing 5.10 Calculating the information paths of a (nested) map

```
function informationPaths (obj, path = []) {
  return _.reduce(obj,
    function(acc, v, k) {
      if (_.isObject(v)) {
        return _.concat(acc,
          informationPaths(v,
            _.concat(path, k)));
      }
      return _.concat(acc, [_.concat(path, k)]); ❶
    },
    []);
}
```

**THEO:** Let me see if your code works as expected with the structural diffs of the mutations (see [5.11](#) and [5.12](#)).

### Listing 5.11 The information paths of the structural diff between previous and next

```
informationPaths(diff(previous, next));
//["catalog.booksByIsbn.978-1779501127.publicationYear"]
```

### Listing 5.12 The information paths of the structural diff between previous and current

```
informationPaths(diff(previous, current));
//[[
//  [
//    "catalog",
//    "booksByIsbn",
//    "978-1779501127",
//    "title"
//  ],
//  [
//    "authorsById",
//    "dave-gibbons",
//    "name"
//  ]
//]
```

**THEO:** Nice! I assume that Lodash has a function that checks whether two arrays have an element in common.

**JOE:** Almost. There is `_.intersection` that returns an array of the unique values that are in two given arrays. For our purpose, we need to check whether the intersection is empty (see [5.13](#)).

### Listing 5.13 Checking whether two diff maps have a common information path

```
function havePathInCommon(diff1, diff2) {
  return !_.isEmpty(_.intersection(informationPaths(diff1),
                                    informationPaths(diff2)));
}
```

**THEO:** You told me earlier that in the case of non-conflicting mutations, we can safely patch the changes induced by the transition from `previous` to `next` into `current`. How do you implement it?

**JOE:** We do a *recursive* merge between `current` and the diff between `previous` and `next`.

**THEO:** Does Lodash provide an immutable version of recursive merge?

**JOE:** Yes.

**THEO:** Could it be as simple as the code in [5.14](#)?

**JOE:** Indeed.

### Listing 5.14 Applying a patch

```
_.merge(current, (diff(previous, next)));
//{
// "authorsById": {
//   "dave-gibbons": {
//     "name": "David Chester Gibbons"
//   }
// },
// "catalog": {
//   "authorsById": {
//     "alan-moore": {
//       "bookIsbn": ["978-1779501127"]
//       "name": "Alan Moore"
//     },
//     "dave-gibbons": {
//       "bookIsbn": ["978-1779501127"],
//       "name": "Dave Gibbons"
//     }
//   },
//   "booksByIsbn": {
//     "978-1779501127": {
//       "authorIds": ["alan-moore", "dave-gibbons"],
//       "isbn": "978-1779501127",
//       "publicationYear": 1986,
//       "title": "The Watchmen"
//     }
//   }
// }
```

## 5.6 The implementation of the reconciliation algorithm

**JOE:** All the pieces are in place to implement our reconciliation algorithm.

**THEO:** What kind of changes are required?

**JOE:** It only requires changes in the code of `SystemData.commit`. (see [5.15](#))

### Listing 5.15 The `SystemData` class

```
class SystemState {
    systemData;

    get() {
        return this.systemData;
    }

    set(_systemData) {
        this.systemData = _systemData;
    }

    commit(previous, next) {
        var nextSystemData = SystemConsistency.reconcile(this.systemData, ①
            previous,
            next);
        if(!SystemValidity.validate(previous, nextSystemData)) {
            throw "The system data to be committed is not valid!";
        };
        this.systemData = nextSystemData;
    }
}
```

- ① `SystemConsistency` class is implemented below

**THEO:** And how does `SystemConsistency` do the reconciliation?

**JOE:** The `SystemConsistency` class (see [5.16](#)) starts the reconciliation process by comparing `previous` and `current`. If they are the same, then we fast-forward and return `next`.

### Listing 5.16 The reconciliation flow in action

```
class SystemConsistency {
    static threeWayMerge(current, previous, next) {
        var previousToCurrent = diff(previous, current); // <1>
        var previousToNext = diff(previous, next);
        if(havePathInCommon(previousToCurrent, previousToNext)) {
            return _.merge(current, previousToNext);
        }
        throw "Conflicting concurrent mutations.";
    }
    static reconcile(current, previous, next) {
        if(current == previous) {
            return next;
        }
        return SystemConsistency.threeWayMerge(current,
            previous,
            next);
    }
}
```

- ① When the system state is the same as the state used by the calculation phase, we fast-forward.

**THEO:** Wait a minute! Why do you compare `previous` and `current` by reference? You should be comparing them by value - and it would be quite expensive to compare all the leaves of the two nested hash maps!

**JOE:** That's another benefit of immutable data: when the data is not mutated, it is safe to compare references. If they are the same, we know for sure that the data is the same.

**TIP**

**When data is immutable, it is safe to compare by reference, which is super fast: when the references are the same, it means that the data is the same.**

**THEO:** What about the implementation of the 3-way merge algorithm?

**JOE:** When `previous` differs from `current`, it means that concurrent mutations have run. In order to determine whether there is a conflict or not, we calculate two diffs: the diff between `previous` and `current` and the diff between `previous` and `next`. If the intersection between the two diffs is empty, it means there is no conflict. We can safely patch the changes between `previous` to `next` into `current`.

Theo takes a closer look at the code for `SystemConsistency` class in [5.16](#) and he is trying to figure out if the code is thread-safe or not.

**THEO:** I think the code for `SystemConsistency` class (see [5.16](#)) is not thread-safe! If there's a context switch between checking whether the system has changed in the `SystemConsistency` class and the updating of the state in `SystemData` class, a mutation might override the changes of a previous mutation.

**JOE:** You are totally right! The code works fine in a single-threaded environment like JavaScript where concurrency is handled via an event loop. However in a multi-threaded environment, the code needs to be refined in order to be thread-safe. I'll show you some day.

**WARNING**

**The `SystemConsistency` class is not thread-safe. We will make it thread-safe in Chapter 8.**

**THEO:** I think I understand why you called it *optimistic* concurrency control. It's because we assume that conflicts don't occur too often. Right?

**JOE:** Correct. It makes me wondering what would your therapist say about conflicts that cannot be resolved. Are there some cases where it's not possible to reconcile the couple?

**THEO:** I don't think she ever mentioned such a possibility.

**JOE:** She must be a very optimistic person.

## 5.7 Summary

- Optimistic concurrency control allows mutations to ask forgiveness instead of permission.
- Optimistic concurrency control is lock-free.
- Managing concurrent mutations of our system state with optimistic concurrency control allows our system to support a high throughput of reads and writes.
- Optimistic concurrency control with immutable data is super efficient.
- Before updating the state, we need to reconcile the conflicts between possible concurrent mutations.
- We reconcile between concurrent mutations in a way that is similar to how git handles a merge between two branches: either a fast-forward or a 3-way merge.
- The changes required to let our system manage concurrency are only in the commit phase.
- The calculation phase does its calculation as if it were the only mutation running.
- The commit phase is responsible for trying to reconcile concurrent mutations.
- The reconciliation algorithm is universal in the sense that it can be used in any system where the system data is represented as an immutable hash map.
- The implementation of the reconciliation algorithm is efficient as it leverages the fact that subsequent versions of the system state are created via structural sharing.
- In a user facing system, conflicting concurrent mutations are fairly rare.
- When we cannot safely reconcile between concurrent mutations, we abort the mutation and ask the user to try again.
- Calculating the structural diff between two versions of the state is efficient because two hash maps created via structural sharing from the same hash map have most of their nodes in common.
- When data is immutable, it is safe to compare by reference, which is super fast: when the references are the same, it means that the data is the same.
- There are three kinds of structural differences between two nested hash maps: replacement, addition and deletion.
- Our structural diff algorithm supports replacements and additions but not deletions.

**Table 5.6 The two phases of a mutation with Optimistic Concurrency Control**

Phase	Responsibility	State	Implementation
Calculation	Compute next state in isolation	Stateless	Specific
Commit	Reconcile and update system state	Stateful	Common

**Table 5.7** The analogy between git and Data-Oriented Programming

Data-Oriented Programming	git
concurrent mutations	different branches
a version of the system data	a commit
state	a reference
calculation phase	branching
validation	pre-commit hook
reconciliation	merge
fast-forward	fast-forward
3-way merge	3-way merge
abort	manual conflict resolution
hash map	tree (folder)
leaf node	blob (file)
data field	line of code

**Table 5.8** Lodash functions introduced in this chapter

Function	Description
concat(arrA, arrB)	Creates a new array concatenating arrA and arrB
intersection(arrA, arrB)	Creates an array of unique values both in arrA and arrB
union(arrA, arrB)	Creates an array of unique values from arrA and arrB
find(coll, pred)	Iterates over elements of coll, returning the first element for which pred returns true
isEmpty(coll)	Checks if coll is empty
reduce(coll, f, initVal)	Reduces coll to a value which is the accumulated result of running each element in coll through f, where each successive invocation is supplied the return value of the previous
isArray(coll)	Checks if coll is an array
isObject(coll)	Checks if coll is a collection

# Unit tests

This chapter covers:

- Generation of the minimal data input for a test case.
- Comparison of the output of a function with the expected output.
- Guidance about the quality and the quantity of the test cases.

## 6.1 Programming at a Coffee shop

In a Data-Oriented system, our code deals mainly with data manipulation: most of our functions receive data and return data. As a consequence, it's quite easy to write unit tests to check whether our code behaves as expected. A unit test is made of test cases that generate a data input and compare the data output of the function with the expected data output.

In this chapter, we write unit tests for queries and mutations that we wrote in previous chapters.

## 6.2 The simplicity of Data-Oriented test cases

Theo and Joe are seated around a large wooden table in a corner of "La vie est belle", a nice little French coffee shop, located near the Golden Gate Bridge. Theo orders a café au lait with a croissant and Joe orders a tight espresso with a pain au chocolat.

Instead of the usual general discussions about programming and life when they're out of the office, Joe leads the discussion towards a very concrete topic: Unit tests. Theo asks Joe for an explanation.

**THEO:** Are unit tests such a simple topic that we can tackle it here in a coffee shop?

**JOE:** Unit tests in general, no. But unit tests for Data-Oriented code, yes.

**THEO:** Why does that make a difference?

**JOE:** The vast majority of the code base of a Data-Oriented system deals with data manipulation.

**THEO:** Yeah. I noticed that almost all the functions we wrote so far receive data and return data.

**TIP**

**Most of the code in a Data-Oriented system deals with data manipulation.**

**JOE:** Writing a test case for functions that deal with data is only about generating data input and expected output, and comparing the output of the function with the expected output.

**SIDE BAR****The steps of a test case**

1. **Generate data input:** `dataIn`
2. **Generate expected output:** `dataOut`
3. **Compare the output of the function with the expected output:** `f(dataIn)` **and** `dataOut`

**THEO:** That's it?

**JOE:** Yes. As you'll see in a moment, in DOP there's usually no need for mock functions.

**THEO:** I understand how to compare primitive values like strings or numbers. But I'm not sure how I would compare data collections like maps.

**JOE:** You compare field by field!

**THEO:** Recursively?

**JOE:** Yes!

**THEO:** Oh no! I'm not able to write any recursive code in a coffee shop. I need the calm of my office for that kind of stuff.

**JOE:** Don't worry! In DOP, data is represented in a generic way. There is a generic function in Lodash called `_.isEqual` for recursive comparison of data collections. It works with both maps and arrays.

Joe is able to convince Theo by executing code snippets as in [6.1](#) and [6.2](#).

### Listing 6.1 Comparing equal data collection recursively with `_.isEqual`

```
_.isEqual({
  "name": "Alan Moore",
  "bookIsbns": ["978-1779501127"]
}, {
  "name": "Alan Moore",
  "bookIsbns": ["978-1779501127"]
});
// true
```

### Listing 6.2 Comparing non-equal data collection recursively with `_.isEqual`

```
_.isEqual({
  "name": "Alan Moore",
  "bookIsbns": ["978-1779501127"]
}, {
  "name": "Alan Moore",
  "bookIsbns": ["bad-isbn"]
});
// false
```

**THEO:** Nice!

**JOE:** Most of the test cases in DOP follow this pattern.

Theo types a few line of pseudo-code as in [6.3](#).

### Listing 6.3 The general pattern of a Data-Oriented test case

```
var dataIn = {
  // input
};

var dataOut = {
  // expected output
};

_.isEqual(f(dataIn), dataOut);
```

**TIP**

It's straightforward to write unit tests for code that deals with data manipulation.

**THEO:** Indeed, it looks like something we can tackle in a coffee shop!

## 6.3 Unit tests for data manipulation code

A waiter in a very elegant bow tie brings Theo his croissant and Joe his pain au chocolat. The two friends momentarily interrupt their discussion to savor the food.

When they're done enjoying their french pastries, they ask the waiter to bring them their coffee. Meanwhile, they resume the discussion...

**JOE:** Do you remember the code flow of the implementation of the search query?

**THEO:** Let me quickly look again at the code that implements the search query.

Theo brings up the implementation of the search query as in [6.4](#).

#### Listing 6.4 The code involved in the implementation of the search query

```

class Catalog {
    static authorNames(catalogData, authorIds) {
        return _.map(authorIds, function(authorId) {
            return _.get(catalogData, ["authorsById", authorId, "name"]);
        });
    }

    static bookInfo(catalogData, book) {
        var bookInfo = {
            "title": _.get(book, "title"),
            "isbn": _.get(book, "isbn"),
            "authorNames": Catalog.authorNames(catalogData, _.get(book, "authorIds"))
        };
        return bookInfo;
    }

    static searchBooksByTitle(catalogData, query) {
        var allBooks = _.get(catalogData, "booksByIsbn");
        var matchingBooks = _.filter(allBooks, function(book) {
            return _.get(book, "title").includes(query);
        });
        var bookInfos = _.map(matchingBooks, function(book) {
            return Catalog.bookInfo(catalogData, book);
        });
        return bookInfos;
    }
}

class Library {
    static searchBooksByTitleJSON(libraryData, query) {
        var catalogData = _.get(libraryData, "catalog");
        var results = Catalog.searchBooksByTitle(catalogData, query);
        var resultsJSON = JSON.stringify(results);
        return resultsJSON;
    }
}

```

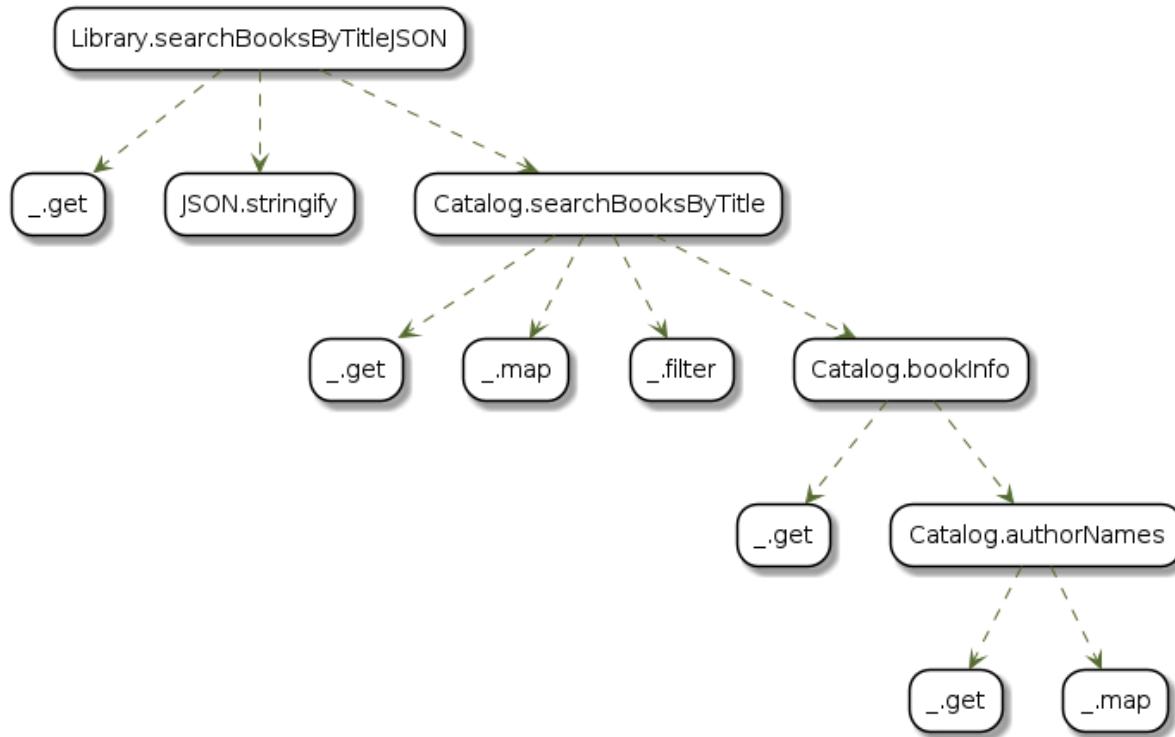
### 6.3.1 The tree of function calls

The waiter brings Theo his café au lait and Joe his tight espresso. They continue their discussion while enjoying their coffee.

**JOE:** Before writing a unit test for a code flow, I find it useful to visualize the tree of function calls of the code flow.

**THEO:** What do you mean by a *tree of function calls*?

**JOE:** Here, I'll draw the tree of function calls (see [6.1](#)) for the `Library.searchBooksByTitleJSON` code flow.



**Figure 6.1 The tree of function calls for the search query code flow**

**THEO:** Nice! Can you teach me how to draw a tree of function calls like that?

**JOE:** Sure! The root of the tree is the name of the function for which you draw the tree: in our case `Library.searchBooksByTitleJSON`. The children of a node in the tree are the names of the functions called by the function. For example, if you look again at the code for `Library.searchBooksByTitleJSON` in [6.4](#), you'll see that it calls `Catalog.searchBooksByTitle`, `_.get` and `JSON.stringify`.

**THEO:** How long would I continue recursively expanding the tree?

**JOE:** You continue until you reach a function that doesn't belong to the code base of your application: those nodes are the leaves of our tree. For example, the functions from Lodash: `_.get`, `_.map` etc...

**THEO:** What if the code of a function doesn't call any other functions?

**JOE:** A function that doesn't call any other function would be a leaf in the tree.

**THEO:** What about functions that are called inside anonymous functions, like `Catalog.bookInfo`?

**JOE:** `Catalog.bookInfo` appears in the code of `Catalog.searchBooksByTitle`. Therefore, it is considered to be a child node of `Catalog.searchBooksByTitle`. The fact that it is nested inside an anonymous function is not relevant in the context of the tree of function calls.

**NOTE**

A tree of function calls for a function  $f$  is a tree where the root is  $f$  and the children of a node  $g$  in the tree are the functions called by  $g$ . The leaves of the tree are functions that are not part of the code base of the application and functions that don't call any other functions.

**THEO:** It's very cool to visualize my code as a tree. But I don't see how it relates to unit tests.

**JOE:** The tree of function calls guides us about the quality and the quantity of test cases we should write.

**THEO:** How?

**JOE:** You'll see in a moment.

### 6.3.2 Unit test for functions down the tree

**JOE:** Let's start from the function that appears in the deepest node in our tree: `Catalog.authorName`. Take a look at the code of `Catalog.authorNames` and tell me: what are the input and the output of `Catalog.authorNames`

Theo takes a closer look at the code in [6.5](#).

#### **Listing 6.5 The code of Catalog.authorNames**

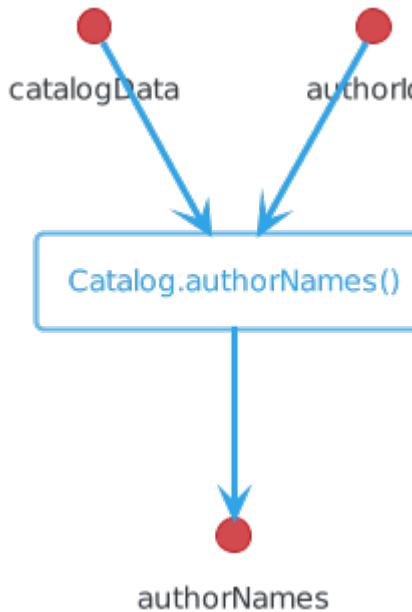
```
Catalog.authorNames = function (catalogData, authorIds) {
  return _.map(authorIds, function(authorId) {
    return _.get(catalogData, ["authorsById", authorId, "name"]);
  });
};
```

**THEO:** The input of `Catalog.authorNames` is: `catalogData` and `authorIds`. The output is: `authorNames`.

**JOE:** Would you do me a favor and express it visually?

**THEO:** Sure.

Theo grabs a napkin to draw a small rectangle (see [6.2](#)) with two inward arrows and one outward arrow.



**Figure 6.2 Visualization of the input and output of `Catalog.authorNames`**

**JOE:** Excellent! Now, how many combination of input would you include in the unit test for `Catalog.authorNames`?

**THEO:** Let me see. To begin with, I would have a `catalogData` with 2 author ids and call `Catalog.authorNames` with: 1) an empty array, 2) an array with a single author id, 3) an array with two author ids.

**Table 6.1 The table of test cases for `Catalog.authorNames`**

<code>catalogData</code>	<code>authorIds</code>	<code>authorNames</code>
catalog with two authors	empty array	empty array
catalog with two authors	array with one author id	array with author name
catalog with two authors	array with two author ids	array with two author name

**JOE:** How would you generate the `catalogData`?

**THEO:** Exactly as we generated it before (see [6.6](#)).

### Listing 6.6 A complete catalogData map

```
var catalogData = {
  "booksByIsbn": {
    "978-1779501127": {
      "isbn": "978-1779501127",
      "title": "Watchmen",
      "publicationYear": 1987,
      "authorIds": ["alan-moore", "dave-gibbons"],
      "bookItems": [
        {
          "id": "book-item-1",
          "libId": "nyc-central-lib",
          "isLent": true
        },
        {
          "id": "book-item-2",
          "libId": "nyc-central-lib",
          "isLent": false
        }
      ]
    },
    "authorsById": {
      "alan-moore": {
        "name": "Alan Moore",
        "bookIsbns": ["978-1779501127"]
      },
      "dave-gibbons": {
        "name": "Dave Gibbons",
        "bookIsbns": ["978-1779501127"]
      }
    }
};
```

**JOE:** You could use this big catalogData map for the unit test, but you could also use a smaller map in the context of Catalog.authorNames. You could get rid of the booksByIsbn field of the catalogData and the bookIsbns fields of the authors.

Joe removes a few lines from catalogData and gets a much smaller map as in [6.7](#).

### Listing 6.7 A minimal version of catalogData

```
var catalogData = {
  "authorsById": {
    "alan-moore": {
      "name": "Alan Moore"
    },
    "dave-gibbons": {
      "name": "Dave Gibbons"
    }
};
```

**THEO:** Wait a minute: This catalogData is not valid!

**JOE:** In Data-Oriented programming, data validity depends on the context. In the context of Library.searchBooksByTitleJSON and Catalog.searchBooksByTitle, the minimal version of catalogData is indeed not valid. However, in the context of Catalog.bookInfo and

`Catalog.authorNames`, it is perfectly valid. The reason is that those 2 functions access only the `authorsById` field of `catalogData`.

**TIP**

**The validity of the data depends on the context.**

**THEO:** Why is it better to use a minimal version of the data in a test case?

**JOE:** For a very simple reason: the smaller the data, the easier it is to manipulate.

**TIP**

**The smaller the data, the easier it is to manipulate.**

**THEO:** I'll appreciate that when I write the unit tests!

**JOE:** Definitely. Last thing before we start coding: how would you check that the output of `Catalog.authorNames` is as expected.

**THEO:** I would check that the value returned by `Catalog.authorNames` is an array with the expected author names.

**JOE:** How would you handle the array comparison?

**THEO:** Let me think. I want to compare by value, not by reference. I guess I'll have to check that the array is of the expected size and check member by member, recursively.

**JOE:** That's too much of a mental burden when you're in a Coffee shop. As I showed you earlier (see [6.1](#)), we can recursively compare two data collections by value with `_.isEqual` from Lodash.

**TIP**

**We compare the output and the expected output of our functions with `_.isEqual`.**

**THEO:** Sounds good! Let me write the test cases.

Theo starts typing on his laptop. After a few minutes, he has test cases (see [6.8](#)) for `Catalog.authorNames`, each made from a function call to `Catalog.authorNames` wrapped in `_.isEqual`.

### Listing 6.8 Unit test for Catalog.authorNames

```

var catalogData = {
  "authorsById": {
    "alan-moore": {
      "name": "Alan Moore"
    },
    "dave-gibbons": {
      "name": "Dave Gibbons"
    }
  }
};

_.isEqual(Catalog.authorNames(catalogData, []), []);
_.isEqual(Catalog.authorNames(catalogData, ["alan-moore"]), ["Alan Moore"]);
_.isEqual(Catalog.authorNames(catalogData, ["alan-moore", "dave-gibbons"]),
  ["Alan Moore", "Dave Gibbons"]);

```

**JOE:** Well done! Can you think of more test cases?

**THEO:** Yes. Test cases where the author id doesn't appear in the catalog data. Test cases with empty catalog data. With minimal catalog data and `_.isEqual`, it's really easy to write many test cases

Theo creates a few more test cases, as shown in [6.9](#).

### Listing 6.9 More test cases for Catalog.authorNames

```

_.isEqual(Catalog.authorNames({}, []), []);
_.isEqual(Catalog.authorNames({}, ["alan-moore"]), [undefined]);

_.isEqual(Catalog.authorNames(catalogData, ["alan-moore", "albert-einstein"]),
  ["Alan Moore", undefined]);
_.isEqual(Catalog.authorNames(catalogData, []), []);
_.isEqual(Catalog.authorNames(catalogData, ["albert-einstein"]), [undefined]);

```

**THEO:** How do I run those unit tests?

**JOE:** You use your preferred test framework!

<b>WARNING</b>	<b>We don't deal here with test runners and test frameworks. We deal only with the logic of the test cases.</b>
----------------	---

### 6.3.3 Unit test for nodes in the tree

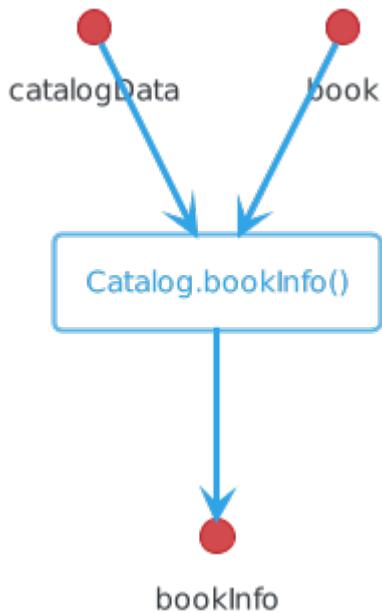
**THEO:** I'm curious to see what unit tests for an upper node in the tree of function calls look like.

**JOE:** Sure. Let's write a unit test for `Catalog.bookInfo`. How many test cases would you have for `Catalog.bookInfo`?

Theo takes another look at the code of `Catalog.bookInfo` in [6.10](#) and he draws quickly a visualization of its input and output, as in [6.3](#).

### Listing 6.10 The code of Catalog.bookInfo

```
Catalog.bookInfo = function (catalogData, book) {
  return {
    "title": _.get(book, "title"),
    "isbn": _.get(book, "isbn"),
    "authorNames": Catalog.authorNames(catalogData, _.get(book, "authorIds"))
  };
};
```



**Figure 6.3 Visualization of the input and output of Catalog.bookInfo**

**THEO:** I would have a similar number test cases for Catalog.authorNames: a book with a single author, with two authors, existing authors, non-existent authors...

**JOE:** It's not necessary. Given that we have already written unit tests for Catalog.authorNames, we don't need to check all the cases again. We simply need to write a minimal test case to confirm that the code works.

**TIP**

When we write a unit test for a function, we assume that the functions called by this function are covered by unit tests and work as expected. It significantly reduces the quantity of test cases in our unit tests.

**THEO:** Makes sense.

**JOE:** How would you write a minimal test case for Catalog.bookInfo?

Theo takes a look at the code of Catalog.bookInfo in [6.10](#) and answers Joe's question.

**THEO:** I would use the same catalog data as for `Catalog.authorNames` and a book record. I'd test that the function behaves as expected by comparing its return value with a book info record using `_.isEqual`.

This time, it takes Theo a bit more time to write the unit test. The reason is that the input and the output of `Catalog.authorNames` are both records. Dealing with a record is more complex than dealing with an array of strings (as it was the case for `Catalog.authorNames`). Theo appreciates the fact that `_.isEqual` saves him from writing code that compares the two maps property by property. The result of Theo's efforts appears in [6.11](#).

### Listing 6.11 Unit test for `Catalog.bookInfo`

```
var catalogData = {
  "authorsById": {
    "alan-moore": {
      "name": "Alan Moore"
    },
    "dave-gibbons": {
      "name": "Dave Gibbons"
    }
  }
};

var book = {
  "isbn": "978-1779501127",
  "title": "Watchmen",
  "publicationYear": 1987,
  "authorIds": ["alan-moore", "dave-gibbons"]
};

var expectedResult = {
  "authorNames": ["Alan Moore", "Dave Gibbons"],
  "isbn": "978-1779501127",
  "title": "Watchmen",
};

var result = Catalog.bookInfo(catalogData, book);

_.isEqual(result, expectedResult);
```

**JOE:** Perfect! How would you compare the kind of unit tests for `Catalog.bookInfo` with the unit tests for `Catalog.authorNames`?

**THEO:** On one hand, there is only a single test case in the unit test for `Catalog.bookInfo`. On the other hand, the data involved in the test case is more complex than the data involved in the test cases for `Catalog.authorNames`.

**JOE:** Exactly! Functions that appear in a deep node in the tree of function calls tend to require more test cases but the data involved in the test cases is less complex.

#### TIP

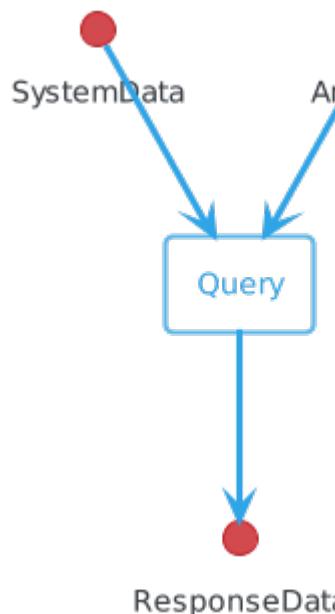
Functions that appear in a low level in the tree of function calls tend to involve less complex data than functions that appear in a higher level in the tree.

**Table 6.2 The correlation between the depth of a function in the tree of function calls and the quality and quantity of the test cases**

Depth in the tree	Complexity of the data	Number of test cases
Lower	Higher	Lower
Higher	Lower	Higher

## 6.4 Unit tests for queries

In the previous section, we have seen how to write unit tests for utility functions like `Catalog.bookInfo` and `Catalog.authorNames`. Now, we are going to see how to write unit tests for the nodes of query tree of function calls that are close to the root of the tree.



**Figure 6.4 The input and output of the main function of a query**

**JOE:** How would you write a unit test for the code of the entry point of the search query?

Theo looks again at the code of `Library.searchBooksByTitleJSON` in [6.12](#).

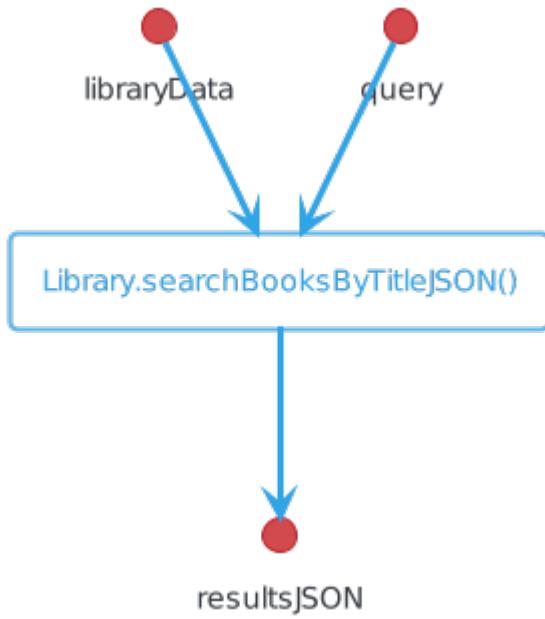
### Listing 6.12 The code of `Library.searchBooksByTitleJSON`

```

Library.searchBooksByTitleJSON = function (libraryData, query) {
  var catalogData = _.get(libraryData, "catalog");
  var results = Catalog.searchBooksByTitle(catalogData, query);
  var resultsJSON = JSON.stringify(results);
  return resultsJSON;
};
  
```

**THEO:** The input of `Library.searchBooksByTitleJSON` are library data and a query string. The output is a JSON string. (See [6.5](#).) Therefore, I would create a library data record with a single book and write tests with query strings that match the name of the book, and ones that

don't match.



**Figure 6.5 The input and output of `Library.searchBooksByTitleJSON()`**

**JOE:** What about the expected results of the test cases?

**THEO:** In cases where the query string matches, the expected result is a JSON string with the book info. In cases where the query string doesn't match, the expected result is a JSON string with an empty array.

**JOE:** Hmm.

**THEO:** What?

**JOE:** I don't like your answer.

**THEO:** Why?

**JOE:** Because your test case relies on string comparison instead of data comparison.

**THEO:** What difference does it make? After all, the strings I'm comparing come from serialization of data.

**JOE:** It's inherently much more complex to compare JSON strings than to compare data. For example, two different strings might be the serialization of the same piece of data.

**THEO:** Really? How?

**JOE:** Take a look at the two strings in [6.13](#): they are the serialization of the same data. They're

different strings because, the fields appear in a different order. But in fact, they serialize the same data!

### **Listing 6.13 Two different strings that serialize the same data**

```
var stringA = "{\"title\":\"Watchmen\", \"publicationYear\":1987}";  
var stringB = "{\"publicationYear\":1987, \"title\":\"Watchmen\"}";
```

**TIP**

**Avoid to use string comparison in unit tests for functions that deals with data.**

**THEO:** I see. So what can I do instead?

**JOE:** Instead of comparing the output of `Library.searchBooksByTitleJSON` with a string, you could deserialize the output and compare it to the expected data.

**THEO:** What do you mean by *deserialize* a string?

**JOE:** Deserializing a string `s` means to generate a piece of data whose serialization is `s`.

**THEO:** Is there a Lodash function for string deserialization?

**JOE:** Actually, there is a native JavaScript function for string deserialization: it's called `JSON.parse`.

Joe uses a code snippet as in [6.14](#) to illustrate a common usage of `JSON.parse`.

### **Listing 6.14 Example of string deserialization**

```
var myString = "{\"publicationYear\":1987, \"title\":\"Watchmen\"}";  
var myData = JSON.parse(myString);  
_.get(myData, "title");  
// "Watchmen"
```

**THEO:** Cool! Let me try writing a unit test for `Library.searchBooksByTitleJSON` using `JSON.parse`.

It doesn't take too much time to come up with a piece of code like the one in [6.15](#).

### Listing 6.15 Unit test for Library.searchBooksByTitleJSON

```

var libraryData = {
  "catalog": {
    "booksByIsbn": {
      "978-1779501127": {
        "isbn": "978-1779501127",
        "title": "Watchmen",
        "publicationYear": 1987,
        "authorIds": ["alan-moore",
                      "dave-gibbons"]
      }
    },
    "authorsById": {
      "alan-moore": {
        "name": "Alan Moore",
        "bookIsbns": ["978-1779501127"]
      },
      "dave-gibbons": {
        "name": "Dave Gibbons",
        "bookIsbns": ["978-1779501127"]
      }
    }
  }
};

var bookInfo = {
  "isbn": "978-1779501127",
  "title": "Watchmen",
  "authorNames": ["Alan Moore",
                  "Dave Gibbons"]
};

_.isEqual(JSON.parse(Library.searchBooksByTitleJSON(libraryData,
                                                    "Watchmen")),
         [bookInfo]);

_.isEqual(JSON.parse(Library.searchBooksByTitleJSON(libraryData,
                                                    "Batman")),
         []);

```

**JOE:** Well done! I think you're ready to move on to the last piece of the puzzle and write the unit test for Catalog.searchBooksByTitle

Theo looks again at the code of Catalog.searchBooksByTitle in [6.16](#).

### Listing 6.16 The code of Catalog.searchBooksByTitle

```

Catalog.searchBooksByTitle = function(catalogData, query) {
  var allBooks = _.get(catalogData, "booksByIsbn");
  var matchingBooks = _.filter(allBooks, function(book) {
    return _.get(book, "title").includes(query);
  });
  var bookInfos = _.map(matchingBooks, function(book) {
    return Catalog.bookInfo(catalogData, book);
  });
  return bookInfos;
};

```

Writing the unit test for Catalog.searchBooksByTitle (see [6.17](#)) is a more pleasant

experience for Theo than writing the unit test for `Library.searchBooksByTitleJSON` for two reasons:

1. It's not necessary to deserialize the output because the function returns data
2. It's not necessary to wrap the catalog data in a library data map

#### **Listing 6.17 Unit test for Catalog.searchBooksByTitle**

```
var catalogData = {
  "booksByIsbn": {
    "978-1779501127": {
      "isbn": "978-1779501127",
      "title": "Watchmen",
      "publicationYear": 1987,
      "authorIds": ["alan-moore",
                     "dave-gibbons"]
    }
  },
  "authorsById": {
    "alan-moore": {
      "name": "Alan Moore",
      "bookIsbns": ["978-1779501127"]
    },
    "dave-gibbons": {
      "name": "Dave Gibbons",
      "bookIsbns": ["978-1779501127"]
    }
  }
};

var bookInfo = {
  "isbn": "978-1779501127",
  "title": "Watchmen",
  "authorNames": ["Alan Moore",
                  "Dave Gibbons"]
};

_.isEqual(Catalog.searchBooksByTitle(catalogData, "Watchmen"), [bookInfo]);
_.isEqual(Catalog.searchBooksByTitle(catalogData, "Batman"), []);
```

**JOE:** That's a good start!

**THEO:** I thought I was done. What did I miss?

**JOE:** You forgot to test cases where the query string is all lower case.

**THEO:** You're right! Let me quickly add one more test case.

In less than a minute, Theo creates an additional test case as in [6.18](#).

#### **Listing 6.18 Additional test case for Catalog.searchBooksByTitle**

```
_.isEqual(Catalog.searchBooksByTitle(catalogData, "watchmen"),
          [bookInfo]);
```

What a disappointment when Theo discovers that the test case with "watchmen" in lower case fails!

**JOE:** Don't be too upset, my friend! After all, the purpose of unit tests is to find bugs in the code so that you can fix them. Can you fix the code of `CatalogData.searchBooksByTitle`?

**THEO:** Sure. All I need to do is to lower-case both the query string and the book title before comparing them.

Theo's fix appears in [6.19](#).

#### Listing 6.19 Fixed code of `Catalog.searchBooksByTitle`

```
Catalog.searchBooksByTitle = function(catalogData, query) {
  var allBooks = _.get(catalogData, "booksByIsbn");
  var queryLowerCased = query.toLowerCase(); ①
  var matchingBooks = _.filter(allBooks, function(book) {
    return _.get(book, "title")
      .toLowerCase() ②
      .includes(queryLowerCased);
  });
  var bookInfos = _.map(matchingBooks, function(book) {
    return Catalog.bookInfo(catalogData, book);
  });
  return bookInfos;
};
```

- ① convert the query to lower case
- ② convert the book title to lower case

After fixing the code of `Catalog.searchBooksByTitle`, Theo runs all the test cases again (see [6.20](#)). This time, all of them pass: what a relief!

#### Listing 6.20 Additional test case for `Catalog.searchBooksByTitle`

```
_.isEqual(Catalog.searchBooksByTitle(catalogData, "watchmen"),
          [bookInfo]);
```

**JOE:** It's such good feeling when all the test cases pass...

**THEO:** Sure is.

**JOE:** I think we've written unit tests for all the search query code, so now we're ready to write unit tests for mutations.

## 6.5 Unit tests for mutations

**JOE:** Before writing unit tests for the add member mutation, let's draw the tree of function calls for `System.addMember`.

**THEO:** I can do that.

Theo takes a look at the code for the functions involved in the add member mutation in [6.21](#): the

code is spread over 3 classes: System, Library and UserManagement.

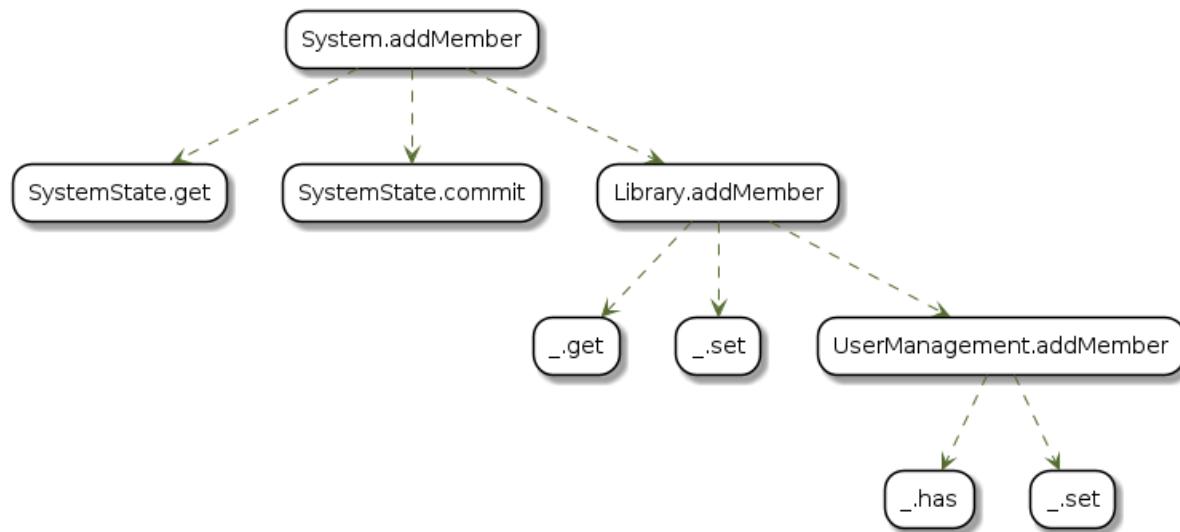
### **Listing 6.21 The functions involved in add member mutation**

```
System.addMember = function(systemState, member) {
    var previous = systemState.get();
    var next = Library.addMember(previous, member);
    systemState.commit(previous, next);
};

Library.addMember = function(library, member) {
    var currentUserManagement = _.get(library, "userManagement");
    var nextUserManagement = UserManagement.addMember(currentUserManagement, member);
    var nextLibrary = _.set(library, "userManagement", nextUserManagement);
    return nextLibrary;
};

UserManagement.addMember = function(userManagement, member) {
    var email = _.get(member, "email");
    var infoPath = ["membersByEmail", email];
    if(_.has(userManagement, infoPath)) {
        throw "Member already exists.";
    }
    var nextUserManagement = _.set(userManagement,
                                    infoPath,
                                    member);
    return nextUserManagement;
};
```

Drawing the tree of function calls for `System.addMember` is quite easy (see [6.6](#)).



**Figure 6.6 The tree of function calls for `System.addMember`**

**JOE:** Excellent! So which functions of the tree should be unit tested for the add member mutation?

**THEO:** I think the functions we need to test are: `System.addMember`, `SystemState.get`, `SystemState.commit`, `Library.addMember` and `UserManagement.addMember`.

**JOE:** You're totally right. Let's defer writing unit tests for functions that belong to

`SystemState`, until later. Those are generic functions that should be tested outside the context of a specific mutation. Let's assume for now that we've already written unit tests for the `SystemState` class. We're left with 3 functions: `System.addMember`, `Library.addMember` and `UserManagement.addMember`.

**THEO:** In what order should write the unit tests: bottom up or top down?

**JOE:** Let's start where the real meat is: in `UserManagement.addMember`. The two other functions are just wrappers.

**THEO:** OK.

**JOE:** Writing a unit test for the main function of a mutation requires more effort than for a query. The reason is that a query returns a response based on the system data, while a mutation computes a new state of the system based on the current state of the system and some arguments.

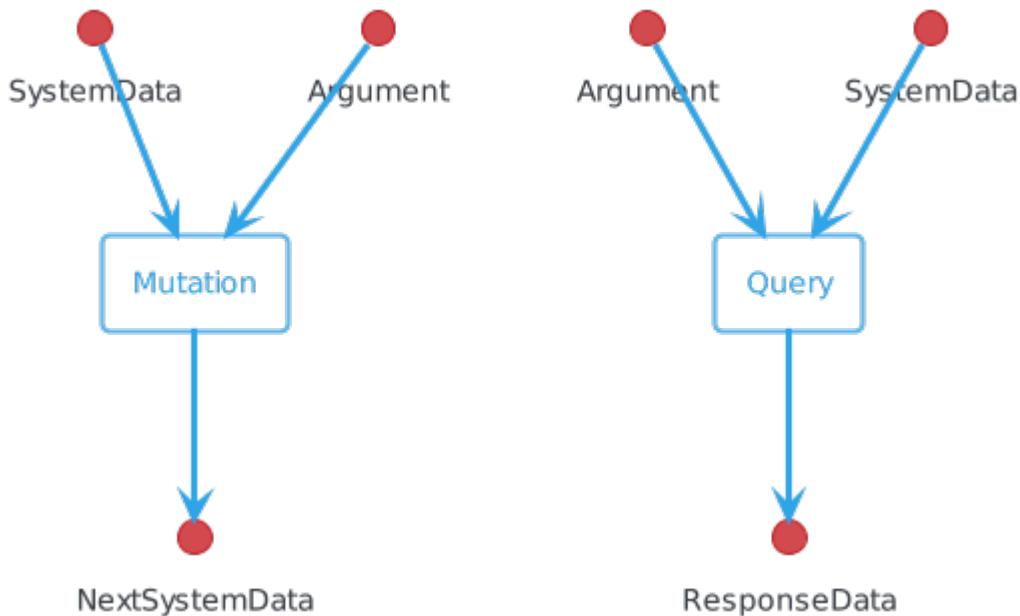


Figure 6.7 The output of a mutation is more complex than the output of a query

**TIP**

Writing a unit test for the main function of a mutation requires more effort than for a query.

**THEO:** It means that in the test cases of `UserManagement.addMember`, both the input and the expected output are maps that describe the state of the system.

**JOE:** Exactly. Let's start with the simplest case, where the initial state of the system is empty.

**THEO:** You mean that `userManagementData` passed to `UserManagement.addMember` is an empty map?

**JOE:** Yes.

Once again, Theo poises his hands over his laptop keyboard begins typing the code for adding a member to an empty user management map and checking that the resulting map is as expected. The code appears in [6.22](#).

### Listing 6.22 Test case for Catalog.addMember without members

```
var member = {
    "email": "jessie@gmail.com",
    "password": "my-secret"
};

var userManagementStateBefore = {};

var expectedUserManagementStateAfter = {
    "membersByEmail": {
        "jessie@gmail.com": {
            "email": "jessie@gmail.com",
            "password": "my-secret"
        }
    }
};

var result = UserManagement.addMember(userManagementStateBefore, member);
.assertEqual(result, expectedUserManagementStateAfter);
```

**JOE:** Very nice! Keep going and write a test case when the initial state is not empty.

It requires a few more lines of code but nothing very complicated, as shown in [6.23](#).

### Listing 6.23 Test case for Catalog.addMember with existing members

```

var jessie = {
    "email": "jessie@gmail.com",
    "password": "my-secret"
};

var franck = {
    "email": "franck@gmail.com",
    "password": "my-top-secret"
};

var userManagementStateBefore = {
    "membersByEmail": {
        "franck@gmail.com": {
            "email": "franck@gmail.com",
            "password": "my-top-secret"
        }
    }
};

var expectedUserManagementStateAfter = {
    "membersByEmail": {
        "jessie@gmail.com": {
            "email": "jessie@gmail.com",
            "password": "my-secret"
        },
        "franck@gmail.com": {
            "email": "franck@gmail.com",
            "password": "my-top-secret"
        }
    }
};

var result = UserManagement.addMember(userManagementStateBefore, jessie);
_.isEqual(result, expectedUserManagementStateAfter);

```

**JOE:** Awesome! Can you think of other test cases for UserManagement.addMember?

**THEO:** No.

**JOE:** What about cases where the mutation fails?

**THEO:** Right. I always forget to think about negative test cases. I assume it relates to the fact that I'm an optimistic person.

**TIP**

**Don't forget to include negative test cases in your unit tests.**

**JOE:** Me too. The more I meditate the more I'm able to focus on the positive side of life. Anyway, how would you write a test case where the mutation fails?

**THEO:** I would pass to UserManagement.addMember a member that already exists in userManagementStateBefore.

**JOE:** And how would you check that the code behaves as expected in case of a failure?

**THEO:** Let me see. When a member already exists, `UserManagement.addMember` throws an exception. Therefore, what I need to do in my test case, is to wrap the code in a `try/catch` block.

**JOE:** Sounds good to me.

Once again, it doesn't require too much of an effort for Theo to create a new test case as in [6.24](#).

#### Listing 6.24 Test case for `UserManagement.addMember` when it is expected to fail

```
var jessie = {
    "email": "jessie@gmail.com",
    "password": "my-secret"
};

var userManagementStateBefore = {
    "membersByEmail": {
        "jessie@gmail.com": {
            "email": "jessie@gmail.com",
            "password": "my-secret"
        }
    }
};

var expectedException = "Member already exists.";
var exceptionInMutation;

try {
    UserManagement.addMember(userManagementStateBefore, jessie);
} catch (e) {
    exceptionInMutation = e;
}

_.isEqual(exceptionInMutation, expectedException);
```

**THEO:** Now, I think I'm ready to move forward and write unit tests for `Library.addMember` and `System.addMember`

**JOE:** I agree with you. Please start with `Library.addMember`.

**THEO:** `Library.addMember` is quite similar to `UserManagement.addMember`. So I guess, I'll write similar test cases.

**JOE:** In fact, it won't be required. As I told you when we wrote unit tests for a query, when you write a unit test for a function, you can assume that the functions down the tree work as expected.

**THEO:** Right. So I'll just write the test case for existing members.

**JOE:** Go for it!

Theo starts with a "copy and paste" of the code from the `UserManagement.addMember` test case with existing members in [6.23](#). After a few modifications, the unit test for `Library.addMember` is ready (see [6.25](#)).

### Listing 6.25 Unit test for Library.addMember

```

var jessie = {
    "email": "jessie@gmail.com",
    "password": "my-secret"
};

var franck = {
    "email": "franck@gmail.com",
    "password": "my-top-secret"
};

var libraryStateBefore = {
    "userManagement": {
        "membersByEmail": {
            "franck@gmail.com": {
                "email": "franck@gmail.com",
                "password": "my-top-secret"
            }
        }
    }
};

var expectedLibraryStateAfter = {
    "userManagement": {
        "membersByEmail": {
            "jessie@gmail.com": {
                "email": "jessie@gmail.com",
                "password": "my-secret"
            },
            "franck@gmail.com": {
                "email": "franck@gmail.com",
                "password": "my-top-secret"
            }
        }
    }
};

var result = Library.addMember(libraryStateBefore, jessie);
_.isEqual(result, expectedLibraryStateAfter);

```

**JOE:** Beautiful! Now, We're ready for the last piece: write a unit test for `System.addMember`. Could you please describe the input and the output of `System.addMember`?

Theo takes another look at the code for `System.addMember` in [6.26](#) and hesitates, confused. The function doesn't seem to return anything.

### Listing 6.26 The code of system.addMember

```

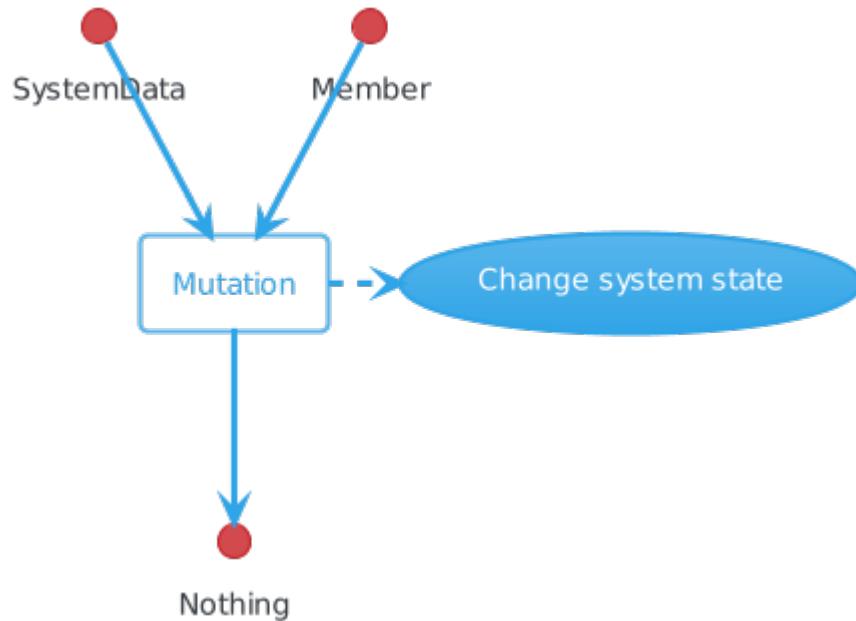
System.addMember = function(systemState, member) {
    var previous = systemState.get();
    var next = Library.addMember(previous, member);
    systemState.commit(previous, next);
};

```

**THEO:** The input of `System.addMember` is a system state instance and a member. But I'm not sure what the output of `System.addMember` is.

**JOE:** In fact, `System.addMember` doesn't have any output. It belongs to this stateful part of our

code that doesn't deal with data manipulation. DOP allows us to reduce the size of the stateful part of our code. But it still exists. Here is how I visualize it.



**Figure 6.8** `System.addMember` doesn't return data: it changes the system state

**THEO:** How do we validate that the code works as expected then?

**JOE:** We'll retrieve the system state after the code is executed and compare it with the expected value of the state.

**THEO:** Ok, I'll try to write the unit test.

**JOE:** Writing unit tests for stateful code is more complicated than for data manipulation code. It requires the calm of the office.

**THEO:** Ok, then let's go back to the office first.

When they're back at the office, Theo starts coding the unit test for `Library.addMember`.

**THEO:** Can we use `_.isEqual` with system state?

**JOE:** Definitely. The system state is a map like any other map.

**TIP**

The system state is a map. Therefore, in the context of a test case, we can compare the system state after a mutation is executed to the expected system state using `_.isEqual`

Theo copies and pastes the code for `Library.addMember` that initializes the data for the test.

Then, he passes a `SystemState` object that is initialized with `libraryStateBefore` to `System.addMember`. Finally, he compares the system state after the mutation is executed with the expected value of the state as in [6.27](#).

### **Listing 6.27 Unit test for `System.addMember`**

```
var jessie = {
    "email": "jessie@gmail.com",
    "password": "my-secret"
};

var franck = {
    "email": "franck@gmail.com",
    "password": "my-top-secret"
};

var libraryStateBefore = {
    "userManagement": {
        "membersByEmail": {
            "franck@gmail.com": {
                "email": "franck@gmail.com",
                "password": "my-top-secret"
            }
        }
    }
};

var expectedLibraryStateAfter = {
    "userManagement": {
        "membersByEmail": {
            "jessie@gmail.com": {
                "email": "jessie@gmail.com",
                "password": "my-secret"
            },
            "franck@gmail.com": {
                "email": "franck@gmail.com",
                "password": "my-top-secret"
            }
        }
    }
};

var systemState = new SystemState(); ①
systemState.commit(null, libraryStateBefore); ②
System.addMember(systemState, jessie); ③

_.isEqual(systemState.get(), expectedLibraryStateAfter); ④
```

- ① Creating an empty `SystemState` object (see Chapter 4)
- ② Initializing the system state with the library data before the mutation
- ③ Executing the mutation on the `SystemState` object
- ④ Validating the state after the mutation is executed

**JOE:** Wow, I'm impressed, you did it! Congratulations!

**THEO:** Thank you. I'm so glad that in DOP most of our code deals with data manipulation. It's definitely more pleasant to write unit tests for stateless code that only deals with data

manipulation.

**JOE:** Now that you know the basics of Data-Oriented Programming, would you like to refactor the code of your Klfafim prototype according to DOP principles?

**THEO:** Definitely. Nancy told me yesterday that Klfafim is getting nice market traction. I'm supposed to have a meeting with her in a week or so about the next steps. Hopefully, she'll be willing to work with Albatross for the long term.

**JOE:** Exciting! Do you know What might influence Nancy's decision?

**THEO:** Our cost estimate, certainly. I know she's in touch with other software companies. If we come up with a competitive proposal, I think we'll get the deal.

**JOE:** I'm quite sure that after refactoring to DOP, features will take much less time to implement. That means you should be able to quote Nancy a lower total cost than the competition, right?

**THEO:** I'll keep my fingers crossed!

## 6.6 Moving forward

The meeting with Nancy went well. You got the deal and it's going to be a long-term project with a nice budget. You'll need to hire a team of developers in order to meet the tough deadlines. While driving back to the office, you get a phone call from Joe.

**JOE:** How was your meeting with Nancy?

**THEO:** We got the deal!

**JOE:** Awesome! I told you that with DOP the cost estimation will be lower.

**THEO:** In fact, we are not going to use DOP for this project.

**JOE:** Why?

**THEO:** After refactoring the library management system prototype to DOP, I have done a deep analysis with my engineers and we have come to the conclusion that DOP might be a good fit for the prototype phase, but it won't work well at scale.

**JOE:** Could you share the details of your analysis?

**THEO:** I can't right now while I'm driving.

**JOE:** Could we meet in your office later today?

**THEO:** I'm quite busy with the new project and the tough deadlines.

**JOE:** Let's meet at least in order to have a proper farewell.

**THEO:** OK. Let's meet at 4PM then.

**WARNING** The story continues in the opener of Part 2.

## 6.7 Summary

- Most of the code in a Data-Oriented system deals with data manipulation.
- It's straightforward to write unit tests for code that deals with data manipulation.
- Test cases follow the same simple general pattern: 1. Generate data input 2. Generate expected data output 3. Compare the output of the function with the expected data output
- In order to compare the output of a function with the expected data output, we need to recursively compare the two pieces of data
- The recursive comparison of two pieces of data is implemented via a generic function
- When a function returns a JSON string, we parse the string back to data so that we deal with data comparison instead of string comparison
- A tree of function calls for a function `f` is a tree where the root is `f` and the children of a node `g` in the tree are the functions called by `g`.
- The leaves of the tree are functions that are not part of the code base of the application and functions that don't call any other functions.
- The tree of function calls visualization guides us regarding the quality and quantity of the test cases in a unit test.
- Functions that appear in a low level in the tree of function calls tend to involve less complex data than functions that appear in a higher level in the tree.
- Functions that appear in a low level in the tree of function calls usually need to be covered with more test cases than function that appear in a higher level in the tree.
- Unit tests for mutations focus on the calculation phase of the mutation.
- The validity of the data depends on the context.
- The smaller the data, the easier it is to manipulate.
- We compare the output and the expected output of our functions with a generic function that recursively compares two pieces of data (e.g. `_.isEqual`).
- When we write a unit test for a function, we assume that the functions called by this function are covered by unit tests and work as expected. It significantly reduces the quantity of test cases in our unit tests.
- We avoid using string comparison in unit tests for functions that deals with data.
- Writing a unit test for the main function of a mutation requires more effort than for a query.
- Don't forget to include negative test cases in your unit tests.
- The system state is a map. Therefore, in the context of a test case, we can compare the system state after a mutation is executed to the expected system state using a generic function like `_.isEqual`

**Table 6.3 The correlation between the depth of a function in the tree of function calls and the quality and quantity of the test cases**

Depth in the tree	Complexity of the data	Number of test cases
Lower	Higher	Lower
Higher	Lower	Higher

**Table 6.4 Lodash functions introduced in this chapter**

Function	Description
<code>isEqual(collA, collB)</code>	Performs a deep comparison between <code>collA</code> and <code>collB</code>

# Part 2 Scalability

Farewell?

Theo feels a bit uncomfortable about the meeting with Joe. He was so enthusiastic about DOP. And he was very good at teaching it. Every meeting with him was an opportunity to learn new things. Theo feels lot of gratitude for the time Joe spent with him. He doesn't want to hurt him in any fashion.

Surprisingly, Joe enters the office with the same relaxed attitude as usual and he is even smiling.

**JOE:** I'm really glad that you got the deal with Nancy.

**THEO:** Yeah. There is lot of excitement about it here in the office. And a bit of stress also.

**JOE:** What kind of stress?

**THEO:** You know. We need to hire a team of developers. And the deadlines are quite tight.

**JOE:** But you told me that you won't use DOP. I assume that you gave regular deadlines.

**THEO:** No. Monica, my boss, really wanted to close the deal. She feels that success with this project is strategically important for Albatross, and so it is worthwhile to accept some risk by giving what she calls an *optimistic* time estimation. I told her that it was really an *unrealistic* time estimation, but Monica insists that if we make smart decisions and bring in more developers we can do it.

**JOE:** I see. Now, I understand why you told me over the phone that you were very busy. Anyway, would you please share the reasons that made you think DOP would not be a good fit at scale?

**THEO:** First of all, let me tell you that I feel lot of gratitude for all the teaching you shared with me. Reimplementing the Klafim prototype with DOP was really fun and productive due to the flexibility this paradigm offers.

**JOE:** I'm happy that you found it valuable.

**THEO:** But as I told you over the phone, now we're scaling up into a long-term project with several developers working on a large code base. And we came to the conclusion that DOP will not be a good fit at scale.

**JOE:** Could you share the reasons behind your conclusion?

**THEO:** There are many of them. First of all, as DOP deals only with generic data structures, it's hard to know what kind of data we have in hand. While in OOP, we know the type of every piece of data. For the prototype, it was kind of OK. But as the code base grows and more developers are involved in the project, it would be too painful.

**JOE:** I hear you. What else my friend?

**THEO:** Our system is going to run on a multi-threaded environment. I reviewed the concurrency control strategy that you presented and it's not thread-safe.

**JOE:** I hear you. What else my friend?

**THEO:** I have been doing a bit of research about implementing immutable data structures with structural sharing and I discovered that when the size of the data structures grows, there is a significant performance hit.

**JOE:** I hear you. What else my friend?

**THEO:** As our system grows we will use a database to store the application data and external services to enrich book information, while in what you have showed me so far data lives in memory.

**JOE:** I hear you. What else my friend?

**THEO:** Don't you think I have shared enough reasons to abandon DOP?

**JOE:** I think that your concerns about DOP at scale totally make sense. But it doesn't mean that you should abandon DOP.

**THEO:** What do you mean?

**JOE:** With the help of meditation, I learned not be attached to the objections that flow in my mind while I'm practicing. Sometimes all that is needed to quiet our minds is to keep breathing; sometimes a deeper level of practice is needed.

**THEO:** I don't see how breathing would convince be to give DOP a second chance!

**JOE:** Breathing might not be enough in this case, but a deeper knowledge of DOP could be helpful. Until now, I have shared with you only the material that was needed in order to re-factor your prototype. In order to use DOP in a big project, a few more lessons are necessary.

**THEO:** But I don't have time for more lessons. I need to work.

**JOE:** Have you heard the story about the young woodcutter and the old man?

**THEO:** No.

**JOE:** It goes like this...

**SIDE BAR** A young woodcutter strained to saw down a tree. An old man who was watching asked "What are you doing?"

"Are you blind?" the woodcutter replied. "I'm cutting down this tree."

The old man replied: "You look exhausted! Take a break. Sharpen your saw."

The young woodcutter explained to the old man that he had been sawing for hours and did not have time to take a break.

The old man pushed back: "If you sharpen the saw, you would cut down the tree much faster."

The woodcutter said "I don't have time to sharpen the saw. Don't you see I'm too busy?"

**THEO:** Do you really think that with DOP it will take much less time to deliver the project?

**JOE:** I think so.

**THEO:** But if we miss the deadline, I will probably get fired. I'm the one that needs to take the risk. Not you.

**JOE:** Let's make a deal. If you miss the deadline and get fired, I will hire you at my company for double the salary you make at Albatross.

**THEO:** And if we meet the deadline?

**JOE:** If you meet the deadline, you will probably get promoted. In that case, I will ask you to buy a gift for my son Neriah and my daughter Aurelia.

**THEO:** Deal! When will I get my first lesson about deeper DOP?

**JOE:** Why not start right now?

**THEO:** Let me reschedule my meetings.

# *Basic data validation*

## This chapter covers

- The importance of validating data at system boundaries
- Validating data using JSON schema language
- Integrating data validation into an existing code base
- Getting detailed information about data validation failures

## 7.1 A solemn gift

At first sight, it may seem that embracing Data-Oriented programming means accessing data without validating it, and engaging in wishful thinking that data is always valid.

In fact, data validation is not only possible but recommended when we follow Data-Oriented principles. This chapter illustrates how to validate data when data is represented with generic data structures. This chapter focuses on data validation occurring at the boundaries of the system while in Part 3, we will deal with validating data as it flows through the system.

This chapter is a deep dive into the fourth principle of Data-Oriented Programming:

**NOTE**

**Principle #4: Separate data schema and data representation.**

## 7.2 Data validation in DOP

Theo has rescheduled his meetings, still not sure if he's made a big mistake giving DOP a second chance.

**WARNING** The reason why Theo rescheduled his meetings is explained in the opener for Part 2. Please take a moment to read the opener if you missed it.

**JOE:** What aspect of Object-Oriented programming do you think you will miss the most in your big project?

**THEO:** Data validation.

**JOE:** Can you elaborate a bit?

**THEO:** In OOP, I had this strong guarantee that when a class is instantiated, its member fields have the proper names and proper types. While with DOP, it's so easy to have small mistakes in field names and field types.

**JOE:** Well, I have good news for you. There is a way to validate data in DOP.

**THEO:** How does it work? I thought Data-Oriented programming and data validation were two contradictory concepts!

**JOE:** Not at all. It's true that DOP doesn't force you to validate data but it doesn't prevent you from doing so. In DOP, data schema is separate from data representation.

**THEO:** I don't get how that would eliminate data consistency issues.

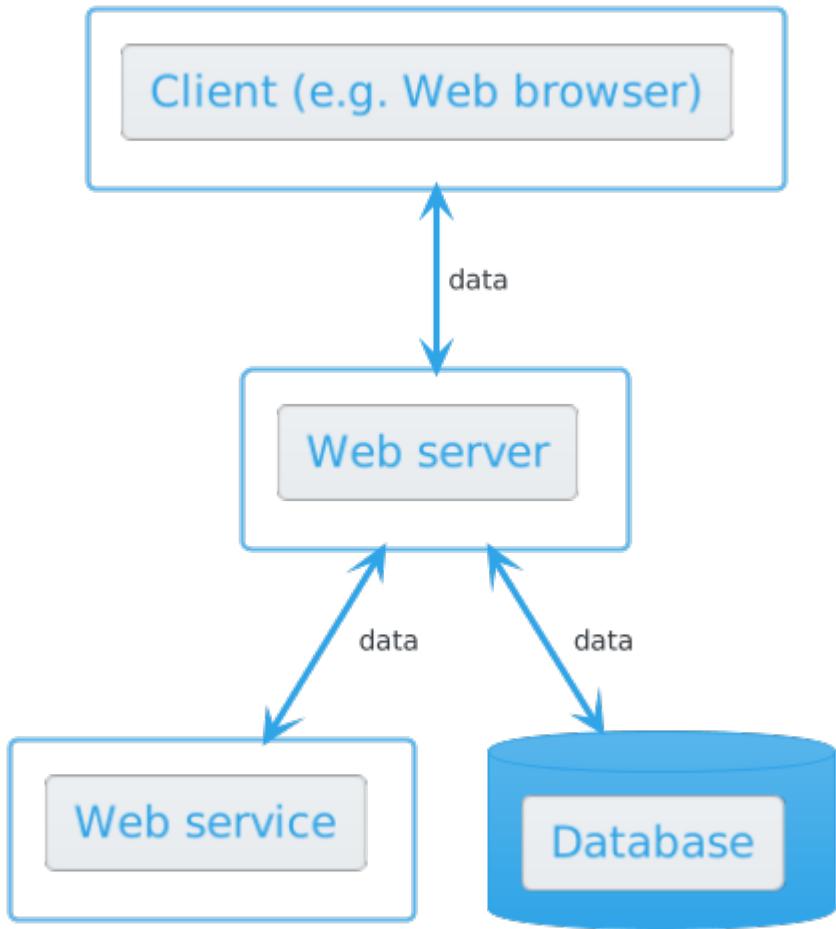
**JOE:** According to DOP, the most important data to validate is data that crosses the boundaries of the system.

**THEO:** Which boundaries are you referring to?

**JOE:** In the case of a Web server, it would be the areas where the Web server communicates with its clients and with its data sources.

**THEO:** A diagram might help me see it better.

Joe comes to the white board and draws a diagram like the one in [7.1](#).



**Figure 7.1 High level architecture of a modern web server**

**JOE:** This architecture diagram defines what we call the boundaries of the system, in terms of data exchange. Can you tell me what the 3 boundaries of the system are?

<b>NOTE</b>	The boundaries of a system are defined to be the areas where the system exchanges data.
-------------	---

**THEO:** Let me see. The first one is the client boundary. Then we have the database boundary, and finally the web service boundary.

**JOE:** Exactly! It's important to identify the boundaries of a system because in DOP, we differentiate between two kinds of data validation. Validation that occurs at the boundaries of the system and validation that occurs inside the system. Today, we're going to focus on validation that occurs at the boundaries of the system.

**THEO:** Does that mean data validation at the boundaries of the system is more important?

**JOE:** Absolutely! Once you've ensured that data going into and out of the system is valid, the

odds for an unexpected piece of data *inside* the system are pretty low.

**TIP**

**When data at system boundaries has been validated, it's not critical to validate data again inside the system.**

**THEO:** Why do we need data validation inside the system then?

**JOE:** It has to do with making it easier to code your system as your code base grows.

**THEO:** And what's the main purpose of data validation at the boundaries?

**JOE:** To prevent invalid data from going in and out of the system, and to display informative errors when we encounter invalid data.

**Table 7.1 Two kinds of data validation**

Kind of data validation	Purpose	Environment
Boundaries	Guardian	Production
Inside	Ease of development	Dev

**THEO:** When will you teach me about data validation inside the system?

**JOE:** Later. When the code base is bigger.

### 7.3 JSON schema in a nutshell

**THEO:** For now, the library management system is an application that runs in memory, with no database and no HTTP clients connected to it. But Nancy will probably want me to make the system into a real web server with clients, database and external services.

**JOE:** OK. Let's imagine how a client request for searching books would look.

**THEO:** Basically, a search request is made of a string and the fields you'd like to retrieve for the books whose title contains the string. So the request has two fields: `title` which is a string and `fields` which is an array of strings.

Theo quickly writes on the whiteboard, the example in [7.1](#).

#### **Listing 7.1 An example of a search request**

```
{
  "title": "habit",
  "fields": ["title", "weight", "number_of_pages"]
}
```

**JOE:** I see. Let me show you how to express the *schema* of a search request separately from the

representation of the search request data.

**THEO:** What do you mean exactly by *separately*?

**JOE:** Data representation stands on its own and the data schema stands on its own and you are free to validate that a piece of data conforms with a data schema as you will and when you will.

**TIP**

In DOP, data schema is separate from data representation.

**THEO:** It's a bit abstract for me.

**JOE:** I know. It will become much clearer in a moment. Now, I am going to show you how to build the data schema for the search request in a schema language called JSON Schema (<https://json-schema.org>).

**THEO:** I love JSON!

**JOE:** First, we have to express the data type of the request. What's the data type in the case of a book search request?

**THEO:** It's a map.

**JOE:** In JSON schema, the data type for maps is called: `object`. Look at this basic skeleton of a map: it's a map with two fields: `type` and `properties`. The value of `type` is "object" and the value of `properties` is a map with the schema for the map fields.

Joe writes on the whiteboard. It looks like the map in [7.2](#)

### **Listing 7.2 Basic schema skeleton of a map**

```
{
  "type": "object",
  "properties": {...}
}
```

**THEO:** I assume that inside properties we are going to express the schema of the map fields as JSON schema.

**JOE:** Correct.

**THEO:** I am starting to feel the dizziness of recursion.

**JOE:** In JSON schema, a schema is a JSON object with at least one field called `type`. For example, the type for the `title` field is `string` and...

**THEO:** ... the type for the `fields` field is `array`.

**JOE:** Right.

Theo comes to the whiteboard, and he fills the holes in the search request schema with the information about the fields. The result is in [7.3](#).

### **Listing 7.3 Schema skeleton for search request**

```
{
  "type": "object",
  "properties": {
    "title": {"type": "string"},
    "fields": {"type": "array"}
  }
}
```

On his way back from the whiteboard to Theo's desk, Joe makes a sign with his right hand that means: "Stay near the whiteboard, please".

**JOE:** We can be a little more precise about the `fields` property by providing information about the type of the elements in the array. In JSON schema, an array schema has a property called `items`, whose value is the schema for the array elements.

Without any hesitation, Theo adds this information on the whiteboard as in [7.4](#).

### **Listing 7.4 Schema for search request with information about the array elements**

```
{
  "type": "object",
  "properties": {
    "title": {"type": "string"},
    "fields": {
      "type": "array",
      "items": {"type": "string"}
    }
  }
}
```

Before going back to his desk, Theo asks Joe:

**THEO:** Are we done now?

**JOE:** Not yet. We can be more precise about the `fields` field in the search request. I assume that the fields in the request should be part of a closed list of fields. Therefore, instead of allowing any string, we could have a list of allowed values.

**THEO:** Like an enumeration value?

**JOE:** Exactly! In fact JSON schema supports enumeration values with the `enum` keyword. Instead of `{"type": "string"}`, you need to have `{"enum": [...]}` and fill the dots with the supported fields.

Theo fills the dots and he gets the schema in [7.5](#).

### Listing 7.5 Schema for the search request with enumeration values

```
{
  "type": "object",
  "properties": {
    "title": {"type": "string"},
    "fields": {
      "type": "array",
      "items": {
        "enum": [
          "publishers",
          "number_of_pages",
          "weight",
          "physical_format",
          "subjects",
          "publish_date",
          "physical_dimensions"
        ]
      }
    }
  }
}
```

**THEO:** Are we done, now?

**JOE:** Almost. We need to decide whether the fields of our search request are *optional* or *required*. In our case, both `title` and `fields` are required.

**THEO:** How do we express this information in JSON schema?

**JOE:** There is a field called `required` whose value is an array made of the names of the required fields in the map.

After adding the `required` field as in [7.6](#), Theo looks at Joe and this time he makes a move with his right hand that means: "now, you can come back to your desk".

### Listing 7.6 Schema of a search request

```
var searchBooksRequestSchema = {
  "type": "object",
  "properties": {
    "title": {"type": "string"},
    "fields": {
      "type": "array",
      "items": {
        "enum": [
          "publishers",
          "number_of_pages",
          "weight",
          "physical_format",
          "subjects",
          "publish_date",
          "physical_dimensions"
        ]
      }
    }
  },
  "required": ["title", "fields"]
};
```

**JOE:** Now, I'll show you how to validate a piece of data according to a schema.

**THEO:** What do you mean, *validate*?

**JOE:** Validating data according to a schema means checking whether data conforms to the schema. In our case, it means checking whether a piece of data is a valid 'search books' request.

**NOTE** Data validation in DOP means checking whether a piece of data conforms to a schema.

**THEO:** I see.

**JOE:** There are a couple of libraries that provides JSON schema validation. They provide a `validate` function that receives a schema and a piece of data and returns `true` when data is valid and `false` when data is not valid.

A list of schema validation libraries of the sort that Joe is referring to can be found in [7.2](#).

**Table 7.2 Libraries for JSON schema validation**

Language	Library	URL
JavaScript	Ajv	<a href="https://github.com/ajv-validator/ajv">https://github.com/ajv-validator/ajv</a>
Java	Snow	<a href="https://github.com/ssilverman/snowy-json">https://github.com/ssilverman/snowy-json</a>
C#	JSON.net Schema	<a href="https://www.newtonsoft.com/jsonschema">https://www.newtonsoft.com/jsonschema</a>
Python	jschon	<a href="https://github.com/marksparkza/jschon">https://github.com/marksparkza/jschon</a>
Ruby	JSONSchemer	<a href="https://github.com/davishmcclurg/json_schemer">https://github.com/davishmcclurg/json_schemer</a>

**THEO:** So if I call `validate` with this search request and that schema, it will return `true`?

Theo indicates the search request example from [7.7](#) and the schema from [7.6](#)

### **Listing 7.7 An example of a search request**

```
{
  "title": "habit",
  "fields": ["title", "weight", "number_of_pages"]
}
```

**JOE:** Give it a try and you'll see.

Indeed, when Theo executes the code snippet from [7.8](#), it returns `true`.

### Listing 7.8 Validating a valid search request

```
var searchBooksRequestSchema = {
  "type": "object",
  "properties": {
    "title": {"type": "string"},
    "fields": {
      "type": "array",
      "items": {"type": "string"}
    }
  },
  "required": ["title", "fields"]
};

var searchBooksRequest = {
  "title": "habit",
  "fields": ["title", "weight", "number_of_pages"]
};

validate(searchBooksRequestSchema, searchBooksRequest);
// true
```

**JOE:** Now, please try a invalid request.

**THEO:** Let me think about what kind of invalidity to try... I know, I'll make a typo in the title field and call it tilte with the l before the t.

As expected, the code snippet from [7.9](#) returns false.

### Listing 7.9 Validating a invalid search request

```
var invalidSearchBooksRequest = {
  "tilte": "habit",
  "fields": ["title", "weight", "number_of_pages"]
};

validate(searchBooksRequestSchema, invalidSearchBooksRequest);
// false
```

**THEO:** The syntax of JSON schema is much more verbose than the syntax for declaring the members in a class. Why is that so?

**JOE:** For two reasons. First, because JSON schema is language independent. It can be used in any programming language. As I told you, there are JSON schema validators available in most programming languages.

**THEO:** I see.

**JOE:** Second, JSON schema allows you to express validation conditions that are much harder—*if not impossible*—to express when data is represented with classes.

**TIP**

The expressive power of JSON schema is high!

**THEO:** Now you have triggered my curiosity. Can you give me some examples?

**JOE:** In a moment, we'll talk about schema composition. Some day I'll show you some examples of advanced validation.

**TIP**

Advanced validation is covered in Chapter 12.

**THEO:** What kind of advanced validation?

**JOE:** For instance, validating that a number falls within a given range, or validating that a string matches a regular expression.

**THEO:** Is there a way to get details about why the request is invalid?

**JOE:** Absolutely, I'll show you later. For now, let me show you how to make sure the response the server sends back to the client is valid.

**THEO:** It sounds much more complicated than a search book request!

**JOE:** Why?

**THEO:** Because a search response is made of multiple book results and in each book result some of the fields are optional!

## 7.4 Schema flexibility and strictness

**JOE:** Can you give me an example of what a book search response would look like?

**THEO:** Take a look at this example.

Theo points to his screen, indicating the example in [7.10](#).

**THEO** It's a search response with information about two books "7 Habits of Highly Effective People" and "The Power of Habit".

**JOE:** It's funny that you mention "The Power of Habit". I'm reading this book in order to get rid of my habit of biting my nails. Anyway, what fields are required and what fields are optional in a book search response?

**THEO:** In book information, the `title` and `available` fields are required. The other fields are optional.

### Listing 7.10 An example of a search response

```
[
  {
    "title": "7 Habits of Highly Effective People",
    "available": true,
    "isbn": "978-0812981605",
    "subtitle": "Powerful Lessons in Personal Change",
    "number_of_pages": 432
  },
  {
    "title": "The Power of Habit",
    "available": false,
    "isbn_13": "978-1982137274",
    "subtitle": "Why We Do What We Do in Life and Business",
    "subjects": [
      "Social aspects",
      "Habit",
      "Change (Psychology)"
    ]
  }
]
```

**JOE:** As I told you when we built the schema for the book search request, fields in a map are optional by default. In order to make a field mandatory, we have to include it in the `required` array.

The resulting schema is in [7.11](#).

**TIP**

In JSON schema, map fields are optional by default.

### Listing 7.11 Schema of a search response

```
var searchBooksResponseSchema = {
  "type": "array",
  "items": {
    "type": "object",
    "required": ["title", "available"],
    "properties": {
      "title": {"type": "string"},
      "available": {"type": "boolean"},
      "subtitle": {"type": "string"},
      "number_of_pages": {"type": "integer"},
      "subjects": {
        "type": "array",
        "items": {"type": "string"}
      },
      "isbn": {"type": "string"},
      "isbn_13": {"type": "string"}
    }
  }
};
```

**THEO:** I must admit that specifying a list of required fields is much simpler than having to specify that a member in a class is nullable!

**JOE:** Agreed.

**THEO:** On the other hand, I find the nesting of book information schema in the search response schema a bit hard to read.

**JOE:** Nothing prevents you from separating the book information schema from the search response schema.

**THEO:** How?

**JOE:** It's just JSON my friend. It means, you are free to manipulate the schema as any other map in your program. For instance, you could have the book information schema in a variable named `bookInfoSchema` and use it in the search books response schema, like this:

Joe indicates the code in [7.12](#).

### Listing 7.12 Schema of a search response

```
var bookInfoSchema = {
  "type": "object",
  "required": ["title", "available"],
  "properties": {
    "title": {"type": "string"},
    "available": {"type": "boolean"},
    "subtitle": {"type": "string"},
    "number_of_pages": {"type": "integer"},
    "subjects": {
      "type": "array",
      "items": {"type": "string"}
    },
    "isbn": {"type": "string"},
    "isbn_13": {"type": "string"}
  }
};

var searchBooksResponseSchema = {
  "type": "array",
  "items": bookInfoSchema
};
```

**THEO:** Once again, I have to admit that JSON schemas are more composable than class definitions.

<b>TIP</b>	<b>JSON schemas are just maps. We are free to compose and manipulate them like any other map.</b>
------------	---

**JOE:** Let's move on to validating data received from external data sources.

**THEO:** Is that different?

**JOE:** Not really. But I'll take it as an opportunity to show you some other features of JSON schema.

**THEO:** I'm curious to learn how data validation is used when we access data from the database.

**JOE:** Each time we access data from the outside, it's a good practice to validate it. Can you show me an example of how a database response for a search query would look?

**TIP**

**It's a good practice to validate data that comes from an external data source.**

**THEO:** When we query books from the database we expect to receive an array of books with 3 fields: `title`, `isbn` and `available`. The first two values should be strings and the third one should be a boolean.

**JOE:** Are those fields optional or required?

**THEO:** What do you mean?

**JOE:** Could there be books for which some of the fields are not defined?

**THEO:** No.

**JOE:** In that case the schema is quite simple. Would you try writing the schema for the database response?

**THEO:** Let me see. It's an array of objects where each object has three properties. Something like this?

Theo writes out the schema shown in [7.13](#).

### Listing 7.13 Schema of a database response

```
{
  "type": "array",
  "items": {
    "type": "object",
    "required": ["title", "isbn", "available"],
    "properties": {
      "title": {"type": "string"},
      "available": {"type": "boolean"},
      "isbn": {"type": "string"}
    }
  }
}
```

**JOE:** Well done, my friend! Now, I want to tell you about the `additionalProperties` in JSON schema.

**THEO:** What's that?

**JOE:** Take a look at this array.

Joe points to the the array in [7.14](#).

**JOE:** Is it a valid database response?

#### Listing 7.14 A book array with an additional property

```
[  
  {  
    "title": "7 Habits of Highly Effective People",  
    "available": true,  
    "isbn": "978-0812981605",  
    "dummy_property": 42  
  },  
  {  
    "title": "The Power of Habit",  
    "available": false,  
    "isbn": "978-1982137274",  
    "dummy_property": 45  
  }  
]
```

**THEO:** No. A database response should not have a `dummy_property` field. It should have only the three required fields specified in the schema.

**JOE:** It might be surprising but by default in JSON schema, fields not specified in the schema of an object are allowed. In order to disallow them, one has to set `additionalProperties` to `false`, like this:

Joe shows Theo the code in [7.15](#).

#### Listing 7.15 Schema of a database response, disallowing properties not mentioned in the schema

```
var booksFromDBSchema = {  
  "type": "array",  
  "items": {  
    "type": "object",  
    "required": ["title", "isbn", "available"],  
    "additionalProperties": false,  
    "properties": {  
      "title": {"type": "string"},  
      "available": {"type": "boolean"},  
      "isbn": {"type": "string"}  
    }  
  }  
};
```

**TIP**

In JSON schema, by default, fields not specified in the schema of a map are allowed.

**THEO:** Why is that?

**JOE:** The reason is that usually having additional fields in a map doesn't cause trouble. If your code doesn't care about a field, it will simply ignore it. But sometimes we want to be as strict as possible and we set `additionalProperties` to `false`.

**THEO:** What about the search request and response schema from the previous sections? Should we set `additionalProperties` to `false`?

**JOE:** That's an excellent question. I'd say it's a matter of taste. Personally, I like to allow additional fields in requests and disallow them in responses.

**THEO:** What's the advantage?

**JOE:** Well, the web server is responsible for the responses it sends to its clients. It makes sense then to be as strict as possible. However the requests are created by the clients and I prefer to do my best to serve my clients even when they are not as strict as they should be.

**TIP**

**It's a good practice to be strict regarding data that you send and to be flexible regarding data that you receive.**

**THEO:** "The client is always right". Isn't it?

**JOE:** Actually I prefer the way Jon Postel formulated his robustness principle: "Be conservative in what you send, be liberal in what you accept" ([https://en.wikipedia.org/wiki/Robustness\\_principle](https://en.wikipedia.org/wiki/Robustness_principle)).

## 7.5 Schema composition

**THEO:** What about validating data that comes from an external web service?

**JOE:** Can you give me an example?

**THEO:** In the near future, we'll have to integrate with a service called Open Library Books API (<https://openlibrary.org/dev/docs/api/books>) that provides detailed information about books.

**JOE:** Can you show me for, instance the, service response for Watchmen?

Theo taps a few keys on his keyboard, and brings up the response shown in [7.16](#).

**THEO:** Sure. Here you go.

### Listing 7.16 A Open Library Books API response example

```
{
  "publishers": [
    "DC Comics"
  ],
  "number_of_pages": 334,
  "weight": "1.4 pounds",
  "physical_format": "Paperback",
  "subjects": [
    "Graphic Novels",
    "Comics & Graphic Novels",
    "Fiction",
    "Fantastic fiction"
  ],
  "isbn_13": [
    "9780930289232"
  ],
  "title": "Watchmen",
  "isbn_10": [
    "0930289234"
  ],
  "publish_date": "April 1, 1995",
  "physical_dimensions": "10.1 x 6.6 x 0.8 inches"
}
```

Joe looks the JSON in [7.16](#) for a long moment.

Theo asks himself what could be so special in this JSON. While Joe is meditating about this piece of JSON, Theo writes down the JSON schema for the Books API response. It doesn't seem to be more complicated than any of the previous schemas. When Theo is done, he asks Joe to take a look at the schema in [7.17](#)

### Listing 7.17 Schema of a Open Library Books API response

```
{
  "type": "object",
  "required": ["title"],
  "properties": {
    "title": {"type": "string"},
    "publishers": {
      "type": "array",
      "items": {"type": "string"}
    },
    "number_of_pages": {"type": "integer"},
    "weight": {"type": "string"},
    "physical_format": {"type": "string"},
    "subjects": {
      "type": "array",
      "items": {"type": "string"}
    },
    "isbn_13": {
      "type": "array",
      "items": {"type": "string"}
    },
    "isbn_10": {
      "type": "array",
      "items": {"type": "string"}
    },
    "publish_date": {"type": "string"},
    "physical_dimensions": {"type": "string"}
  }
}
```

**JOE:** Good job!

**THEO:** That wasn't so hard. I really don't see why you looked at this JSON response for such a long time.

**JOE:** Well, it has to do with the `isbn_10` and `isbn_13` fields. I assume that they're not both mandatory.

**THEO:** Right. That's why I didn't include them in the `required` field of my schema in [7.17](#).

**JOE:** But one of them should always be there. Right?

**THEO:** Sometimes one of them and sometimes both of them like for *Watchmen*. It depends on the publication year of the book: books published before 2007 have `isbn_10` while books published after 2007 have `isbn_13`.

**JOE:** Oh I see. And *Watchmen* has both because it was originally published in 1986 but published again after 2007.

**THEO:** Correct.

**JOE:** Then, you need your schema to indicate that one of the `isbn` fields is mandatory. That's a good opportunity for me to tell you about JSON schema composition.

**THEO:** What's that?

**JOE:** It's a way to combine schemas, similarly to how we combine logical conditions with AND, OR and NOT.

**THEO:** I'd like to see that.

**JOE:** Sure. How would you express the schema for the Books API response as a composition of three schemas: `basicBookInfoSchema` (the schema that you wrote where only `title` is required), `mandatoryIsbn13` (a schema where only `isbn_13` is required) and `mandatoryIsbn10` (a schema where only `isbn_10` is required).

**THEO:** I think it should be `basicBookInfoSchema` AND (`mandatoryIsbn13` OR `mandatoryIsbn10`).

**JOE:** Exactly. The only thing is that in JSON schema, we use `allOf` instead of AND and `anyOf` instead of OR.

The result is in [7.18](#) and an example of usage in [7.19](#)

### Listing 7.18 Schema of an external API response

```

var basicBookInfoSchema = {
    "type": "object",
    "required": ["title"],
    "properties": {
        "title": {"type": "string"},
        "publishers": {
            "type": "array",
            "items": {"type": "string"}
        },
        "number_of_pages": {"type": "integer"},
        "weight": {"type": "string"},
        "physical_format": {"type": "string"},
        "subjects": {
            "type": "array",
            "items": {"type": "string"}
        },
        "isbn_13": {
            "type": "array",
            "items": {"type": "string"}
        },
        "isbn_10": {
            "type": "array",
            "items": {"type": "string"}
        },
        "publish_date": {"type": "string"},
        "physical_dimensions": {"type": "string"}
    }
};

var mandatoryISBN13 = {
    "type": "object",
    "required": ["isbn_13"]
};

var mandatoryISBN10 = {
    "type": "object",
    "required": ["isbn_10"]
};

var bookInfoSchema = {
    "allOf": [
        basicBookInfoSchema,
        {
            "anyOf": [mandatoryISBN13, mandatoryISBN10]
        }
    ]
};

```

### Listing 7.19 Validating an external API response

```

var bookInfo = {
  "publishers": [
    "DC Comics"
  ],
  "number_of_pages": 334,
  "weight": "1.4 pounds",
  "physical_format": "Paperback",
  "subjects": [
    "Graphic Novels",
    "Comics & Graphic Novels",
    "Fiction",
    "Fantastic fiction"
  ],
  "isbn_13": [
    "9780930289232"
  ],
  "title": "Watchmen",
  "isbn_10": [
    "0930289234"
  ],
  "publish_date": "April 1, 1995",
  "physical_dimensions": "10.1 x 6.6 x 0.8 inches"
};

validate(bookInfoSchema, bookInfo);
//true

```

**THEO:** I see why they call it `allOf` and `anyOf`. The first one means that data must conform to all the schemas and the second one means that data must conform to any of the schemas.

**JOE:** Yup.

**THEO:** Nice. With schema composition, JSON schema seems to have more expressive power than what I was used to when representing data with classes.

**JOE:** That's only the beginning. Some day, I'll show you more data validation conditions that can't be expressed when data is represented with classes.

**THEO:** Something still bothers me, though: when data isn't valid, you don't know what went wrong.

## 7.6 Details about data validation failures

**JOE:** So far we've treated JSON schema validation as though it were binary: either a piece of data is valid or it isn't.

**THEO:** Right.

**JOE:** But in fact, when a piece of data is not valid, we can get details about the reason of the invalidity.

**THEO:** Like when a required field is missing? Can we get the name of the missing field?

**JOE:** Yes. Or when a piece of data is not of the expected type, we can get information about that also.

**THEO:** That sounds very useful!

**JOE:** Indeed. Let me show you how it works. Until now, we used a generic `validate` function but when we deal with validation failures, we need to be more specific.

**THEO:** Why?

**JOE:** Because each data validator library has its own way of exposing the details of a data validation failure. For instance, in JavaScript `ajv`, the errors from the last data validation are stored, as an array, inside the validator instance.

**THEO:** Why an array?

**JOE:** Because there could be several failures. But let's start with the case of a single failure. Imagine we encounter a search book request where the title field is named `myTitle` instead of `title`. Take a look at this example. As you can see, we first instantiate a validator instance.

Joe indicates the code in [7.20](#).

### Listing 7.20 Accessing validation errors in ajv

```
var searchBooksRequestSchema = {
  "type": "object",
  "properties": {
    "title": {"type": "string"},
    "fields": {
      "type": "array",
      "items": {"type": "string"}
    }
  },
  "required": ["title", "fields"]
};

var invalidSearchBooksRequest = {
  "myTitle": "habit",
  "fields": ["title", "weight", "number_of_pages"]
};

var ajv = new Ajv(); ①
ajv.validate(searchBooksRequestSchema, invalidSearchBooksRequest);
ajv.errors ②
```

① Instantiation of a validator instance

② Display the validation errors

**THEO:** And what does the information inside the `errors` array look like?

**JOE:** Execute the code snippet and you'll see.

When Theo executes the code snippets from [7.20](#), he gets something that he finds hard to digest, as in [7.21](#).

### **Listing 7.21 Details about a single data validation failure in an array format**

```
[  
 {  
   "instancePath": "",  
   "schemaPath": "#/required",  
   "keyword": "required",  
   "params": {  
     "missingProperty": "title"  
   },  
   "message": "must have required property 'title'"  
 }  
]
```

**THEO:** I find the contents of the `errors` array a bit hard to digest.

**JOE:** Me too. Fortunately, `ajv` provides a `errorsText` utility function to convert the `errors` array in a human readable format. See for instance what is returned when you call `errorsText`.

Theo is shown the code in [7.22](#).

### **Listing 7.22 Display the errors in human readable format**

```
ajv.errorsText(ajv.errors);  
// "data must have required property 'title'"
```

**THEO:** Let me see what happens when there are more than one validation failure in the data.

**JOE:** By default, `ajv` catches only one validation error.

**THEO:** I guess that's for performance reasons. Once the validator encounters an error, it doesn't continue the data parsing.

**JOE:** Probably. Anyway, in order to catch more than one validation failure, you need to pass the `allErrors` options to the `Ajv` constructor, like this:

Joe points to [7.23](#).

**JOE:** We validate a search request with `myTitle` instead of `title` and numbers instead of strings in the `fields` array. As you can see in the output of the code snippet, three errors are returned.

**TIP**

**By default, `ajv` catches only the first validation failure.**

### Listing 7.23 Catching multiple validation failures

```

var searchBooksRequestSchema = {
  "type": "object",
  "properties": {
    "title": {"type": "string"},
    "fields": {
      "type": "array",
      "items": {"type": "string"}
    }
  },
  "required": ["title", "fields"]
};

var invalidSearchBooksRequest = { ❶
  "myTitle": "habit",
  "fields": [1, 2]
};

var ajv = new Ajv({allErrors: true}); ❷
ajv.validate(searchBooksRequestSchema,
             invalidSearchBooksRequest);

ajv.errorsText(ajv.errors); ❸
// "data must have required property 'title',
// /data/fields/0 must be string,
// /data/fields/1 must be string"

```

- ❶ A request with three failures
- ❷ Instantiation of the `Ajv` constructor with `allErrors: true` in order to catch more than one failure
- ❸ Convert the errors to a human readable format

**THEO:** I think I have all what I need in order to add data validation to the boundaries of my system, when Nancy asks me to make the library management system into a web server!

**JOE:** Would you allow me to give you a small gift as a symbol of our friendship?

**THEO:** I'd be honored.

Joe takes a small package out of his bag, wrapped in a kind of light green ribbon. He hands Theo the package with a solemn posture. When Theo undoes the ribbon, he discovers an elegant piece of paper decorated with pretty little designs. In the center of the paper, Theo manages to read the inscription "JSON schema cheatsheet". He smiles while browsing the JSON schema (see [7.24](#)). Then, Theo turns the paper over to find that the back is also filled with drawings. In the center of the paper he reads the inscription "Example of valid data" (see [7.25](#)).

### Listing 7.24 JSON schema cheatsheet

```
{
  "type": "array",      ①
  "items": {
    "type": "object",   ②
    "properties": {      ③
      "myNumber": { "type": "number" },  ④
      "myString": { "type": "string" },  ⑤
      "myEnum": { "enum": [ "myVal", "yourVal" ] },  ⑥
      "myBool": { "type": "boolean" }    ⑦
    },
    "required": [ "myNumber", "myString" ],  ⑧
    "additionalProperties": false  ⑨
  }
}
```

- ① At the root level, data is an array
- ② Each element of the array is a map
- ③ The properties of each field in the map
- ④ myNumber is a number
- ⑤ myString is a string
- ⑥ myEnum is a enumeration value with two possibilities: "myVal" and "yourVal"
- ⑦ myBool is a boolean
- ⑧ The mandatory fields in the map are myNumber and myString. Other fields are optional.
- ⑨ We don't allow fields that are not explicitly mentioned in the schema

### Listing 7.25 An example of valid data

```
[
  {
    ①
    "myNumber": 42,
    "myString": "Hello",
    "myEnum": "myVal",
    "myBool": true
  },
  {
    ②
    "myNumber": 54,
    "myString": "Happy"
  }
]
```

- ① This map is valid because all its fields are valid
- ② This map is valid because it contains all the required fields

## 7.7 Summary

- DOP Principle #4: Separate data schema and data representation.
- The *boundaries* of a system are defined to be the areas where the system exchanges data.
- Some examples of data validation at the boundaries of the system: validation of client requests and responses, validation of data that comes from external sources.
- Data validation in DOP means checking whether a piece of data conforms to a schema.
- When a piece of data is not valid, we get information about the validation failures and send this information back to the client in a human readable format.
- When data at system boundaries has been validated, it's not critical to validate data again inside the system.
- JSON schema is a language that allows us to separate data validation from data representation.
- JSON schema syntax is a bit verbose.
- The expressive power of JSON schema is high.
- JSON schemas are just maps and as so, we are free to manipulate them like any other maps in our programs.
- We can store a schema definition in a variable and use this variable in another schema.
- In JSON schema, map fields are optional by default.
- It's a good practice to validate data that comes from an external data source.
- It's a good practice to be strict regarding data that you send and to be flexible regarding data that you receive.
- `ajv` is a JSON schema library in JavaScript
- By default, `ajv` catches only the first validation failure.
- Advanced validation is covered in Chapter 12.

**Table 7.3 Libraries for JSON schema validation**

Language	Library	URL
JavaScript	Ajv	<a href="https://github.com/ajv-validator/ajv">https://github.com/ajv-validator/ajv</a>
Java	Snow	<a href="https://github.com/ssilverman/snowy-json">https://github.com/ssilverman/snowy-json</a>
C#	JSON.net Schema	<a href="https://www.newtonsoft.com/jsonschema">https://www.newtonsoft.com/jsonschema</a>
Python	jschon	<a href="https://github.com/marksparkza/jschon">https://github.com/marksparkza/jschon</a>
Ruby	JSONSchemer	<a href="https://github.com/davishmcclurg/json_schemer">https://github.com/davishmcclurg/json_schemer</a>



# *Advanced concurrency control*

## This chapter covers

- Atoms as an alternative to locks
- Managing a thread-safe counter and a thread-safe in-memory cache with atoms
- Managing the whole system state in a thread-safe way with atoms

## 8.1 No more deadlocks!

The traditional way to manage concurrency in a multi-threaded environment involves lock mechanisms like *mutexes*. Lock mechanisms tend to increase the complexity of the system as it's not trivial to make sure the system is free of deadlocks. In DOP, we leverage the fact that data is immutable and we use a lock-free mechanism called *atom* to manage concurrency.

Atoms are simpler to manage than locks because they are lock-free. As a consequence, the usual complexity of locks that is required to avoid deadlocks don't apply to them.

This chapter is mostly relevant to multi-threaded environments like Java, C#, Python and Ruby. It is less relevant to single-threaded-environments like JavaScript.

The JavaScript code snippets in this chapter are written as though JavaScript were multi-threaded.

## 8.2 The complexity of locks

This Sunday afternoon, while riding his bike on the Golden Bridge, Theo thinks about the Klamfim project with concern, not yet sure that betting on DOP was a good choice. Suddenly, Theo realizes that he hasn't yet scheduled the next session with Joe. He gets off his bike to call Joe. Bad luck, the line is busy. When Theo gets home, he tries to call Joe again, but once again it is busy. After dinner, Theo tries to call Joe one more time, with the same result: a busy signal. "Obviously, Joe is very busy today" Theo tells himself. Exhausted by his 50 mile bike ride at an average of 17 miles per hour, he falls asleep on the sofa.

When Theo wakes up, he's elated to see a text message from Joe: "See you Monday morning at 11am?". Theo answers with a thumbs up, and prepares for another week of work.

When Joe arrives at the office, Theo asks him why his phone was constantly busy the day before. Joe answers that he was about to ask Theo the very same question. They look at each other, puzzled, and then simultaneously break into laughter as they realize what has happened: in an amazing coincidence, they'd tried to phone each other at exactly the same times.

They both say at once:

—A deadlock!

When they come to Theo's desk, Joe tells him that today's session is going to be about concurrency management in multi-threaded environments.

**JOE:** How do you usually manage concurrency in a multi-threaded environment?

**THEO:** I protect access to critical sections with a lock mechanism, a *mutex* for instance.

**JOE:** When you say access, do you mean write access or also read access?

**THEO:** Both.

**JOE:** Why do you need to protect read access with a lock?

**THEO:** Because, without a lock protection, in the middle of a read, a write could happen in another thread and it would make my read logically inconsistent.

**JOE:** Another option would be to clone the data before processing it in a read.

**THEO:** Sometimes I would clone but in many cases, when data is large, it's too expensive to clone.

**TIP**

Cloning data to avoid read locks, doesn't scale.

**JOE:** In DOP, we don't need to clone or to protect read access.

**THEO:** Because data is immutable?

**JOE:** Right. When data is immutable, even if a write happens in another thread during a read, it won't make the read inconsistent, as the write never mutates the data that is read.

**THEO:** In a sense, a read always works on a data snapshot.

**JOE:** Exactly!

**TIP**

When data is immutable, read is always safe.

**THEO:** But what about write access? Don't you need to protect them with locks?

**JOE:** Nope.

**THEO:** Why not?

**JOE:** We have a simpler mechanism: it's called *atom*.

**THEO:** I am glad to hear there is something simpler than locks. I really struggle each time I have to integrate locks into a multi-threaded system.

**JOE:** Me too. I remember a bug we had in production, 10 years ago, when we forgot to release a lock when an exception was thrown in a critical section. It caused a terrible deadlock!

**THEO:** Deadlocks are really hard to avoid. Last year, we had a deadlock because two locks were not released in the proper order.

**JOE:** I have great news for you: with atoms, deadlocks never happen!

**TIP**

With atoms, deadlocks never happen.

**THEO:** That sounds great. Tell me more!

**TIP**

Atoms provide a way to manage concurrency without locks.

## 8.3 Thread-safe counter with atoms

**JOE:** Let's start with a simple case: a counter shared between threads.

**THEO:** What do you mean by a counter?

**JOE:** Imagine, we'd like to count the number of database accesses and write the total number of database accesses to the log every minute.

**THEO:** OK.

**JOE:** Could you write JavaScript code for this multi-threaded counter using locks?

**THEO:** But JavaScript is single threaded!

**JOE:** I know. But it's just for the sake of illustration. Imagine that JavaScript were multi-threaded and that it provided a `Mutex` object that you could lock and unlock.

**THEO:** It's a bit awkward. I guess it would look like this.

Theo shows Joe the code in [8.1](#).

### Listing 8.1 A thread-safe counter protected by a mutex

```
var mutex = new Mutex();
var counter = 0;

function dbAccess() {
    mutex.lock();
    counter = counter + 1;
    mutex.unlock();
    // access the database
}

function logCounter() {
    mutex.lock();
    console.log('Number of database accesses: ' + counter);
    mutex.unlock();
}
```

**JOE:** Excellent. Now, I am going to show you how to write the same code with atoms. An atom provides three methods: `get`, `set` and `swap`.

**JOE:** `get` returns the current value of the atom. `set` overwrites the current value of the atom. `swap` receives a function and updates the value of the atom with the result of the function called on the current value of the atom.

**Table 8.1** The three methods of an atom

Method	Description
<code>get</code>	Return the current value
<code>set</code>	Overwrite the current value
<code>swap</code>	Update the current value with a function

**THEO:** How would it look like to implement a thread-safe counter with an atom?

**JOE:** It's quite simple.

Joe shows Theo the code snippet in [8.2](#).

### Listing 8.2 A thread-safe counter stored in an atom

```
var counter = new Atom();
counter.set(0);

function dbAccess() {
    counter.swap(function(x) { ①
        return x + 1;
    });
    // access the database
}

function logCounter() {
    console.log('Number of database accesses: ' + counter.get());
}
```

- ① the argument `x` is the current value of the atom (same as `counter.get()`)

**THEO:** Could you tell me what's going on here?

**JOE:** Sure. First, we create an empty atom. Then, we initialize the value of the atom with `counter.set(0)`. In the logger thread, we read the current value of the atom with `counter.get()`.

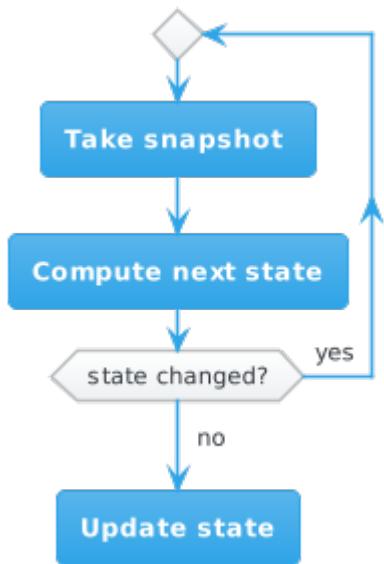
**THEO:** And how do you increment the counter in the threads that access the database?

**JOE:** We call `swap` with a function that receives `x` and returns `x + 1`.

**THEO:** I don't understand how `swap` could be thread-safe without using any locks.

Joe quickly sketches diagram [8.1](#) on the whiteboard.

**JOE:** You see, `swap` computes the next value of the atom and before modifying the current value of the atom, it checks whether the value of the atom has changed during the computation. If so, it tries again until no changes occur during the computation.



**Figure 8.1** High level flow of `swap`

**THEO:** Is it easy to implement?

**JOE:** Let me show you the implementation of the `Atom` class and you'll see.

Joe show him the code in [8.3](#).

### **Listing 8.3 Implementation of the `Atom` class**

```

class Atom {
    state;

    constructor() {}

    get() {
        return this.state;
    }

    set(state) {
        this.state = state;
    }

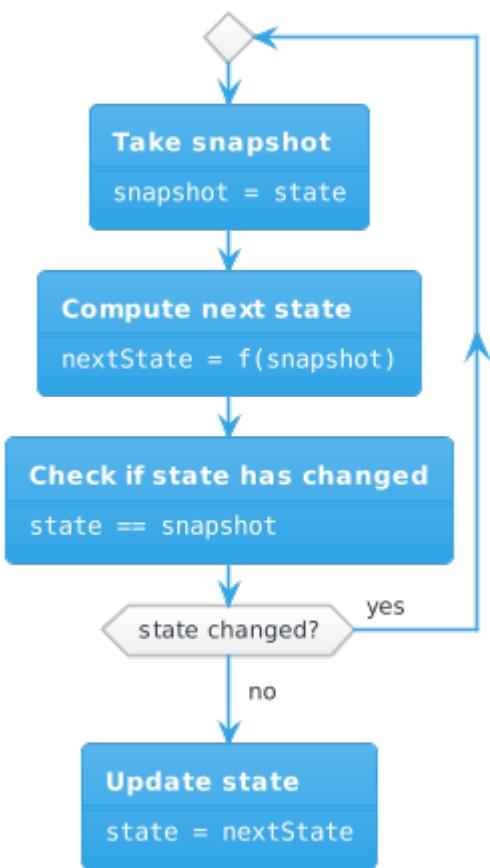
    swap(f) {
        while(true) {
            var stateSnapshot = this.state;
            var nextState = f(stateSnapshot);
            if (!atomicCompareAndSet(this.state, stateSnapshot, nextState)) { ①
                continue;
            }
            return nextState;
        }
    }
}

```

- ① We need a special thread-safe comparison operation here because `this.state` might have changed in another thread during execution of `f`

Theo comes closer to the white board and he modifies a bit Joe's diagram to make the flow of

the swap operation a bit more detailed. The resulting diagram is in [8.2](#).



**Figure 8.2 Detailed flow of `swap`**

**THEO:** What is exactly `atomicCompareAndSet`?

**JOE:** It's the core operation of an atom. It atomically sets the state to a new value if and only if the state equals the provided old value. It returns `true` upon success and `false` upon failure.

**THEO:** How could it be atomic without using locks?

**JOE:** That's a great question. In fact, `atomicCompareAndSet` is a compare-and-swap operation (<https://en.wikipedia.org/wiki/Compare-and-swap>) provided by the language that relies on a functionality of the CPU itself! For example, in Java the `java.util.concurrent.atomic` package has an `AtomicReference` generic class that provides a `compareAndSet()` method.

Implementations for other multi-threaded languages appear in [8.2](#).

**Table 8.2 Implementation of atomic compare and set in various languages**

Language	Link
Java	<a href="https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicReference.html">https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicReference.html</a>
JavaScript	Not relevant (single-threaded language)
Ruby	<a href="https://ruby-concurrency.github.io/concurrent-ruby/master/Concurrent/Atom.html">https://ruby-concurrency.github.io/concurrent-ruby/master/Concurrent/Atom.html</a>
Python	<a href="https://github.com/maxcountryman/atomos">https://github.com/maxcountryman/atomos</a>
C#	<a href="https://docs.microsoft.com/en-us/dotnet/api/system.threading.interlocked.compareexchange?view=net-5.0">https://docs.microsoft.com/en-us/dotnet/api/system.threading.interlocked.compareexchange?view=net-5.0</a>

**THEO:** Apropos Java, how would the implementation of an atom look?

**JOE:** It's quite the same besides the fact that `Atom` has to use generics and the inner state has to be stored in an `AtomicReference`.

Joe brings up the Java implementation of Atom [8.4](#) on his laptop and shows it to Theo.

#### **Listing 8.4 Implementation of the `Atom` class in Java**

```
class Atom<ValueType> {
    private AtomicReference<ValueType> state;

    public Atom() {}

    ValueType get() {
        return this.state.get();
    }

    void set(ValueType state) {
        this.state.set(state);
    }

    ValueType swap(UnaryOperator<ValueType> f) {
        while(true) {
            ValueType stateSnapshot = this.state.get();
            ValueType nextState = f(stateSnapshot);
            if (!this.state.compareAndSet(stateSnapshot, nextState)) {
                // this.state might have changed in another thread
                continue;
            }
        }
        return nextState;
    }
}
```

**THEO:** What about *using* an atom in Java?

**JOE:** Here, take a look. It's quite simple.

Joe shows Theo the code snippet in [8.5](#).

### Listing 8.5 Using an Atom in Java

```
Atom<Integer> counter = new Atom<Integer>();
counter.set(0);
counter.swap(x -> x + 1);
counter.get();
```

Theo takes a couple of minutes to meditate about this atom stuff and digest what he just learned. Then, he asks Joe:

**THEO:** What if `swap` never succeeds? I mean: could the `while` loop inside the code of `swap` be an infinite loop?

**JOE:** No. By definition, when `atomicCompareAndSet` fails on a thread, it means that the same atom was changed on another thread during the execution of `swap`. In this race between threads, there is always a winner.

**THEO:** But isn't it possible that some thread *never* succeeds because it always loses the race against other threads?

**JOE:** In theory, yes. But I've never encountered such a situation. If you have thousands of threads that do nothing beside swapping an atom, it could happen. But in practice, once the atom is swapped, the threads do some real work (e.g. database access or I/O) which gives an opportunity for other threads to swap the atom successfully.

**WARNING**

In theory atoms could create starvation in a system with thousands of threads that do nothing beside swapping an atom. In practice, once an atom is swapped, the threads do some real work (e.g. database access) which creates an opportunity for other threads to swap the atom successfully.

**THEO:** Interesting. Indeed, atoms look much easier to manage than locks.

**JOE:** Now let me show you how to use atoms with composite data.

**THEO:** Why would that be different?

**JOE:** Usually, dealing with composite data is more difficult than dealing with primitive types.

**THEO:** When you sold me DOP, you told me that we are able to manage data with the same simplicity as we manage numbers.

**TIP**

In DOP, data is managed with the same simplicity as numbers.

**JOE:** That's exactly what I am about to show you!

## 8.4 Thread-safe cache with atoms

**JOE:** Are you familiar with the notion of in-memory cache?

**THEO:** You mean memoization?

**JOE:** Kind of. Imagine that in your application, the database queries do not vary too much. It makes sense in that case to store the results of previous queries in memory, in order to improve the response time.

**THEO:** Of course.

**JOE:** What data structure would you use to store the in-memory cache?

**THEO:** Probably a string map where the keys are the queries and the values are the results from the database.

**TIP**

It's quite common to represent an in-memory cache as a string map.

**JOE:** Excellent. Could you write code to cache database queries in a thread-safe way, using a lock?

**THEO:** Let me see. I'm going to use an immutable string map. Therefore, I don't need to protect read access with a lock. Only the cache update needs to be protected.

**JOE:** You're getting the hang of this!

**THEO:** The code should be something like this.

Theo grabs his laptop, writes the code in [8.6](#) and shows it to Joe.

### Listing 8.6 Thread-safe cache with locks

```
var mutex = new Mutex();
var cache = {};  
  
function dbAccessCached(query) {
  var resultFromCache = _.get(cache, query);
  if (resultFromCache != nil) {
    return resultFromCache;
  }
  var result = dbAccess(query);
  mutex.lock();
  cache = _.set(cache, query, result);
  mutex.unlock();
  return result;
}
```

**JOE:** Nice. Now, let me show you how to write the same code using an *atom* instead of a *lock*. Take a look at this code and let me know if it's clear to you.

Joe types on his laptop for a bit and shows the code in [8.7](#) to Theo.

### Listing 8.7 Thread-safe cache with atoms

```
var cache = new Atom();
cache.set({});

function dbAccessCached(query) {
  var resultFromCache = _.get(cache.get(), query);
  if (resultFromCache != nil) {
    return resultFromCache;
  }
  var result = dbAccess(query);
  cache.swap(function(oldCache) {
    return _.set(oldCache, query, result);
  });
  return result;
}
```

**THEO:** I don't understand the function you're passing to the `swap` method.

**JOE:** The function passed to `swap` receives the current value of the cache, which is a string map, and returns a new version of the string map with an additional key-value pair.

**THEO:** I see. But something bothers me with the performance of the `swap` method in the case of a string map. How does the comparison work? I mean: comparing two string maps might take some time.

**JOE:** Not if you compare them by reference! As we discussed in the past, when data is immutable it is safe to compare by reference and it's super fast.

**TIP**

**When data is immutable, it is safe (and fast) to compare by reference.**

**THEO:** Cool. So atoms play well with immutable data.

**JOE:** Yes.

## 8.5 State management with atoms

**JOE:** Do you remember a couple of weeks ago when I showed you how we resolve potential conflicts between mutations, you told me that the code was not thread-safe?

**THEO:** Let me look again at the code and I'll tell you.

Theo takes a look at the code of the `SystemData` class that he wrote in [8.8](#) (without the validation logic, omitted to make the code easier to grasp). It takes him a few minutes to

remember how the `commit` method works. And suddenly, he has a Eureka moment!

### **Listing 8.8 SystemData class from Part 1**

```
class SystemState {
    systemData;

    get() {
        return this.systemData;
    }

    set(_systemData) {
        this.systemData = _systemData;
    }

    commit(previous, next) {
        this.systemData = SystemConsistency.reconcile(this.systemData,
                                                       previous,
                                                       next);
    }
}
```

**THEO:** This code is not thread-safe because the `SystemConsistency.reconcile` code inside the `commit` method is not protected. Nothing prevents from two threads to execute this code concurrently.

**JOE:** Exactly! Now, could you tell me how to make it thread-safe?

**THEO:** With locks?

**JOE:** Come on!

**THEO:** I was kidding. Of course, not with locks but with an atom.

**JOE:** Nice joke....

**THEO:** Let me see. I'd need to store the system data inside an atom. The `get` and `set` method of `SystemData` would simply call `get` and `set` methods of the atom.

Theo show Joe his code, as in [8.9](#).

**Listing 8.9** SystemData class with atom (except the commit method)

```
class SystemState {
    systemData;

    constructor() {
        this.systemData = new Atom();
    }

    get() {
        return this.systemData.get();
    }

    commit(prev, next) {
        this.systemData.set(next);
    }
}
```

**JOE:** Excellent. Now, for the fun part. Implement the `commit` method by calling the `swap` method of the atom.

**THEO:** Instead of calling `SystemConsistency.reconcile()` directly, I need to wrap it into a call to `swap`. Something like this?

Theo shows Joe his code as in [8.10](#).

**Listing 8.10** Implementation of `systemData.commit` with atom

```
SystemData.commit = function(previous, next) {
    this.systemData.swap(function(current) {
        return SystemConsistency.reconcile(current,
                                              previous,
                                              next);
    });
};
```

**JOE:** Perfect.

**THEO:** This atom stuff makes me think about what happened to us yesterday when we tried to call each other at the exact same time.

**JOE:** What do you mean?

**THEO:** I don't know but I am under the impression that mutexes are like phone calls and atoms are like text messages.

JOE smiles at Theo but doesn't reveal the meaning of his smile.

## 8.6 Summary

- Managing concurrency with atoms is much simpler than managing concurrency with locks as we don't have to deal at all with the risk of deadlocks.
- Cloning data to avoid read locks, doesn't scale.
- When data is immutable, read is always safe.
- Atoms provide a way to manage concurrency without locks.
- With atoms, deadlocks never happen.
- Using atoms for a thread-safe counter is trivial as the state of the counter is represented with a primitive type (an integer).
- We manage composite data in a thread-safe way with atoms.
- We make the highly scalable state management approach from Part 1, thread-safe, simply by keeping the whole system state inside an atom.
- It's quite common to represent an in-memory cache as a string map.
- When data is immutable, it is safe (and fast) to compare by reference.
- In theory, atoms could create starvation, in a system with thousands of threads that do nothing beside swapping an atom. In practice, once an atom is swapped, the threads do some real work (e.g. database access) so it gives an opportunity for other threads to swap the atom successfully.

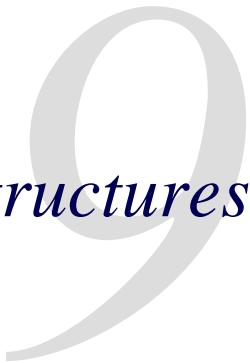
**Table 8.3 The three methods of an atom**

Method	Description
get	Return the current value
set	Overwrite the current value
swap	Update the current value with a function

**Table 8.4 Implementation of atomic compare and set in various languages**

Language	Link
Java	<a href="https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicReference.html">https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicReference.html</a>
JavaScript	Not relevant (single-threaded language)
Ruby	<a href="https://ruby-concurrency.github.io/concurrent-ruby/master/Concurrent/Atom.html">https://ruby-concurrency.github.io/concurrent-ruby/master/Concurrent/Atom.html</a>
Python	<a href="https://github.com/maxcountryman/atomos">https://github.com/maxcountryman/atomos</a>
C#	<a href="https://docs.microsoft.com/en-us/dotnet/api/system.threading.interlocked.compareexchange?view=net-5.0">https://docs.microsoft.com/en-us/dotnet/api/system.threading.interlocked.compareexchange?view=net-5.0</a>

# Persistent data structures



## This chapter covers

- The internal details of persistent data structures
- The time and memory efficiency of persistent data structures
- Using persistent data structures in an application

## 9.1 Standing on the shoulders of giants

In Part 1, we illustrated how to manage the state of a system without mutating data, where immutability is maintained by constraining ourselves to manipulate the state only with immutable functions, leveraging structural sharing.

In the current chapter, we present a safer and more scalable way to preserve data immutability by representing data with so-called persistent data structures.

There exist efficient implementations of persistent data structures for most programming languages via third-party libraries.

## 9.2 The need for persistent data structures

It's at the university that Theo meets Joe this time. When Theo asks Joe if today's topic is academic in nature, Joe tells him that the use of persistent data structures only became possible in programming languages following a discovery by a computer researcher named Phil Bagwell in 2001.<sup>7</sup> In 2007, Rich Hickey, the creator of Clojure, used this discovery as the foundation of persistent data structures in Clojure. Unveiling the secrets of these data structures to Theo in a university classroom is a way for Joe to honor the memory of Phil Bagwell who unfortunately passed away in 2012.

When they get to the university classroom, Joe starts the conversation with a question.

**JOE:** Are you getting used to DOP's prohibition against mutating data in place, and creating new versions instead?

**THEO:** I think so. But two things bother me about the idea of structural sharing that you showed me.

**JOE:** What bothers you my friend?

**THEO:** Safety and performance.

**JOE:** What do you mean by *safety*?

**THEO:** I mean that using immutable functions to manipulate data doesn't prevent it from being modified accidentally.

**JOE:** Right. Would you like me to show you the *naive* way to handle immutability or the *real* way?

**THEO:** What's are the pros and cons of each way?

**JOE:** The naive way is easy but not efficient while the real way is efficient but not easy.

**THEO:** Let's start with the naive way, then.

**JOE:** Each programming language provides its own way to protect data from being mutated.

**THEO:** How would I do that in Java for instance?

**JOE:** Java provides immutable collections and there is a way to convert a list or a map to an immutable list or an immutable map.

**WARNING**      **Immutable collections are not the same as persistent data structures.**

Joe brings up 2 listings on his screen, [9.1](#) and [9.2](#).

#### **Listing 9.1 Converting a mutable list to an immutable list in Java**

```
var myList = new ArrayList<Integer>();
myList.add(1);
myList.add(2);
myList.add(3);

var myImmutableList = List.of(myList.toArray());
```

## Listing 9.2 Converting a mutable map to an immutable map in Java

```
var myMap = new HashMap<String, Object>();
myMap.put("name", "Isaac");
myMap.put("age", 42);

var myImmutableMap = Collections.unmodifiableMap(myMap);
```

**THEO:** What happens when you try to modify an immutable collection?

**JOE:** A `UnsupportedOperationException` is thrown!

**THEO:** And in JavaScript?

**JOE:** JavaScript provides a `Object.freeze()` function that prevent data from being mutated. It works both with JavaScript arrays and objects.

Joe shows Theo the code we see in [9.3](#).

## Listing 9.3 Making an object immutable in JavaScript

```
var a = [1, 2, 3];
Object.freeze(a);

var b = {foo: 1};
Object.freeze(b);
```

**THEO:** And what happens when you try to modify a frozen object?

**JOE:** It depends. In JavaScript strict mode,<sup>8</sup> a `TypeError` exception is thrown, and in non-strict mode, it fails silently.

**THEO:** In case of a nested collection, are the nested collections also frozen?

**JOE:** No. But in JavaScript, one can write a `deepFreeze()` function that freezes an object recursively. It looks like this.

Joe show Theo the code in [9.4](#).

### Listing 9.4 Freezing an object recursively in JavaScript

```
function deepFreeze(object) {
    // Retrieve the property names defined on object
    const propNames = Object.getOwnPropertyNames(object);

    // Freeze properties before freezing self

    for (const name of propNames) {
        const value = object[name];

        if (value && typeof value === "object") {
            deepFreeze(value);
        }
    }

    return Object.freeze(object);
}
```

**THEO:** I see that it's possible to ensure that data is never mutated, which answers my concerns about safety. Now let me share with you my concerns about performance.

**TIP**

**It is possible to manually ensure that our data is not mutated but it is cumbersome.**

**JOE:** Sure.

**THEO:** If I understand correctly, the main idea behind structural sharing is that most of data is usually shared between two versions.

**JOE:** Correct.

**THEO:** This insight allowed us to create new versions of our collections using a shallow copy, instead of a deep copy. And you claimed that it was efficient.

**JOE:** Exactly!

**THEO:** Now here is my concern: In the case of a collection with many entries, a shallow copy might be expensive!

**JOE:** Could you give me an example of a collection with many entries?

**THEO:** A catalog with 100,000 books for instance.

**JOE:** On my machine, making a shallow copy of a collection with 100,000 entries doesn't take more than 50 milliseconds.

**THEO:** Sometimes, even 50 milliseconds per update isn't acceptable.

**JOE:** I totally agree with you. When one needs data immutability at scale, naive structural

sharing is not appropriate.

**THEO:** Also, shallow copying an array of 100,000 elements on each update would increase the program memory by 100 KB.

**JOE:** Indeed at scale, we have a problem both with memory and computation.

**TIP**

**At scale, naive structural sharing causes a performance hit both in terms of memory and computation.**

**THEO:** Is there a better solution?

**JOE:** Yes! For that, you'll need to learn the real way to handle immutability: It's called persistent data structures.

### 9.3 The efficiency of persistent data structures

**THEO:** In what sense are those data structures *persistent*?

**JOE:** Persistent data structures are so named because they always preserve their previous versions.

**TIP**

**Persistent data structures always preserve the previous version of themselves when they are modified.**

**JOE:** Persistent data structures address the two main limitations of naive structural sharing: safety and performance.

**THEO:** Let's start with safety: How do persistent data structures prevent data from being mutated accidentally?

**JOE:** In a language like Java, they implement the mutation methods of the collection interfaces by throwing a run time exception: `UnsupportedOperationException`.

**THEO:** And in a language like JavaScript?

**JOE:** In JavaScript, persistent data structures provide their own methods to access data and none of those methods mutate data.

**THEO:** Does it mean that we can't use the dot notation to access fields?

**JOE:** Correct. Fields of persistent data structures are accessed via a specific API.

**THEO:** What about efficiency? How do persistent data structures make it possible to create a

new version of a huge collection in an efficient way?

**JOE:** Persistent data structures organize data in such a way that we can leverage structural sharing at the level of the data structure.

**THEO:** Could you explain?

**JOE:** Let's start with the simplest data structure: a linked list. Imagine that you have a linked list with 100,000 elements.

**THEO:** OK.

**JOE:** What would it take to prepend an element to the head of the list?

**THEO:** You mean to create a new version of the list with an additional element?

**JOE:** Exactly!

**THEO:** Well, we could copy the list and then prepend an element to the list. But, it would be quite expensive.

**JOE:** And if I tell you that the original linked list is guaranteed to be immutable?

**THEO:** In that case, I could create a new list with a new head that points to the head of the original list. Something like this:

Theo goes to the classroom blackboard and draws the diagram shown in [9.1](#).

**JOE:** Would the efficiency of this operation depends on the size of the list?

**THEO:** No. It would be very efficient, no matter the size of the list.

**JOE:** That's what I mean by structural sharing at the level of the data structure itself. It relies on a simple but powerful insight: When data is immutable, it is safe to share it.

**TIP**

When data is immutable, it is safe to share it.

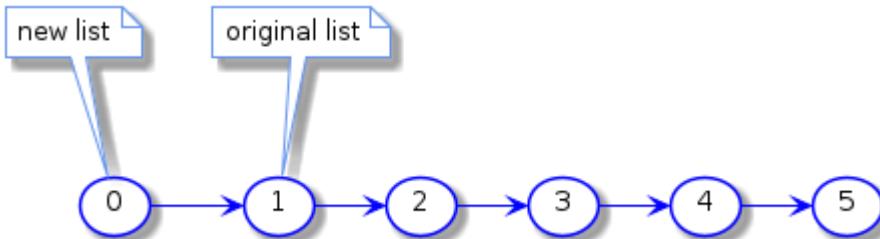


Figure 9.1 Structural sharing with linked lists

**THEO:** I understand how to leverage structural sharing at the level of the data structure for linked lists and prepend operations. But how would it work with operations like appending or modifying an *element* in a list?

**JOE:** For that purpose, we need to be smarter and represent our list as a tree.

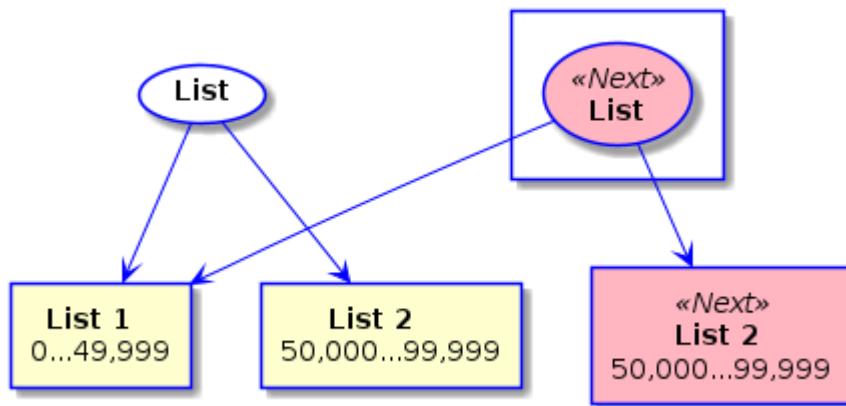
**THEO:** How does that help?

**JOE:** It helps because when a list is represented as a tree, most of the nodes in the tree can be shared between two versions of the list.

**THEO:** I am totally confused.

**JOE:** Imagine that you take a list with 100,000 elements and split it into two lists of 50,000 elements: Elements 0–49,999 in List #1 and elements 50,000–99,999 in List #2. How many operations would you need to create a new version of the list where a single element—let's say element at index #75,100—is modified?

It's hard for Theo to visualize mentally this kind of stuff. He goes back to the classroom blackboard and draws a diagram like [9.2](#). Once Theo looks at the diagram, it's easy for him to answer Joe's question.



**Figure 9.2 Structural sharing when a list of 100,000 elements is split**

**THEO:** List #1 could be shared, with 1 operation. And I'd need to create a new version of List #2, where element #75,100 is modified. It would take 50,000 operations. So overall, it's one operation of sharing and one operation of copying 50,000 elements. Overall, it's 50,001 operations.

**JOE:** Correct. You see that by splitting our original list in 2 lists, we were able to create a new version of the list with a number of operations in the order of the size of the list divided by 2.

**THEO:** I agree. But 50,000 is still a big number!

**JOE:** Indeed. But nobody prevents us from applying the same trick again and split List #1 and #2 in two lists.

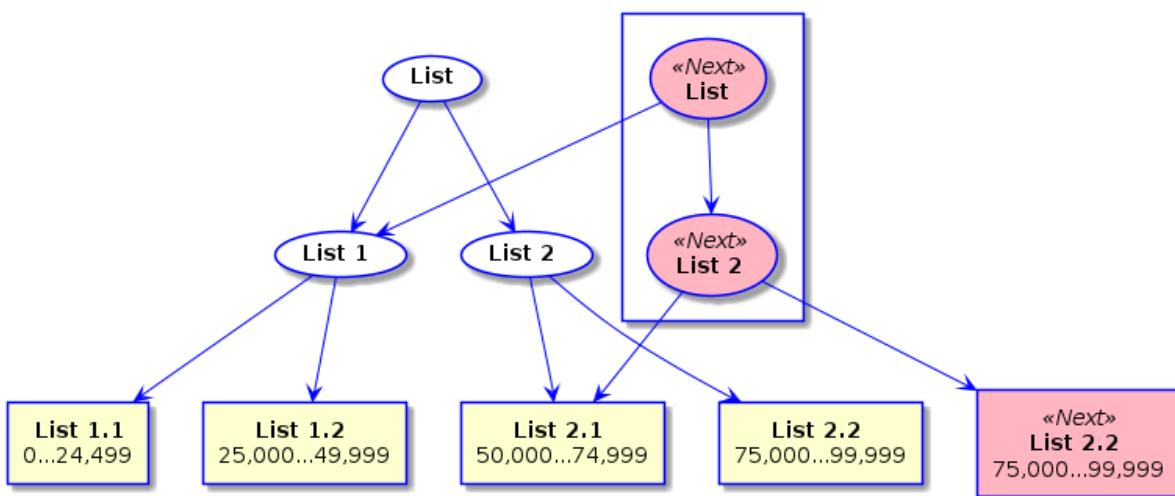
**THEO:** How exactly?

**JOE:** List #1.1 with elements 0–24,999, List #1.2 with elements 25,000–49,999, List #2.1 with elements 50,000–74,999 and List #2.2 with elements 75,000–99,999.

**THEO:** Could you draw it on the blackboard?

**JOE:** Sure.

Now, it's Joe that comes closer to the blackboard and draws a diagram like [9.3](#).



**Figure 9.3 Structural sharing when a list of 100,000 elements is split twice**

**THEO:** Let me count the number of operations for updating a single element. It takes 2 operations of sharing and one operation of copying 25,000 elements. Overall, it takes 25,002 operations to create a new version of the list.

**JOE:** Correct!

**THEO:** Let's split again then!

**JOE:** Absolutely. In fact we split again and again until the size of the lists is at most 2. Could you guess what is the complexity of creating a new version then?

**THEO:** I'd say around  $\log_2 N$  operations.

**JOE:** I see that you remember well your material from school. Do you have a gut feeling about what is  $\log_2 N$  when  $N$  is 100,000?

**THEO:** Let me see. 2 to the 10 is around 1000. 2 to the 7 is 128. So it should be a bit less than

17.

**JOE:** It's 16.6 to be precise. It means that in order to update an element in a persistent list of 100,000 elements, we need around 17 operations. And the same for accessing elements.

**THEO:** Nice. But 17 is still not negligible.

**JOE:** I agree. We can easily improve the performance of accessing elements by using a higher branching factor in our tree.

**THEO:** What do you mean?

**JOE:** Instead of splitting by 2 at each level, we could split by 32.

**THEO:** But the running time of our algorithm would still grow with  $\log n!$

**JOE:** You're right. From a *theoretical* perspective it's the same. But from a *practical* perspective it makes a big difference.

**THEO:** Why?

**JOE:** Because  $\log_{32} n$  is 5 times lower than  $\log_2 n!$

**THEO:** That's true. 2 to the 5 is 32.

**JOE:** Back to our list of 100,000 elements: Could you tell me how many operations are required to access an element if the branching factor is 32?

**THEO:** With a branching factor of 2, it was 16.6. If I divide it by 5, I get 3.3.

**JOE:** Correct.

**TIP**

By using a branching factor of 32, we make elements access in persistent lists efficient.

**THEO:** Does this trick also improve the performance of updating an element in a list?

**JOE:** Yes.

**THEO:** How? We'd have to copy 32 elements at each level instead of 2 elements. So it's a 16x performance hit that's not compensated for by the fact that the tree depth is reduced by 5x!

**JOE:** I see that you are quite sharp with numbers. There is another thing to take into consideration in our practical analysis of the performance: modern CPU architecture.

**THEO:** Interesting. The more you tell me about persistent data structures, the more I understand

why you wanted to have this session at a university.

**JOE:** Modern CPUs read and write data from and to the main memory in units of cache lines, often 32 or 64 bytes long.

**THEO:** What difference does it make?

**JOE:** A nice consequence of this data access pattern is that copying an array of size 32 is much faster than copying 16 arrays of size 2 that belong to different levels of the tree.

**THEO:** Why is that?

**JOE:** The reason is that copying an array of size 32 can be done in a single pair of cache accesses: one for read and one for write. While for arrays that belong to different tree level, each array requires its own pair of cache accesses, even if there are only 2 elements in the array.

**THEO:** In other words, the performance of updating a persistent list is dominated by the depth of the tree.

**TIP**

In modern CPU architectures, the performance of updating a persistent list is dominated much more by the depth of the tree than by the number of nodes at each level of the tree.

**JOE:** That's correct up to a certain point. In today's CPUs, using a branching factor of 64 would in fact decrease the performance of update operations.

**THEO:** I see.

**JOE:** Now I am going to make another interesting claim that is not accurate from a theoretical perspective but accurate in practice.

**THEO:** What is it?

**JOE:** The number of operations it takes to get or update an element in a persistent list with branching factor 32 is constant.

**THEO:** How could it be? You just made the point that the number of operations was  $\log_{32} N$ !

**JOE:** Be patient. What is the highest number of elements that you could have in a list, in practice?

**THEO:** I don't know. I never thought about that.

**JOE:** Let's assume that it takes 4 bytes to store an element in a list.

**THEO:** OK.

**JOE:** Now, could you tell me how much memory it would take to hold a list with 10 billion elements?

**THEO:** You mean 1 with 10 zeros?

**JOE:** Yes.

**THEO:** Each element takes 4 bytes so it would be around 40 GB!

**JOE:** Correct. Do you agree that it doesn't make sense to hold a list that takes 10GB of memory?

**THEO:** I agree.

**JOE:** So let's take 10 billion as an upper bound to the number of elements in a list. What is  $\log_{32}$  of 10 billion?

Once again, Theo uses the blackboard to clarify his thoughts.

**THEO:** 1 billion is approximately  $2^{30}$ . Therefore 10 billion is around  $2^{33}$ . It means that  $\log_2$  of 10 billion is 33. Therefore  $\log_{32}$  of 10 billion should be around  $33/5$ , which is a bit less than 7.

**JOE:** I am impressed again by your sharpness with numbers. To be precise  $\log_{32}$  of 10 billion is 6.64.

**THEO:** I didn't get that far.

**JOE:** Did I convince you that in practice, accessing or updating an element in a persistent list is essentially constant?

**THEO:** Yes. And I find it quite amazing!

**TIP**

Persistent lists can be manipulated in near constant time.

**JOE:** Me too.

**THEO:** What about persistent maps?

**JOE:** It's quite similar but I don't think we have time to discuss it now.

Theo looks at his watch and he notices that it's time to get back to the office and have lunch.

## 9.4 Persistent data structures libraries

On their way back to the office, Theo and Joe don't talk too much. Theo's thoughts take him back to what he learned in the university classroom. He feels lot of respect for Phil Bagwell, who discovered how to manipulate persistent data structures efficiently, and for Rich Hickey who created a programming language incorporating that discovery as a core feature and making it available to the world.

Immediately after lunch Theo asks Joe to show him what it looks like to manipulate persistent data structures for real in a programming language.

**THEO:** Are persistent data structures available in all programming languages?

**JOE:** A few programming languages provide them as part of the language like Clojure, Scala and C#. But in most programming languages, you need a third-party library.

**THEO:** Could you give me a few references?

**JOE:** Sure. Immutable.js for JavaScript (<https://immutable-js.com/>), Paguro for Java (<https://github.com/GlenKPeterson/Paguro>), Immutable Collections for C# (<https://docs.microsoft.com/en-us/archive/msdn-magazine/.../march/net-framework-immutable-collections>), Pyrsistent for Python (<https://github.com/tobgu/pyrsistent>) and Hamster for Ruby (<https://github.com/hamstergem/hamster>).

**Table 9.1 Persistent data structure libraries**

Language	Library
JavaScript	Immutable.js
Java	Paguro
C#	provided by the language
Python	Pyrsistent
Ruby	Hamster

**THEO:** What does it take to integrate persistent data structures provided by a third-party library into your code?

### 9.4.1 Persistent data structures in Java

**JOE:** In an Object-Oriented language like Java, it's quite straightforward to integrate persistent data structures in a program, as persistent data structures implement collection interfaces, beside the parts of the interface that mutate in place.

**THEO:** What do you mean?

**JOE:** Take for instance, Paguro for Java: Paguro persistent maps implement read-only methods of `java.util.Map`, like `get()` and `containsKey()` but not methods like `put()` and `remove()`. And Paguro vectors implement read-only methods of `java.util.List`, like `get()` and `size()` but not methods like `set()`.

**THEO:** What happens when we call `put()` or `remove()` on a Paguro map?

**JOE:** It throws a `UnsupportedOperationException` exception!

**THEO:** What about iterating over the elements of a Paguro collection with a `forEach()`?

**JOE:** It works like with any Java collection.

Joe shows Theo the code snippet in [9.5](#).

### Listing 9.5 Iterating over a Paguro vector

```
var myVec = PersistentVector.ofIter(List.of(10, 2, 3)); ①
for (Integer i : myVec) {
    System.out.println(i);
}
```

① Creating a Paguro vector from a Java list

**THEO:** What about Java streams?

**JOE:** Paguro collections are Java collections. Therefore they support Java stream interface.

Joe shows Theo the code snippet in [9.6](#).

### Listing 9.6 Streaming a Paguro vector

```
var myVec = PersistentVector.ofIter(List.of(10, 2, 3));
vec1.stream().sorted().map(x -> x + 1);
```

**TIP** Paguro collections implement the read-only parts of Java collection interfaces. Therefore, they can be passed to any methods that expect to receive a Java collection without mutating it.

**THEO:** So far, you told me how do use Paguro collections as Java read-only collections. But how do I make modifications to Paguro persistent data structures?

**JOE:** In a similar way to the `_.set()` function of Lodash FP that we introduced earlier: instead of mutating in place, you create a new version.

**THEO:** What methods does Paguro expose for creating new versions of a data structure?

**JOE:** For vectors, you use `replace()` and for maps, you use `assoc()`.

Joe shows Theo the code in in [9.7](#) and [9.8](#).

### Listing 9.7 Creating a modified version of a Paguro vector

```
var myVec = PersistentVector.ofIter(List.of(10, 2, 3));
var myNextVec = myVec.replace(0, 42);
```

### Listing 9.8 Creating a modified version of a Paguro map

```
var myMap = PersistentHashMap.of(Map.of("aa", 1, "bb", 2).entrySet()); ①
var myNextMap = myMap.assoc("aa", 42);
```

- ① Creating a Paguro map from a java map entry set

**THEO:** I see.

## 9.4.2 Persistent data structures in JavaScript

**JOE:** In a language like JavaScript, it's a bit more cumbersome to integrate persistent data structures.

**THEO:** How so?

**JOE:** Because JavaScript objects and arrays don't expose any interface.

**THEO:** Bummer.

**JOE:** It's not as terrible as it sounds because `immutable.js` exposes its own set of functions to manipulate its data structures.

**THEO:** What do you mean?

**JOE:** I'll show you in a moment. But first let me show you how to initiate `Immutable.js` persistent data structures.

**THEO:** OK.

**JOE:** `immutable.js` provides a handy function that recursively converts a native data object to an immutable one. It's called `Immutable.fromJS()`.

**THEO:** What do you mean by recursively?

**JOE:** Consider the map that holds library data from our library management system: It has values that are themselves maps. `Immutable.fromJS()` converts the nested maps into

immutable maps.

**THEO:** Could you show me some code?

**JOE:** Absolutely. Take a look.

Joe shows Theo the code snippet for library data in [9.9](#).

### Listing 9.9 Conversion to immutable data

```
var libraryData = Immutable.fromJS({
  "catalog": {
    "booksByIsbn": {
      "978-1779501127": {
        "isbn": "978-1779501127",
        "title": "Watchmen",
        "publicationYear": 1987,
        "authorIds": ["alan-moore",
                      "dave-gibbons"]
      }
    },
    "authorsById": {
      "alan-moore": {
        "name": "Alan Moore",
        "bookIsbns": ["978-1779501127"]
      },
      "dave-gibbons": {
        "name": "Dave Gibbons",
        "bookIsbns": ["978-1779501127"]
      }
    }
  });
});
```

**THEO:** Do you mean that the `catalog` value in `libraryData` map is itself an immutable map?

**JOE:** Yes. And the same for `booksByIsbn`, `authorIds` etc..

**THEO:** Cool! And how do I access a field inside an immutable map?

**JOE:** As I told you, `Immutable.js` provides its own API for data access. For instance, in order to access a field inside an immutable map, you use `Immutable.get()` or `Immutable.getIn()`, like this:

Joe shows Theo the code in [9.10](#).

### Listing 9.10 Accessing a field and a nested field in an immutable map

```
Immutable.get(libraryData, "catalog");
Immutable.getIn(libraryData, ["catalog", "booksByIsbn", "978-1779501127", "title"]);
// "Watchmen"
```

**THEO:** And how do I make a modification to a map?

**JOE:** In a similar way to what we did with Lodash FP in the past: you use `Immutable.set()` or

`Immutable.setIn()` map to create a new version of the map where a field is modified.

Joe shows Theo the code in [9.11](#)

### Listing 9.11 Creating a new version of a map where a field is modified

```
Immutable.setIn(libraryData,
  ["catalog", "booksByIsbn",
   "978-1779501127", "publicationYear"],
  1988);
```

**THEO:** What happens when I try to access a field in the map using JavaScript's dot or bracket notation?

**JOE:** You access the internal representation of the map instead of accessing a map field.

**THEO:** Does that mean that we can't pass data from `Immutable.js` to `Lodash` for data manipulation?

**JOE:** Yes. But, it's quite easy to convert any `Immutable` collection into a native JavaScript object back and forth.

**THEO:** How?

**JOE:** `Immutable.js` provides a `toJS()` method to convert an arbitrary deeply nested `Immutable` collection into a JavaScript object.

**THEO:** But if I have a huge collection, it could take lots of time to convert it!

**JOE:** True. We need a better solution. `Immutable.js` provides its own set of data manipulation functions, like `map()`, `filter()` and `reduce()`.

**THEO:** And if I need more data manipulation, like `Lodash _ .groupBy()`?

**JOE:** You could write your own data manipulation functions that work with `Immutable.js` collections or use a library like `mudash` (<https://github.com/brienneisler/mudash>) that provides a port of `Lodash` to `Immutable.js`.

**THEO:** What would you advise?

**JOE:** I am going to show you how to port functions from `Lodash` to `Immutable.js` and how to adapt the code from your Library Management System.

## 9.5 Persistent data structures in action

**JOE:** Let's start with our search query. Could you look at the current code and tell me the `Lodash` functions that we used to implement the search query?

**THEO:** Including the code for the unit tests?

**JOE:** Of course!

### 9.5.1 Writing queries with persistent data structures

**THEO:** The Lodash functions we used are: `get`, `map`, `filter` and `isEqual`.

**JOE:** He's the port of those 4 functions to `Immutable.js`.

Joe shows Theo [9.12](#).

#### Listing 9.12 Porting `get`, `map`, `filter` and `isEqual` from Lodash to `Immutable.js`

```
Immutable.map = function(coll, f) {
  return coll.map(f);
};

Immutable.filter = function(coll, f) {
  if(Immutable.isMap(coll)) {
    return coll.valueSeq().filter(f);
  }
  return coll.filter(f);
};

Immutable isEqual = Immutable.is;
```

**THEO:** The code seems quite simple. But could you explain it to me function by function?

**JOE:** Sure. Let's start with `get`: For accessing a field in a map, `Immutable.js` provides two functions: `get` for direct fields and `getIn` for nested fields. It's different than in Lodash where `_.get` works both on direct and nested field.

**THEO:** What about `map`?

**JOE:** `Immutable.js` provides its own `map` function. The only difference is that it is a method of the collection. But we can easily adapt.

**THEO:** What about `filter`? How would you make it work both for arrays and maps like Lodash's `filter`?

**JOE:** `Immutable.js` provides a `valueSeq` method that returns the values of a map.

**THEO:** Cool. And what about `isEqual` to compare two collections?

**JOE:** That's very easy. `Immutable.js` provides a function named `is` that works exactly as `isEqual`.

**THEO:** So far so good. What do I need to do now to make the code of the search query work with `Immutable.js`?

**JOE:** You simply replace each occurrence of `_` with `Immutable`: `_map` becomes `Immutable.map`, `_filter` becomes `Immutable.filter` and `_isEqual` becomes `Immutable.isEqual`.

**THEO:** I can't believe it's so easy!

**JOE:** Try it on your own and you'll see! Sometimes, it's a bit more cumbersome as you need to convert the JavaScript objects to `Immutable.js` objects using `Immutable.fromJS`.

Theo copies and pastes the snippets for the code and the unit tests of the search query. Then, he asks his IDE to replace `_` with `Immutable` and the results are in [9.13](#) for the code and in [9.14](#) for the test. When Theo executes the tests and they pass, he is surprised but satisfied.

### Listing 9.13 Implementing book search with persistent data structures

```
class Catalog {
  static authorNames(catalogData, authorIds) {
    return Immutable.map(authorIds, function(authorId) {
      return Immutable.getIn(catalogData, ["authorsById", authorId, "name"]);
    });
  }

  static bookInfo(catalogData, book) {
    var bookInfo = Immutable.Map({
      "title": Immutable.get(book, "title"),
      "isbn": Immutable.get(book, "isbn"),
      "authorNames": Catalog.authorNames(catalogData, Immutable.get(book, "authorIds"))
    });
    return bookInfo;
  }

  static searchBooksByTitle(catalogData, query) {
    var allBooks = Immutable.get(catalogData, "booksByIsbn");
    var queryLowerCased = query.toLowerCase();
    var matchingBooks = Immutable.filter(allBooks, function(book) {
      return Immutable.get(book, "title").
        toLowerCase().
        includes(queryLowerCased);
    });
    var bookInfos = Immutable.map(matchingBooks, function(book) {
      return Catalog.bookInfo(catalogData, book);
    });
    return bookInfos;
  }
}
```

### Listing 9.14 Testing book search with persistent data structures

```

var catalogData = Immutable.fromJS({
  "booksByIsbn": {
    "978-1779501127": {
      "isbn": "978-1779501127",
      "title": "Watchmen",
      "publicationYear": 1987,
      "authorIds": ["alan-moore",
                     "dave-gibbons"]
    }
  },
  "authorsById": {
    "alan-moore": {
      "name": "Alan Moore",
      "bookIsbns": ["978-1779501127"]
    },
    "dave-gibbons": {
      "name": "Dave Gibbons",
      "bookIsbns": ["978-1779501127"]
    }
  }
});

var bookInfo = Immutable.fromJS({
  "isbn": "978-1779501127",
  "title": "Watchmen",
  "authorNames": ["Alan Moore",
                  "Dave Gibbons"]
});

Immutable.isEqual(
  Catalog.searchBooksByTitle(catalogData, "Watchmen"),
  Immutable.fromJS([bookInfo]));
//true

Immutable.isEqual(
  Catalog.searchBooksByTitle(catalogData, "Batman"),
  Immutable.fromJS([]));
//true

```

## 9.5.2 Writing mutations with persistent data structures

**THEO:** Shall we move forward and port the add member mutation?

**JOE:** Porting the add member mutation from Lodash to Immutable.js only requires you to replace `_` with `Immutable`. Look:

Joe shows Theo [9.15](#) and [9.16](#).

**THEO:** And for the tests, I shouldn't forget to convert the JavaScript objects to Immutable.js objects with `Immutable.fromJS()`.

### Listing 9.15 Implementing member addition with persistent data structures

```
UserManagement.addMember = function(userManagement, member) {
  var email = Immutable.get(member, "email");
  var infoPath = ["membersByEmail", email];
  if(Immutable.hasIn(userManagement, infoPath)) {
    throw "Member already exists.";
  }
  var nextUserManagement =  Immutable.setIn(userManagement,
                                             infoPath,
                                             member);
  return nextUserManagement;
};
```

### Listing 9.16 Testing member addition with persistent data structures

```
var jessie = Immutable.fromJS({
  "email": "jessie@gmail.com",
  "password": "my-secret"
});

var franck = Immutable.fromJS({
  "email": "franck@gmail.com",
  "password": "my-top-secret"
});

var userManagementStateBefore = Immutable.fromJS({
  "membersByEmail": {
    "franck@gmail.com": {
      "email": "franck@gmail.com",
      "password": "my-top-secret"
    }
  }
});

var expectedUserManagementStateAfter = Immutable.fromJS({
  "membersByEmail": {
    "jessie@gmail.com": {
      "email": "jessie@gmail.com",
      "password": "my-secret"
    },
    "franck@gmail.com": {
      "email": "franck@gmail.com",
      "password": "my-top-secret"
    }
  }
});

var result = UserManagement.addMember(userManagementStateBefore, jessie);
Immutable.isEqual(result, expectedUserManagementStateAfter);
//true
```

### 9.5.3 Serialization and deserialization

**THEO:** Does Immutable.js also support JSON serialization and deserialization?

**JOE:** It supports serialization out of the box. And for deserialization, we need to write our own function.

**THEO:** Does Immutable.js provide a `Immutable.stringify()` function?

**JOE:** That's not necessary as the native `JSON.stringify()` function works with `Immutable.js` objects as you can see in [9.17](#).

### Listing 9.17 JSON serialization of a `Immutable.js` collection with `JSON.stringify()`

```
var bookInfo = Immutable.fromJS({
  "isbn": "978-1779501127",
  "title": "Watchmen",
  "authorNames": ["Alan Moore",
    "Dave Gibbons"]
});

JSON.stringify(bookInfo);
//{"isbn":"978-1779501127","title":"Watchmen",
// "authorNames":["Alan Moore","Dave Gibbons"]}
```

**THEO:** How could it be that `JSON.stringify()` knows to handle a `Immutable.js` collection?

**JOE:** As an OOP developer, you shouldn't be surprised by that.

**THEO:** Let me think.

After a few moments, Theo has a guess.

**THEO:** Is that because `JSON.stringify()` calls some method on its argument?

**JOE:** Exactly! If the object passed to `JSON.stringify()` has a `.toJSON()` method, it will be called by `JSON.stringify()`.

**THEO:** Nice. What about JSON deserialization?

**JOE:** That needs to be done in two steps: You convert the JSON string to a JavaScript object and then to an immutable collection.

**THEO:** Something like the code in [9.18](#)?

### Listing 9.18 Converting a JSON string into an immutable collection

```
Immutable.parseJSON = function(jsonString) {
  return Immutable.fromJS(JSON.parse(jsonString));
};
```

**JOE:** Exactly.

## 9.5.4 Data diff

**THEO:** So far we have ported pieces of code that dealt with simple data manipulations. I'm curious to see how it goes with complex data manipulations like the code that compute the data diff between two maps.

**JOE:** That also works smoothly. But we need to port another 8 functions. Here they are.

Joe finds the code in [9.19](#) on his laptop and shows it to Theo.

### **Listing 9.19 Porting Lodash functions involved in data diff computation**

```
Immutable.reduce = function(coll, reducer, initialReduction) {
  return coll.reduce(reducer, initialReduction);
};

Immutable.isEmpty = function(coll) {
  return coll.isEmpty();
};

Immutable.keys = function(coll) {
  return coll.keySeq();
};

Immutable.isObject = function(coll) {
  return Immutable.Map.isMap(coll);
};

Immutable.isArray = Immutable.isIndexed;

Immutable.union = function() {
  return Immutable.Set.union(arguments);
};
```

**THEO:** Everything looks trivial exception the usage of `arguments` in `Immutable.union`.

**JOE:** In JavaScript, `arguments` is an implicit array-like object that contains the values of the function arguments.

**THEO:** I see. It's one of those pieces of JavaScript magic!

**JOE:** We need to use `arguments` because Lodash and `Immutable.js` slightly differ in the signature of the `union` function: `Immutable.Set.union` receives an array of lists while `_.union` receives several arrays.

**THEO:** Makes sense.

Once again, Theo is surprised to discover that after replacing `_` with `Immutable` in [9.20](#) and [9.21](#) the tests pass.

## Listing 9.20 Implementing data diff with persistent data structures

```

function diffObjects(data1, data2) {
    var emptyObject = Immutable.isArray(data1) ? Immutable.fromJS([]) : Immutable.fromJS({});
    if(data1 == data2) {
        return emptyObject;
    }
    var keys = Immutable.union(Immutable.keys(data1), Immutable.keys(data2));
    return Immutable.reduce(keys,
        function (acc, k) {
            var res = diff(Immutable.get(data1, k),
                Immutable.get(data2, k));
            if((Immutable.isObject(res) && Immutable.isEmpty(res)) ||
                (res == "data-diff:no-diff")) {
                return acc;
            }
            return Immutable.set(acc, k, res);
        },
        emptyObject);
}

function diff(data1, data2) {
    if(Immutable.isObject(data1) && Immutable.isObject(data2)) {
        return diffObjects(data1, data2);
    }
    if(data1 !== data2) {
        return data2;
    }
    return "data-diff:no-diff";
}

```

## Listing 9.21 Testing data diff with persistent data structures

```

var data1 = Immutable.fromJS({
    g: {
        c: 3
    },
    x: 2,
    y: {
        z: 1
    },
    w: [5]
});

var data2 = Immutable.fromJS({
    g: {
        c: 3
    },
    x: 2,
    y: {
        z: 2
    },
    w: [4]
});

Immutable.isEqual(diff(data1, data2),
    Immutable.fromJS({
        "w": [
            4
        ],
        "y": {
            "z": 2
        }
    }));

```

**JOE:** What do you think of all this my friend?

**THEO:** I think that using persistent data collections with a library like Immutable.js is much easier than understanding the internals of persistent data structures. But I'm glad I know how it works under the hood.

After accompanying Joe to the office door, Theo meets Dave.

**DAVE:** What did Joe teach you today?

**THEO:** He took me to the university and taught me the foundations of persistent data structures, for dealing with immutability at scale.

**DAVE:** What's wrong with the structural sharing that I implemented a couple of months ago?

**THEO:** When the number of elements in the collection is big enough, naive structural sharing has performance issues.

**DAVE:** I see. Could you tell me more about that?

**THEO:** I'd love to but my brain won't function properly after this interesting but exhausting day. But we'll do it soon.

**DAVE:** No worries. Have a nice evening Theo.

**THEO:** You too, Dave.

## 9.6 Summary

- It is possible to manually ensure that our data is not mutated but it is cumbersome.
- At scale, naive structural sharing causes a performance hit both in terms of memory and computation.
- Naive structural sharing doesn't prevent data structures from being accidentally mutated.
- Immutable collections are not the same as persistent data structures.
- Immutable collections don't provide an efficient way to create new versions of the collections.
- Persistent data structures protect data from mutation.
- Persistent data structures provide an efficient way to create new versions of the collections.
- Persistent data structures always preserve the previous version of themselves when they are modified.
- Persistent data structures represent data internally in such a way that structural sharing scales well both in terms of memory and computation.
- When data is immutable, it is safe to share it.
- Internally, persistent use a branching factor of 32.
- In practice, manipulation of persistent data structures is efficient even for collections with 10 billion entries!
- Due to modern architecture consideration, the performance of updating a persistent list is dominated much more by the depth of the tree than by the number of nodes at each level of the tree.
- Persistent lists can be manipulated in near constant time.
- In most languages, there are third party libraries that provide an implementation of persistent data structures.
- Paguro collections implement the read-only parts of Java collection interfaces.
- Paguro collections can be passed to any methods that expect to receive a Java collection without mutating it.

**Table 9.2 Persistent data structure libraries**

Language	Library
JavaScript	Immutable.js
Java	Paguro
C#	provided by the language
Python	Pyrsistent
Ruby	Hamster

# 10

## *Database operations*

### This chapter covers

- Fetching data from the database
- Storing data in the database
- Manipulating data fetched from the database

### 10.1 A cloud is a cloud

Traditionally in Object-Oriented Programming we use design patterns and complex layers of objects to structure access to the database. In Data-Oriented Programming, we prefer to represent data fetched from the database with generic data collections, namely lists of maps where fields in the maps correspond to database column values. As we'll see throughout the chapter, the fact that fields inside a map are accessible dynamically via their names, allows us to use the same generic code for different data entities.

In Data-Oriented Programming, data from the database is represented with generic data collections. This leads to:

1. Reduced system complexity
2. Increased genericity

**TIP**

The best way to manipulate data is to represent data as data.

In this chapter we illustrate the application of Data-Oriented principles when accessing data from a relational database. Basic knowledge of relational database and SQL query syntax (like `SELECT`, `, AS`, `WHERE` and `INNER JOIN`) is assumed. This approach can be easily adapted to NoSQL

databases.

Applications that run on the server usually store data in a database. In DOP, we represent data retrieved from the database the same way we represent any other data in our application: with generic data collections.

## 10.2 Fetch data from the database

Theo and Joe go for a walk in a park near the office. They sit on a bench close to a beautiful lake and gaze at the clouds in the sky. After a couple of minutes of meditation, Joe breaks the silence and asks Theo: "What do you see?". Theo tells him that this cloud looks to him like a horse and that one looks like a car. On their way back to the office, Theo asks Joe for an explanation about the clouds. Joe answers with a mysterious smile on his lips: "A cloud is a cloud".

**THEO:** So far you've shown me how Data-Oriented Programming represents data that lives in the memory of the application. What about data that comes from the outside?

**JOE:** What do you mean by *outside*?

**THEO:** Data that comes from the database.

**JOE:** I'll return the question to you: How do you think that in DOP we should represent data that comes from the database?

**THEO:** As generic data collections, I guess.

**JOE:** Exactly! In Data-Oriented Programming, we always represent data with generic data collections.

**THEO:** Does that mean that we can manipulate data from the database with the same flexibility as we manipulate in-memory data?

**JOE:** Definitely.

**TIP**

In Data-Oriented Programming, we represent data from the database with generic data collections and we manipulate it with generic functions.

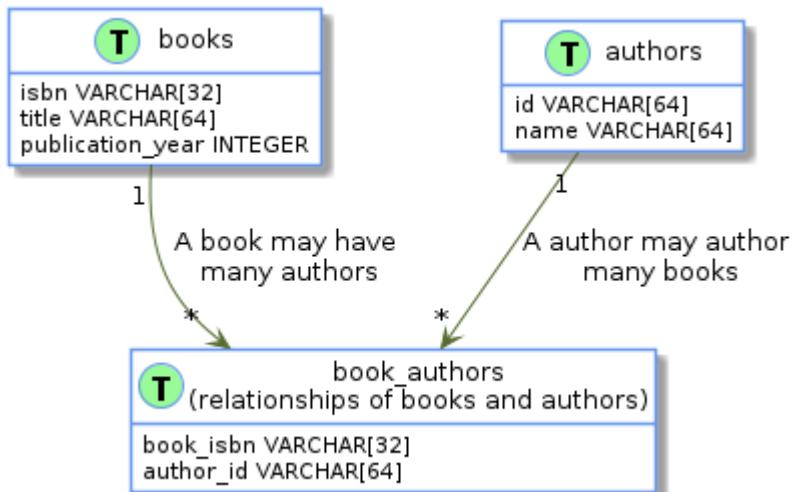
**THEO:** Could you show me how to retrieve book search results when the catalog data is stored in a SQL database?

**JOE:** I'll show you in a moment. First, tell me how you would design the tables that store catalog data?

**THEO:** Do you mean the exact table schemas with the information about primary keys and

nullability of each and every column?

**JOE:** No. I only need a rough overview of the tables, their columns, and the relationships between the tables.



**Figure 10.1** The database model for books and authors

**THEO:** I'd have a books table with 3 columns: title, isbn and publication\_year. And an authors table with two columns: for id and name.

**Table 10.1** The books table filled with two books

title	isbn	publication_year
The Power of Habit	978-0812981605	2012
7 Habits of Highly Effective People	978-1982137274	1989

**Table 10.2** The authors table filled with three authors

id	name
sean-covey	Sean Covey
stephen-covey	Stephen Covey
charles-duhigg	Charles Duhigg

**JOE:** And what about the connection between books and authors?

**THEO:** Let's see: A book could be written by multiple authors, and an author could write multiple books. Therefore, I need a many-to-many book\_authors table that connects authors and books, with two columns: book\_isbn and author\_id.

**Table 10.3** The book\_authors table with rows connecting books with their authors

book_isbn	author_id
978-1982137274	sean-covey
978-1982137274	stephen-covey
978-0812981605	charles-duhigg

**JOE:** Let's start with the simplest case. We're going to write code that searches for books matching a title and returns basic information about the books: title, isbn, and publication year.

**THEO:** What about the book authors?

**JOE:** We'll deal with that later, as it's a bit more complicated. Can you write a SQL query for retrieving books that contain *habit* in their title?

**THEO:** Sure.

It's quite easy for Theo. [10.1](#) shows the SQL query that he wrote.

#### Listing 10.1 SQL query to retrieve books whose title contain habit

```
SELECT
    title,
    isbn,
    publication_year
FROM
    books
WHERE title LIKE '%habit%';
```

The results are displayed in [10.4](#).

**Table 10.4** Results of the SQL query that retrieves books whose title contain habit

title	isbn	publication_year
The Power of Habit	978-0812981605	2012
7 Habits of Highly Effective People	978-1982137274	1989

**JOE:** How would you describe these results as a data collection?

**THEO:** I would say it's a list of maps.

#### TIP

In DOP, accessing data from a NoSQL database is similar to the way we're accessing data from a relational database.

**JOE:** Right. Can you write down the search results as a list of maps?

**THEO:** It doesn't sound too complicated.

Theo creates the list of maps in [10.2](#).

### **Listing 10.2 Search results as a list of maps**

```
[  
  {  
    "title": "7 Habits of Highly Effective People",  
    "isbn": "978-1982137274",  
    "publication_year": 1989  
  },  
  {  
    "title": "The Power of Habit",  
    "isbn": "978-0812981605",  
    "publication_year": 2012  
  }  
]
```

**JOE:** How about the JSON schema of the search results?

**THEO:** It shouldn't be too difficult if you allow me to take a look at the JSON schema cheatsheet you kindly offered me the other day.

**JOE:** Of course. The purpose of a gift is to be used by the one who received it.

After a few minutes, with the help of the cheatsheet in [10.3](#) , Theo comes up with a JSON schema for the search results in [10.4](#)

### **Listing 10.3 JSON schema cheatsheet**

```
{  
  "type": "array",  
  "items": {  
    "type": "object",  
    "properties": {  
      "myNumber": {"type": "number"},  
      "myString": {"type": "string"},  
      "myEnum": {"enum": ["myVal", "yourVal"]},  
      "myBool": {"type": "boolean"}  
    },  
    "required": ["myNumber", "myString"],  
    "additionalProperties": false  
  }  
}
```

### **Listing 10.4 The JSON schema for search results from the database**

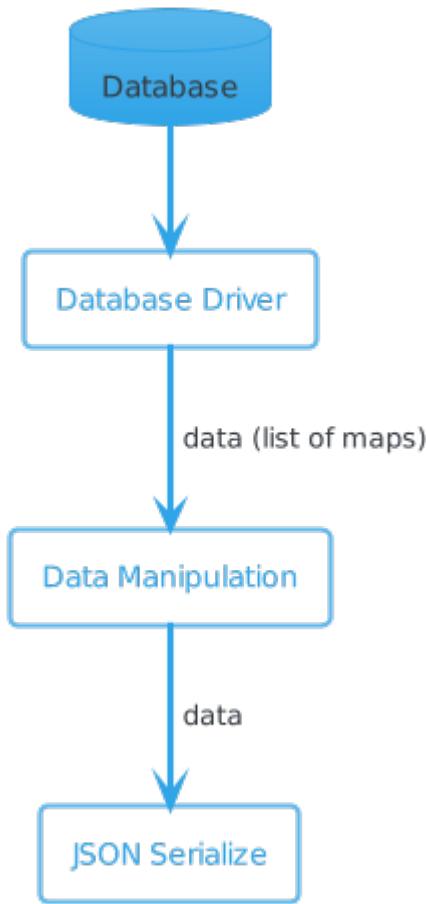
```
var dbSearchResultSchema = {  
  "type": "array",  
  "items": {  
    "type": "object",  
    "required": ["title", "isbn", "publication_year"],  
    "properties": {  
      "title": {"type": "string"},  
      "isbn": {"type": "string"},  
      "publication_year": {"type": "integer"}  
    }  
  }  
};
```

**JOE:** Excellent. Now I'm going to show you how to implement `searchBooks` in a way that fetches data from the database and returns a JSON string with the results. The cool thing is that we're only going to use generic data collections - from the DB layer down to the JSON serialization.

**THEO:** Will it be similar to the implementation of `searchBooks` that we wrote when you taught me the basis of DOP?

**JOE:** Absolutely. The only difference is that then the state of the system was stored locally. We queried it with a function like `_.filter`. Now, we use SQL queries to fetch the state from the Database. But in terms of data representation and manipulation it's exactly the same. The data flow looks like this:

Joe goes to the whiteboard and sketches out [10.2](#).



**Figure 10.2 Data flow for serving a request that fetches data from the database**

**JOE:** The Data manipulation step is implemented via generic functions that manipulate data collections. As our examples get more elaborate, I think you'll see the benefits of being able to manipulate data collections with generic functions.

**THEO:** Sounds intriguing...

**TIP**

**Most of our code is about generic functions that manipulate data collections.**

**JOE:** The core of it is the ability to communicate with the database through a driver that returns a list of maps. In JavaScript, you could use a SQL driver like node-postgres (<https://node-postgres.com>).

**THEO:** And in Java?

**JOE:** In Java, you could use JDBC (Java database connectivity) in addition to a small utility function like the one in [10.5](#) that converts a JDBC result set into a list of maps.

Joe writes a utility function as in [10.5](#)

**Listing 10.5 Converting a JDBC result set into a list of hash maps**

```
List<Map<String, Object>> convertJDBCResultSetToListOfMaps(ResultSet rs) {
    List<Map<String, Object>> listOfMaps = new ArrayList<Map<String, Object>>();
    ResultSetMetaData meta = rs.getMetaData();
    while (rs.next()) {
        Map map = new HashMap();
        for (int i = 1; i <= meta.getColumnCount(); i++) {
            String key = meta.getColumnLabel(i);
            Object value = rs.getObject(i);
            map.put(key, value);
        }
        listOfMaps.add(map);
    }
    return listOfMaps;
}
```

**TIP**

**Converting a JDBC result set into a list of hash maps is quite straightforward.**

**THEO:** I expected it to be much more complicated to convert a JDBC result set into a list of hash maps.

**JOE:** It's straightforward because, in a sense, JDBC is data-oriented!

**THEO:** What about the field types?

**JOE:** When we convert a JDBC result set into a list of maps, each value is considered an `Object`

**THEO:** That's annoying because it means that in order to access the value, we need to cast it to its type.

**JOE:** Yes and no. Look at our book search use case. We pass all the values along without really looking at their type. The concrete value type only matters when we serialize the result into

JSON. And that's handled by the JSON serialization library. It's called *late binding*.

**NOTE**

**With late binding, we defer dealing with data types as long as possible.**

**THEO:** Does that mean in my application that I'm allowed to manipulate data without dealing with concrete types?

**TIP**

**In Data-Oriented Programming, flexibility is increased as many parts of the system are free to manipulate data without dealing with concrete types.**

**JOE:** Exactly. You'll see late binding in action in a moment. That's one of the greatest benefits of Data-Oriented Programming!

**THEO:** Interesting...

**JOE:** One last thing before I show you the code for retrieving search results from the database: In order to make it easier to read, I'm going to write JavaScript code as if JavaScript were dealing with I/O is a synchronous way.

**THEO:** What do you mean?

**JOE:** In JavaScript, an I/O operation like sending a query to the database is done asynchronously. In real-life, it means using either callback functions or using `async` and `await` keywords.

**THEO:** Oh yeah. That's because JavaScript is single-threaded.

**WARNING**

**For sake of simplicity, the JavaScript snippets of this chapter are written as if JavaScript were dealing with I/O in a synchronous way. In real-life JavaScript, we need to use `async` and `await` around I/O calls.**

**JOE:** Indeed. So I'll be writing the code that communicates with the database as though JavaScript were dealing with I/O synchronously. Here's an example.

Joe writes the code in [10.6](#).

### Listing 10.6 Searching books in the database and returning the results in JSON

```

var dbClient;    ①
var ajv = new Ajv({allErrors: true});  ②

var title = "habit";
var matchingBooksQuery = `SELECT title, isbn
                          FROM books
                          WHERE title LIKE '%$1%'`; ③
var books = dbClient.query(matchingBooksQuery, [title]);  ④
if(!ajv.validate(dbSearchResultSchema, books)) {
  var errors = ajv.errorsText(ajv.errors);
  throw "Internal error: Unexpected result from the database: " + errors;
}

JSON.stringify(books);

```

- ① dbClient holds the DB connection.
- ② Ajv (a JSON schema validation library) is initialized with allErrors: true in order to catch all the data validation errors
- ③ We use a parameterized SQL query as a security best practice.
- ④ We pass the parameters to the SQL query as a list of values (in our case a list with a single value)

**THEO:** In a dynamically-typed language like JavaScript, I understand that the types of the values in the list of maps returned by dbClient.query are determined at run time. But how does it work in a statically-typed language like Java: What are the types of the data fields in books?

**JOE:** The function convertJDBCResultSetToListOfMaps we created earlier (see [10.5](#)) returns a list of Map<String, Object>. But JSON serialization libraries like gson (<https://github.com/google/gson>) know how to detect at run time the concrete type of the values in a map and serialize the values according to their type.

**THEO:** What do you mean by *serializing a value according to its type*?

**JOE:** For instance, the value of the field publication\_year is a number, therefore it is not wrapped with quotes. However, the value of the field title is a string, therefore is is wrapped with quotes.

**THEO:** Nice! Now, I understand what you mean by *late binding*.

**JOE:** Cool! Now, let me show you how we store data in the database.

## 10.3 Store data in the database

In the previous section, we have seen how to retrieve data from the database as a list of maps. Now, we are going to see how to store data in the database when data is represented with a map.

**THEO:** I guess that storing data in the database is quite similar to fetching data from the database.

**JOE:** It's similar in the sense that we deal only with generic data collections. Can you write down a parameterized SQL query that inserts a row with user info (only email and encrypted\_password)?

**THEO:** Here you go.

Theo writes a few lines of SQL as in [10.7](#).

### Listing 10.7 SQL statement to add a member

```
INSERT
INTO members
    (email, encrypted_password)
VALUES ($1, $2)
```

**JOE:** Great. And here's how to integrate your SQL query in our application code.

Joe codes up [10.8](#)

### Listing 10.8 Adding a member from inside the application

```
var addMemberQuery = "INSERT INTO members (email, password) VALUES ($1, $2)";
dbClient.query(addMemberQuery,
    [_.get(member, "email"), ①
     _.get(member, "encryptedPassword")]);
```

- ① We pass the two parameters to the SQL query as an array

**THEO:** Your code is very clear. But something still bothers me.

**JOE:** What is it?

**THEO:** I find it cumbersome that you use `_.get(user, "email")` instead of `user.email` like I would if the data were represented with a class.

**JOE:** In JavaScript, you are allowed to use the dot notation `user.email` instead of `_.get(user, "email")`.

**THEO:** Then, why don't you use the dot notation?

**JOE:** Because I want to show you how you could apply DOP principles even in languages where the dot notation is not available for hash maps, like Java.

**WARNING** In the book, we avoid using the JavaScript dot notation to access a field in a hash map in order to illustrate how to apply DOP in languages that don't support dot notation on hash maps.

**THEO:** That's exactly my point: I find it cumbersome in a language like Java to use `_.get(user, "email")` instead of `user.email` like I would if the data were represented with a class.

**JOE:** On one hand, it's cumbersome. On the other hand, representing data with a hash map instead of a static class allows you to access fields in a flexible way.

**THEO:** I know. You've told me so many times! But I can't get used to it.

**JOE:** Let me give you another example of the benefits of the flexible access to data fields, in the context of adding a member to the database. You said that writing `[_.get(member, "email"), _.get(member, "encryptedPassword")]` was less convenient than writing `[member.email, member.encryptedPassword]`. Right?

**THEO:** Absolutely!

**JOE:** Let me show you how to write the same code in a more succinct way, leveraging a function from Lodash called `_.at`.

**THEO:** What does this `_.at` function do?

**JOE:** It receives a map `m` and a list `keyList` and returns a list made of the values in `m` associated with the keys in `keyList`.

**THEO:** How about an example?

**JOE:** Sure. We create a list made of the fields `email` and `encryptedPassword` of a member.

Joe types for a bit, and shows the code in [10.9](#) to Theo.

### Listing 10.9 Creating a list made of some values in a map with `_.at`

```
var member = {
  "email": "samantha@gmail.com",
  "encryptedPassword": "c2VjcmV0",
  "isBlocked": false
};

_.at(member,
  ["email", "encryptedPassword"]);

// [ "samantha@gmail.com",
//   "c2VjcmV0" ]
```

**THEO:** Do the values in the results appear in the same order as the keys in `keyList`?

**JOE:** Yes!

**THEO:** That's cool!

**TIP**

Accessing a field in a hash map is more flexible than accessing a member in an object instantiated from a class.

**JOE:** And here's the code for adding a member using `_.at`.

Joe types quickly and shows Theo the code in [10.10](#).

#### **Listing 10.10 Adding a member leveraging `_.at` to retrieve multiple values from a map**

```
class CatalogDB {
    static addMember(member) {
        var addMemberQuery = `INSERT
            INTO members
            (email, encrypted_password)
            VALUES ($1, $2)`;
        dbClient.query(addMemberQuery,
            _.at(member, ["email",
                "encryptedPassword"]));
    }
}
```

**THEO:** I could see how the `_at` function becomes really beneficial when we need to pass a bigger number of field values.

**JOE:** Yes. And I'll be showing you more examples that leverage the flexible data access that we have in DOP.

## 10.4 Simple data manipulation

Quite often, in a production application, we need to reshape data fetched from the database. The simplest case is when we need to rename the column names from the database to names that are more appropriate for our application.

**JOE:** Did you notice that column names in our database follow the snake case convention?

**THEO:** No.

**JOE:** For instance, the column for the publication year of a book is called `publication_year`.

**THEO:** Go on..

**JOE:** Inside JSON, I like to use Pascal case, like `publicationYear`.

**THEO:** And I'd prefer to have `bookTitle` instead of `title`.

**JOE:** So we're both unhappy with the JSON string that `searchBooks` returns if we pass the data retrieved from the database as-is.

**THEO:** Indeed.

**JOE:** How would you fix it?

**THEO:** I would modify the SQL query so that it renames the columns in the results. Here, let me show you.

Theo writes some SQL as in [10.11](#).

#### Listing 10.11 Rename columns inside the SQL query

```
SELECT
    title AS bookTitle,
    isbn,
    publication_year AS publicationYear
FROM
    books
WHERE title LIKE '%habit%';
```

**JOE:** That would work. But it seems a bit weird to modify a SQL query so that it fits the naming convention of the application.

**THEO:** Yeah, I agree. And I imagine a database like MongoDB doesn't make it easy to rename the fields inside a query.

**JOE:** Yeah. Sometimes, it makes more sense to deal with field names in application code. How would you handle that?

**THEO:** Well, in that case, for every map returned by the db query, I'd use a function to modify the field names.

**JOE:** Could you show me what the code would look like?

**THEO:** Sure.

Theo types for a bit, and shows Joe the code in [10.12](#).

### Listing 10.12 Renaming specific keys in a list of maps

```

function renameBookInfoKeys(bookInfo) {
  return {
    "bookTitle": _.get(bookInfo, "title"),
    "isbn": _.get(bookInfo, "isbn"),
    "publicationYear": _.get(bookInfo, "publication_year")
  };
}

var bookResults = [
  {
    "title": "7 Habits of Highly Effective People",
    "isbn": "978-1982137274",
    "publication_year": 1989
  },
  {
    "title": "The Power of Habit",
    "isbn": "978-0812981605",
    "publication_year": 2012
  }
];

_.map(bookResults, renameBookInfoKeys);

```

**JOE:** And you'd write a similar piece of code for every query that you fetched from the database?

**THEO:** What do you mean?

**JOE:** Suppose, you want to rename the field names returned by a query that returns the books a user has lent.

**THEO:** Yes. I'd write a similar piece of code for each case.

**JOE:** In DOP, we leverage the fact that a field name is just a string and write a generic function called `renameResultKeys` that works on every list of maps.

**THEO:** Wow! And how does `renameResultKeys` know what fields to rename?

**JOE:** You pass the mapping between old and new names as a map.

**TIP**

**In DOP, field names are just strings. It allows us to write generic functions to manipulate list of maps representing data fetched from the database.**

**THEO:** Could you show me an example?

**JOE:** Sure. A map like this could be passed to `renameResultKeys` to rename the fields in book search results.

Joe codes up the example in [10.14](#) and shows it to Theo.

### Listing 10.13 Renaming fields in SQL results

```
renameResultKeys(bookResults, {
  "title": "bookTitle",
  "publication_year": "publicationYear"
});
```

**THEO:** What happened to the field that stores the `isbn`?

**JOE:** When a field is not mentioned, `renameResultKeys` leaves it as is.

**THEO:** Awesome! Could you show me the implementation of `renameResultKeys`?

**JOE:** Sure. It's only about `map` and `reduce`.

Joe writes the code for `renameResultKeys` as in [10.14](#) shows it to Theo.

### Listing 10.14 Renaming the keys in SQL results

```
function renameKeys(map, keyMap) {
  return _.reduce(keyMap, {
    function(res, newKey, oldKey) {
      var value = _.get(map, oldKey);
      var resWithNewKey = _.set(res, newKey, value);
      var resWithoutOldKey = _.omit(resWithNewKey, oldKey);
      return resWithoutOldKey;
    },
    map
  });
}

function renameResultKeys(results, keyMap) {
  return _.map(results, function(result) {
    return renameKeys(result, keyMap);
  });
}
```

**THEO:** That codes isn't exactly easy to understand!

**JOE:** Don't worry. The more you write data manipulation functions with `map`, `filter` and `reduce`, the more you get used to it.

**THEO:** I hope so.

**JOE:** What's really important for now, is that you understand what makes it possible in DOP to write a function like `renameResultKeys`.

**THEO:** I would say, it's because fields are accessible dynamically with strings.

**JOE:** Exactly. You could say that fields are first-class citizens.

**TIP**

In Data-Oriented Programming, fields are first-class citizens.

**THEO:** How would you write unit tests for a data manipulation function like `renameResultKeys`?

**JOE:** It's very similar to the unit tests we wrote earlier. You generate input and expected results and you make sure that the actual results equal the expected results. Hang on.

Joe codes for several mintues, arriving at [10.15](#) and shows Theo.

### Listing 10.15 Unit test for `renameResultKeys`

```
var listOfMaps = [
  {
    "title": "7 Habits of Highly Effective People",
    "isbn": "978-1982137274",
    "publication_year": 1989
  },
  {
    "title": "The Power of Habit",
    "isbn": "978-0812981605",
    "publication_year": 2012
  }
];

var expectedResults = [
  {
    "bookTitle": "7 Habits of Highly Effective People",
    "isbn": "978-1982137274",
    "publicationYear": 1989
  },
  {
    "bookTitle": "The Power of Habit",
    "isbn": "978-0812981605",
    "publicationYear": 2012
  }
];

var results = renameResultKeys(listOfMaps,
  {"title": "bookTitle",
   "publication_year": "publicationYear"});

_.isEqual(expectedResults, results);
```

**THEO:** Nice!

**JOE:** Do you see why you're free to use `renameResultKeys` with the results of any SQL query?

**THEO:** Yes. Because, the code of `renameResultKeys` is decoupled from the internal details of the representation of data it operates on.

**JOE:** Exactly! Suppose a SQL query returns user info in a table. How would you use `renameResultKeys` to rename `email` to `userEmail`? Assume the table looks like this.

Joe shows Theo the table in [10.5](#).

**Table 10.5 Results of a SQL query that returns email and encrypted\_password of some users**

email	encrypted_password
jennie@gmail.com	secret-pass
franck@hotmail.com	my-secret

**THEO:** That's easy!

Theo writes code as in [10.16.](#)

#### **Listing 10.16 Renaming email to userEmail**

```
var listOfMaps = [
  {
    "email": "jennie@gmail.com",
    "encryptedPassword": "secret-pass"
  },
  {
    "email": "franck@hotmail.com",
    "encryptedPassword": "my-secret"
  }
];

renameResultKeys(listOfMaps,
  {"email": "userEmail"});
```

**JOE:** Excellent! I think you're ready to move on to advanced data manipulation.

## **10.5 Advanced data manipulation**

In some cases we need to change the structure of the rows returned by a SQL query, for instance aggregating fields from different rows into a single map. This could be done at the level of the SQL query, leveraging advanced features like JSON aggregation in PostgreSQL. However, sometimes it makes more sense to reshape the data inside the application as it keeps the SQL queries simple.

Like with the simple data manipulation scenario of the previous section, once we have written code that implements some data manipulation, we're free to use the same code for similar use cases, even if they involve data entities of different types.

**THEO:** What kind of advanced data manipulation do you have in mind?

**JOE:** You'll see in a minute. But first a SQL task for you: Write a SQL query that returns books that contain habit in their title, including author names.

**THEO:** Let me give it a try.

After some trial and errors, Theo is able to nail it down and write a SQL query that joins the

three tables books, book\_authors and authors. The result of his endeavor is shown on [10.17](#).

**Listing 10.17 SQL query to retrieve books whose title contain habit, including author names**

```
SELECT
    title,
    isbn,
    authors.name AS author_name
FROM
    books
    INNER JOIN
    book_authors
        ON books.isbn = book_authors.book_isbn
    INNER JOIN
    authors
        ON book_authors.author_id = authors.id
WHERE books.title LIKE '%habit%';
```

**JOE:** How many rows are in the results?

**Table 10.6 Results of the SQL query that retrieves books whose title contain habit, including author names**

title	isbn	author_name
7 Habits of Highly Effective People	978-1982137274	Sean Covey
7 Habits of Highly Effective People	978-1982137274	Stephen Covey
The Power of Habit	978-0812981605	Charles Duhigg

**THEO:** Three rows.

**JOE:** And how many books?

**THEO:** Two books.

**JOE:** Could you show me the results as a list of maps?

**THEO:** Sure.

Theo shows Joe [10.18](#).

**Listing 10.18 A list of maps with the results of the SQL query that retrieves books whose title contain habit, including author names**

```
[
  {
    "title": "7 Habits of Highly Effective People",
    "isbn": "978-1982137274",
    "publication_year": "Sean Covey"
  },
  {
    "title": "7 Habits of Highly Effective People",
    "isbn": "978-1982137274",
    "author_name": "Stephen Covey"
  },
  {
    "title": "The Power of Habit",
    "isbn": "978-0812981605",
    "author_name": "Charles Duhigg"
  }
]
```

**JOE:** And what does the list of maps that we need to return look like?

**THEO:** It's a list with two maps, where the author names are aggregated in a list.

Theo shows Joe [10.19](#).

**Listing 10.19 A list of maps with the results, where the author names are aggregated in a list**

```
[
  {
    "isbn": "978-1982137274",
    "title": "7 Habits of Highly Effective People",
    "authorNames": [
      "Sean Covey",
      "Stephen Covey"
    ]
  },
  {
    "isbn": "978-0812981605",
    "title": "The Power of Habit",
    "authorNames": ["Charles Duhigg"]
  }
]
```

**JOE:** Perfect! Here's an example of an advanced data manipulation task: Converting the list of maps as returned from the database to a list of maps where the author names are aggregated.

**THEO:** Hmm... It sounds tough.

**JOE:** Let me break the task in two steps. First, we group together rows that belong to the same book (with the same isbn). Then, in each group we aggregate author names in a list. Hold on, I'll diagram it as a data processing pipeline.

Joe goes to the whiteboard and draws the diagram in [10.3](#).

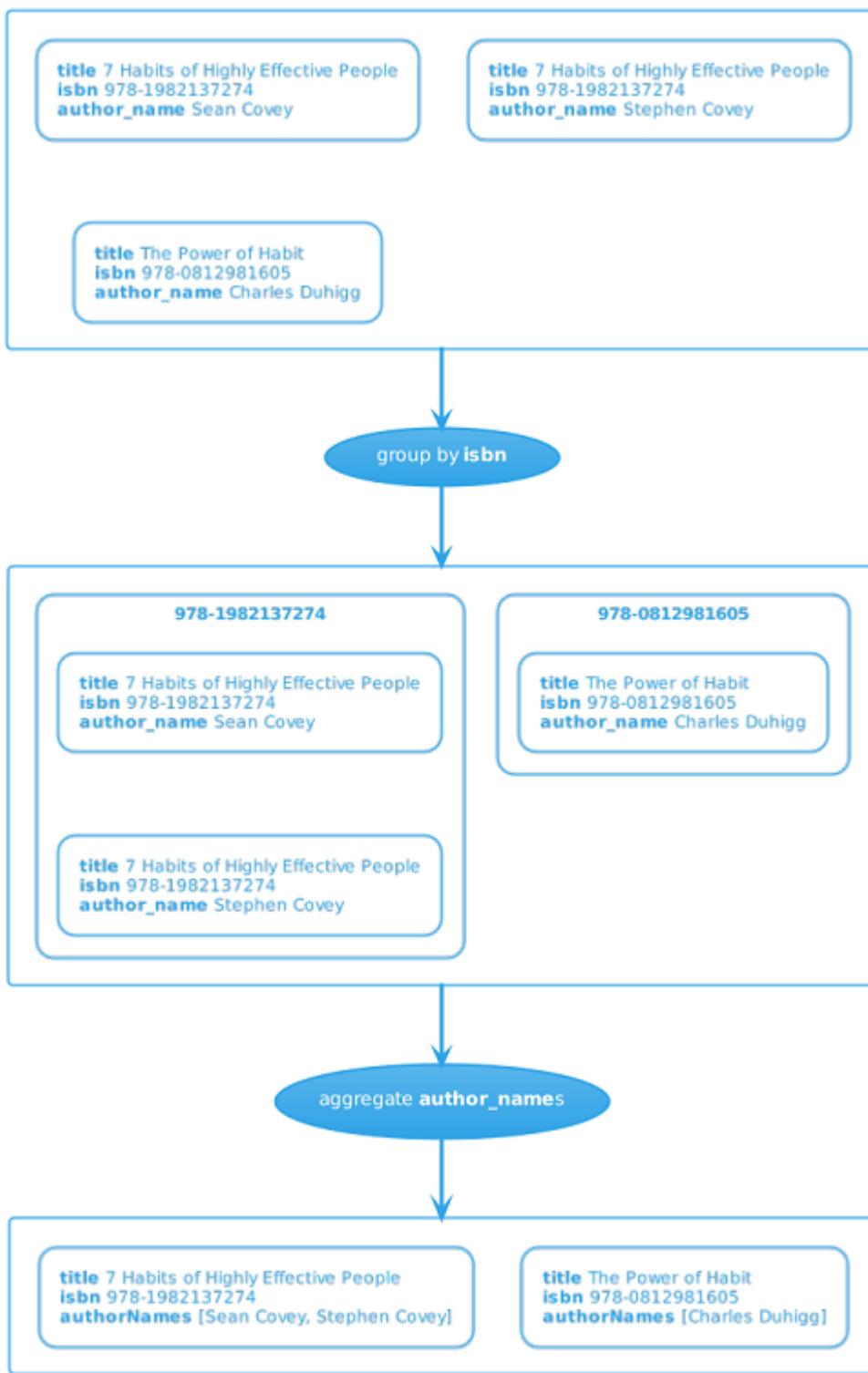


Figure 10.3 Data pipeline for aggregating author names

**JOE:** Does it makes sense to you?

**THEO:** Yes, the data pipeline makes sense, but I have no idea how to write code that implements it!

**JOE:** Let me guide you step by step. Let's start by grouping together books with the same isbn, using `_.groupBy`. Here what it would look like:

Joes writes the code in [10.20.](#)

### Listing 10.20 Grouping rows by isbn

```
var sqlRows = [
  {
    "title": "7 Habits of Highly Effective People",
    "isbn": "978-1982137274",
    "author_name": "Sean Covey"
  },
  {
    "title": "7 Habits of Highly Effective People",
    "isbn": "978-1982137274",
    "author_name": "Stephen Covey"
  },
  {
    "title": "The Power of Habit",
    "isbn": "978-0812981605",
    "author_name": "Charles Duhigg"
  }
];

_.groupBy(sqlRows, "isbn");
```

**THEO:** What does `rowsByIsbn` look like?

**JOE:** It's a map where the keys are `isbn` and the values are lists of rows.

Joe show Theo the results as in [10.21.](#)

### Listing 10.21 Rows grouped by isbn

```
{
  "978-0812981605": [
    {
      "author_name": "Charles Duhigg",
      "isbn": "978-0812981605",
      "title": "The Power of Habit"
    }
  ],
  "978-1982137274": [
    {
      "author_name": "Sean Covey",
      "isbn": "978-1982137274",
      "title": "7 Habits of Highly Effective People"
    },
    {
      "author_name": "Stephen Covey",
      "isbn": "978-1982137274",
      "title": "7 Habits of Highly Effective People"
    }
  ]
}
```

**THEO:** What's the next step?

**JOE:** Now, we need to take each list of rows in `rowsByIsbn` and aggregate author names.

**THEO:** How do we do that?

**JOE:** Let's do it on the list of 2 rows for "7 habits of highly effective people". It looks like this.

Joe shows Theo [10.22](#).

**JOE:** First, we take the author names from all the rows. Then we take the first row as a basis for the book info and we add a field `authorNames` and remove the field `author_name`.

### Listing 10.22 Aggregating author names

```
var rows7Habits = [
  {
    "author_name": "Sean Covey",
    "isbn": "978-1982137274",
    "title": "7 Habits of Highly Effective People"
  },
  {
    "author_name": "Stephen Covey",
    "isbn": "978-1982137274",
    "title": "7 Habits of Highly Effective People"
  }
];

var authorNames = _.map(rows7Habits, "author_name"); ①
var firstRow = _.nth(rows7Habits, 0);
var bookInfoWithAuthorNames = _.set(firstRow, "authorNames", authorNames);
_.omit(bookInfoWithAuthorNames, "author_name"); ②
```

- ① Take the author names from all rows
- ② Remove the field `author_name`

**THEO:** Could we make a function of it?

**JOE:** That's exactly what I was going to suggest.

**THEO:** I'll call the function `aggregateField` and it will receive 3 arguments: the rows, the name of the field to aggregate and the name of the field that holds the aggregation.

Theo gets closer to his laptop and after a couple of minutes, his screen displays the contents of [10.23](#).

### Listing 10.23 Aggregating an arbitrary field

```
function aggregateField(rows, fieldName, aggregateFieldName) {
  var aggregatedValues = _.map(rows, fieldName);
  var firstRow = _.nth(rows, 0);
  var firstRowWithAggregatedValues = _.set(firstRow,
                                             aggregateFieldName,
                                             aggregatedValues);
  return _.omit(firstRowWithAggregatedValues, fieldName);
}
```

**JOE:** Do you mind writing a test case to make sure your function works as expected?

**THEO:** With pleasure! Take a look.

Theo shows Joe his test code as in [10.24](#).

#### Listing 10.24 Test case for aggregateField

```
var expectedResults = {
  "isbn": "978-1982137274",
  "title": "7 Habits of Highly Effective People",
  "authorNames": [
    "Sean Covey",
    "Stephen Covey"
  ]
};

_.isEqual(expectedResults,
  aggregateField(rows7Habits,
    "author_name",
    "authorNames"));
```

**JOE:** Excellent! Now that we have a function that aggregates a field from a list of rows, we only need to map the function over the values of our `rowsByIsbn`.

Joe codes it up as shown in [10.25](#).

#### Listing 10.25 Aggregating author names in rowsByIsbn

```
var rowsByIsbn = _.groupBy(sqlRows, "isbn");
var groupedRows = _.values(rowsByIsbn);

_.map(rowsByIsbn, function(groupedRows) {
  return aggregateField(groupedRows, "author_name", "authorNames");
})
```

**THEO:** Why did you take the values of `rowsByIsbn`?

**JOE:** Because we don't really care about the keys in `rowsByIsbn`. We only care about the grouping of the rows in the values of the hash map.

**THEO:** Let me try to combine everything we've done and write a function that receives a list of rows and returns a list of book infos with the author names aggregated in a list.

**JOE:** Good luck my friend!

It's less complicated than it seemed. After a couple of trials and errors, Theo arrives at the code and the test case that appear in [10.26](#).

### Listing 10.26 Aggregating a field in a list of rows

```

function aggregateFields(rows, idFieldName, fieldName, aggregateFieldName) {
  var groupedRows = _.values(_.groupBy(rows, idFieldName));
  return _.map(groupedRows, function(groupedRows) {
    return aggregateField(groupedRows, fieldName, aggregateFieldName);
  });
}

var sqlRows = [
  {
    "title": "7 Habits of Highly Effective People",
    "isbn": "978-1982137274",
    "author_name": "Sean Covey"
  },
  {
    "title": "7 Habits of Highly Effective People",
    "isbn": "978-1982137274",
    "author_name": "Stephen Covey"
  },
  {
    "title": "The Power of Habit",
    "isbn": "978-0812981605",
    "author_name": "Charles Duhigg"
  }
];

var expectedResults =
[
  {
    "authorNames": [
      "Sean Covey",
      "Stephen Covey"
    ],
    "isbn": "978-1982137274",
    "title": "7 Habits of Highly Effective People"
  },
  {
    "authorNames": ["Charles Duhigg"],
    "isbn": "978-0812981605",
    "title": "The Power of Habit",
  }
];

_.isEqual(aggregateFields(sqlRows,
  "isbn",
  "author_name",
  "authorNames"),
  expectedResults);

```

**THEO:** I think I made it.

**JOE:** Congratulations! I'm proud of you.

Now Theo understands what Joe meant when he told him "A cloud is cloud" while they were walking from the park to the office. Instead of trapping data in the limits of our objects, DOP guides us to represent data as data.

## 10.6 Summary

- Inside our applications, we represent data fetched from the database, no matter if it is relational or non-relational, as a generic data structure.
- In the case of a relational database, data is represented as a list of maps.
- Representing data from the database as data reduces system complexity as we don't need design patterns or complex class hierarchies to do it.
- We are free to manipulate data from the database with generic functions, such as returning a list made of the values of some data fields, creating a version of a map omitting a data field, or grouping maps in a list according to the value of a data field.
- We use generic functions for data manipulation with great flexibility, leveraging the fact that inside a hash map we access the value of a field via its name, represented as a string.
- When we package our data manipulation logic into custom functions that receive field names as arguments, we are able to use those functions on different data entities.
- The best way to manipulate data is to represent data as data.
- We represent data from the database with generic data collections and we manipulate it with generic functions.
- Accessing data from a NoSQL database is done in a similar way to the approach presented in this chapter for accessing data from a relational database.
- With late binding, we care about data types as late as possible.
- Flexibility is increased as many parts of the system are free to manipulate data without dealing with concrete types.
- Accessing a field in a hash map is more flexible than accessing a member in an object instantiated from a class.
- In DOP, field names are just strings. It allows us to write generic functions to manipulate list of maps representing data fetched from the database.
- In DOP, fields are first-class citizens.
- Renaming keys in a list of maps or aggregating rows returned by a database query are implemented via generic functions.
- JDBC stands for Java database connectivity.
- Converting a JDBC result set into a list of maps is quite straightforward.

**Table 10.7 Lodash functions introduced in this chapter**

Function	Description
<code>at(map, [paths])</code>	Creates an array of values corresponding to <code>paths</code> of <code>map</code>
<code>omit(map, [paths])</code>	Creates a map composed of the fields of <code>map</code> not in <code>paths</code>
<code>nth(arr, n)</code>	Get the element at index <code>n</code> in <code>arr</code>
<code>groupBy(coll, f)</code>	Creates a map composed of keys generated from the results of running each element of <code>coll</code> through <code>f</code> . The corresponding value for each key is an array of elements responsible for generating the key

# 11

## *Web services*

### This chapter covers

- Representing a client request as a map
- Representing a server response as a map
- Passing data forward
- Combining data from different sources

## 11.1 A faithful messenger

The architecture of modern information systems is made of software components written in various programming languages that communicate over the wire by sending and receiving data represented in a language independent data exchange format, for instance JSON.

Data-Oriented programming applies the same principle to the communication between inner parts of a program. Inside a program, components communicate by sending and receiving data represented in a component independent format, namely immutable data collections.

In the context of a web service that fulfills a client request by fetching data from a database and other web services, representing data as with immutable data collections leads to the following benefits:

- Leverage generic data manipulation functions to manipulate data from multiple data sources
- Freedom to pass data forward with no additional complexity

## 11.2 Another feature request

After having delivered the database milestone on time, Theo calls Nancy to share the good news. Instead of celebrating Theo's success, Nancy asks him about the ETA for the next milestone: Book Information Enrichment with OpenLibraryBook. Theo tells her that he'll get back to her with an ETA by the end of the day.

When Theo arrives at the office, he tells Joe about the discussion with Nancy.

**THEO:** I just got a phone call from Nancy and she is stressed about the next milestone.

**JOE:** What's in the the next milestone?

**THEO:** Do you remember the Open Library Books API (<https://openlibrary.org/dev/docs/api/books>) I told you about a few weeks ago?

**JOE:** No.

**THEO:** It's a web service that provides detailed information about books.

**JOE:** Cool!

**THEO:** Nancy wants to enrich book search results. Instead of fetching book information from the database, we need to retrieve extended book information from the Open Library Books API.

**JOE:** What kind of information?

**THEO:** Everything! Number of pages, weight, physical format, topics etc...

**JOE:** What about the information from the database?

**THEO:** We don't need it any more, besides the information about the availability of the books.

**JOE:** Have you already looked at Open Library Books API?

**THEO:** It's a nightmare: for some books the information contains dozen of fields and for other books it has only 2 or 3 fields.

**JOE:** What's the problem then?

**THEO:** I have no idea how to represent data that is so sparse and unpredictable!

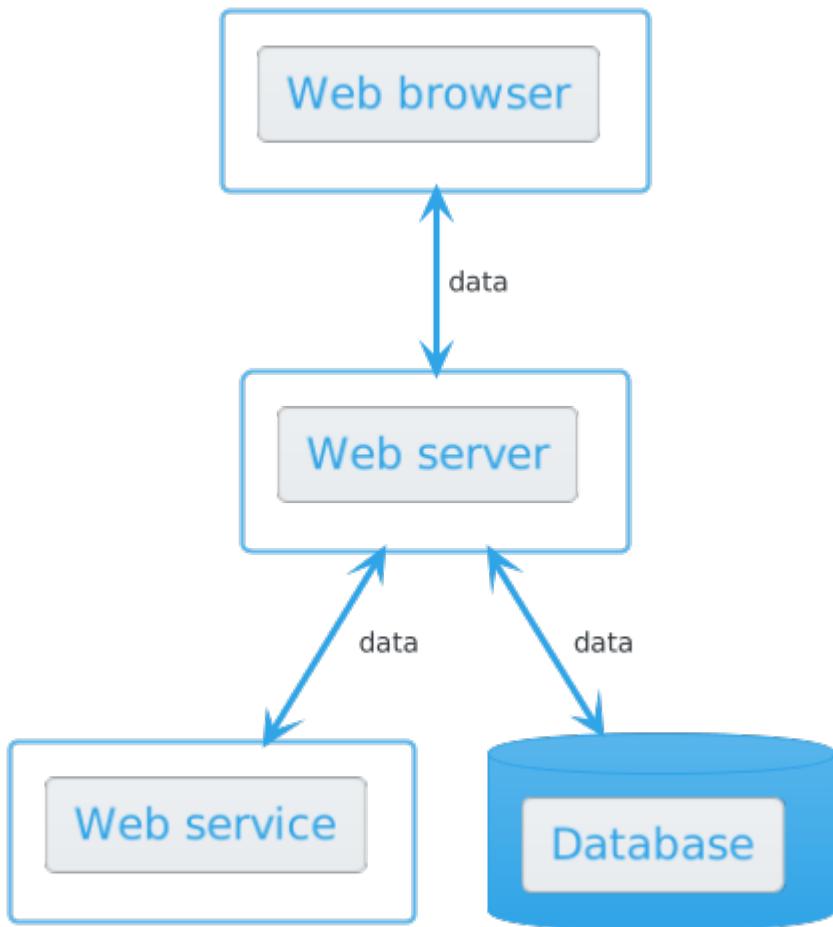
**JOE:** When we represent data as data, that's not an issue. Let's have a coffee and I'll show you.

## 11.3 Build the insides like the outsides

While Theo is drinking his Machiatto, Joe is drawing a diagram on a whiteboard, that looks like the diagram on [11.1](#).

**NOTE**

When a web browser sends a request to a web service, it's quite common that the web service itself sends requests to other web services in order to fulfill the web browser request. One popular data exchange format is JSON.



**Figure 11.1** The high level architecture of a modern information system

**JOE:** Before we dive into the details of the implementation of the book search result enrichment, let me give you a brief intro.

**THEO:** Sure.

Joe points to diagram in [11.1](#) on the whiteboard.

**JOE:** Does this look familiar to you?

**THEO:** Of course!

**JOE:** And can you show me, roughly, the steps in the data flow of a web service?

**THEO:** Sure.

Theo comes closer to the whiteboard and he writes down a list of steps near the architecture (see sidebar).

**SIDE BAR**

**The steps of the data flow inside a web service**

1. Receive a request from a client
2. Apply business logic to the request
3. Fetch data from external sources (e.g. database and other web services)
4. Apply business logic to the responses from external sources
5. Send response to the client

**JOE:** Excellent! Now comes an important insight about Data-Oriented programming.

**THEO:** I'm all ears.

**JOE:** We should build the insides of our systems like we build the outsides.

**THEO:** What do you mean?

**JOE:** How do components of a system communicate over the wire?

**THEO:** By sending data.

**JOE:** Does the data format depend on the programming language of the components?

**THEO:** No. Quite often it's JSON for which we have parsers in all programming languages.

**JOE:** What the idiom says is that inside our program the inner components of a program should communicate in a way that doesn't depend on the components.

**THEO:** I don't get that.

**JOE:** Let me explain why traditional Object-Oriented programming breaks this idiom: When data is represented with classes, the inner components of a program need to know the internals of the class definitions in order to communicate.

**THEO:** What do you mean?

**JOE:** In order to be able to access a member in a class, a component needs to "import" the class

definition.

**THEO:** How could it be different?

**JOE:** In Data-Oriented programming, as we have seen so far throughout the book, the inner components of a program communicate via generic data collections. It's similar to how components of a system communicate over the wire.

**TIP**

We should build the insides of our systems like we build the outsides.

**THEO:** Why is that so important?

**JOE:** From a design perspective, it's important because it means that the inner components of a program are loosely-coupled.

**THEO:** What do you mean by *loosely coupled*?

**JOE:** I mean that components need no knowledge about the internals of other components: the only knowledge required is the names of the fields.

**TIP**

In Data-Oriented programming, the inner components of a program are loosely-coupled.

**THEO:** And from an implementation perspective?

**JOE:** As you'll see in a moment, implementing the steps of the data flow that you just wrote on the whiteboard is easy.

Joe gestures toward the steps of the data flow inside a web service on the whiteboard as in the sidebar.

**JOE:** It comes down to expressing the business logic in terms of generic data manipulation functions. Here let me show you a diagram.

Joe steps up to the board and sketches out [11.2](#).

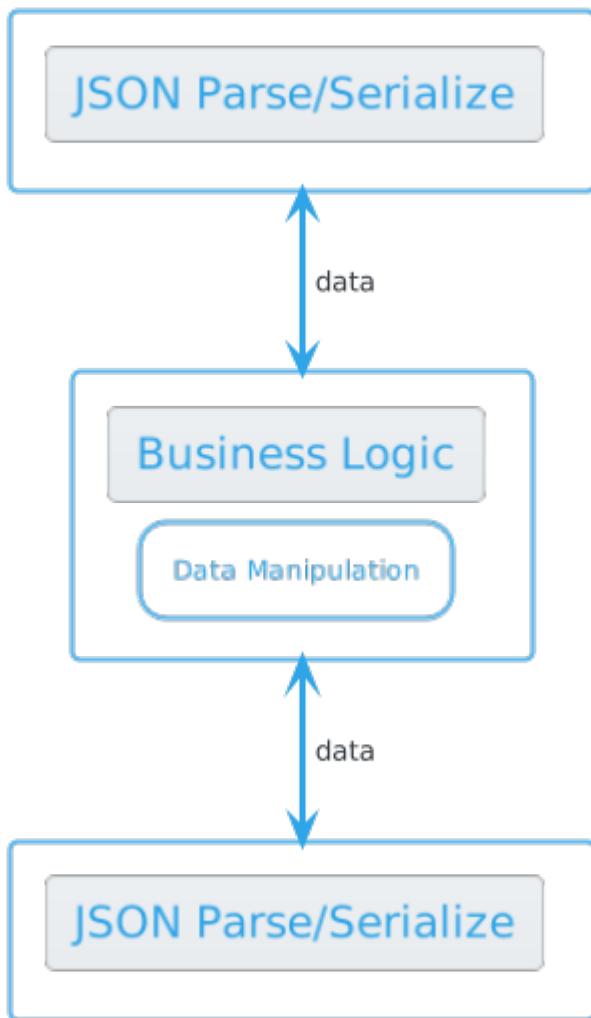
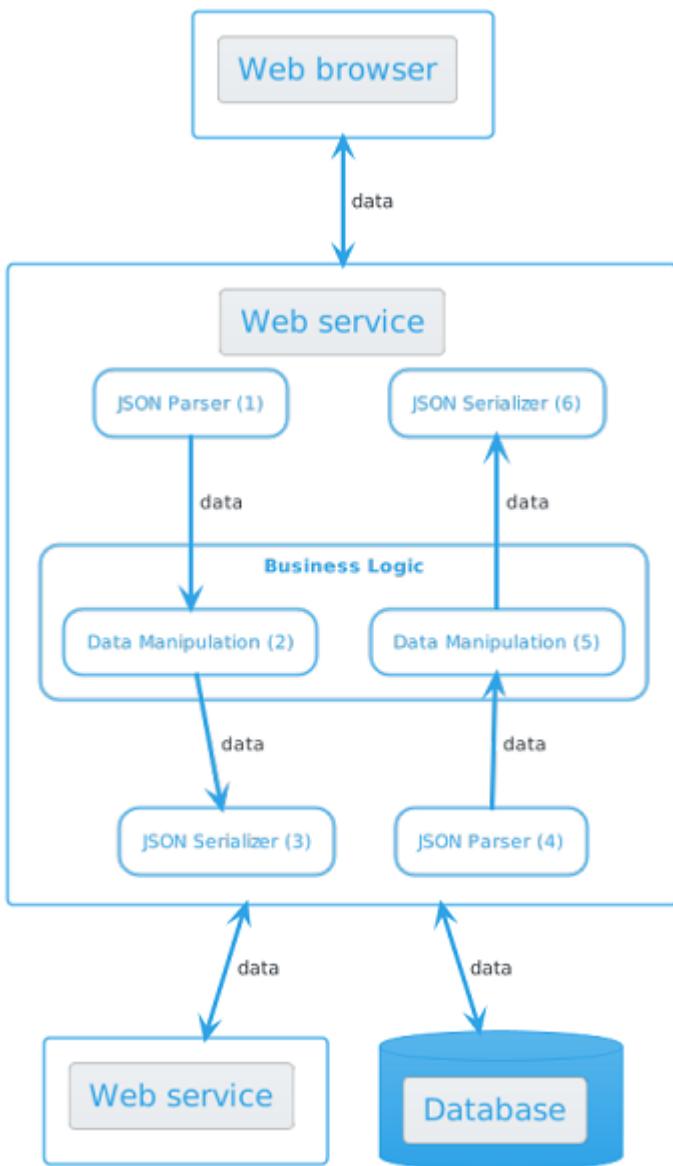


Figure 11.2 The internals of a Data-Oriented web service

Joe's cellphone rings. He excuses himself and steps outside to take the call.

Theo stands alone for a few minutes in front of the whiteboard, meditating about *building the insides of our systems like we build the outsides*. Without really noticing it, he takes a marker and starts drawing a new diagram, as in [11.3](#), that summarizes the insights that Joe just shared with him.



**Figure 11.3** Building the insides of our systems like we build the outsides. The inner components of a Web service communicate with data. As an example, here is a typical flow of a Web service handling a client request: (1) Parsing the client JSON request into data. (2) Manipulate data according to business logic. (3) Serialize data into a JSON requests to a database and another Web service. (4) Parsing JSON responses into data. (5) Manipulate data according to business logic. (6) Serialize data into a JSON response to the client.

## 11.4 Represent a client request as a map

After a few minutes Joe comes back. When he looks at Theo's drawing in [11.3](#), he seems satisfied.

**JOE:** Let's start from the beginning: parsing a client request. How do you usually receive the parameters of a client request?

**THEO:** It depends. The parameters could be sent as URL query parameters in a GET request or a JSON payload in the body of a POST request.

**JOE:** Let's suppose we receive a JSON payload inside a web request. Can you give me an example of a JSON payload for an advanced search request?

**THEO:** It would contain the text that the book title should match.

**JOE:** What about details about the fields to retrieve from the Open Library books API?

**THEO:** They won't be passed as part of the JSON payload since they're the same for all the search requests.

**JOE:** I could imagine a scenario where you want the client to decide what fields to retrieve. For instance a mobile client would prefer to retrieve only the most important fields and save network bandwidth.

**THEO:** Well, in that case I would have two different search endpoints: one for mobile and one for desktop.

**JOE:** What about situations where the client wants to display different pieces of information depending on the application screen. For instance, in an extended search result screen, we display all the fields. In a basic search result screen we display only the most important fields. Now you have 4 different use cases: desktop extended, desktop basic, mobile extended and mobile basic. Would you create 4 different endpoints?

**THEO:** OK. You convinced me: let's have a single search endpoint and let the client decide what fields to retrieve.

**JOE:** Could you show me an example of a JSON payload for a search request?

**THEO:** Sure.

Theo quickly writes the search request on the whiteboard, as in [11.1](#).

**Listing 11.1 Example of search request payload where the clients decide what fields to retrieve for each search result**

```
{
  "title": "habit",
  "fields": ["title", "weight", "number_of_pages"]
}
```

**JOE:** Excellent! The first step is to parse the JSON string into a data structure.

**THEO:** Let me guess: It's gonna be a generic data structure.

**JOE:** Of course! In that case, we'll have a map. Usually, JSON parsing is handled by the web server framework, but I'm going to show you how to do it manually.

**THEO:** What do you mean by a web server framework?

**JOE:** Stuff like express in node.js, Spring in Java, Django in Python, Ruby on Rails, ASP.net in C#.

**THEO:** Oh. I see. So how do you manually parse a JSON string into a map?

**JOE:** In JavaScript, we use `JSON.parse`. In Java we use a third-party library like `gson` (<https://github.com/google/gson>), maintained by Google.

Joe steps to the whiteboard and writes two code fragments, as in [11.2](#) and [11.3](#).

### Listing 11.2 Parsing a JSON string in JavaScript

```
var jsonString = "{\"title\":\"habit\", \"fields\":[\"title\", \"weight\", \"number_of_pages\"]}";  
JSON.parse(jsonString);
```

### Listing 11.3 Parsing a JSON string in Java with gson

```
var jsonString = "{\"title\":\"habit\", \"fields\":[\"title\", \"weight\", \"number_of_pages\"]}";  
gson.fromJson(jsonString, Map.class);
```

**JOE:** Can you write the JSON schema for the payload of a search request?

**THEO:** Sure.

Theo takes some time at the whiteboard writing out the schema, as in [11.4](#).

### Listing 11.4 The JSON schema for a search request

```
var searchBooksRequestSchema = {
  "type": "object",
  "properties": {
    "title": {"type": "string"},
    "fields": {
      "type": "array",
      "items": {
        "enum": [
          "title",
          "full_title",
          "subtitle",
          "publisher",
          "publish_date",
          "weight",
          "physical_dimensions",
          "number_of_pages",
          "subjects",
          "publishers",
          "genre"
        ]
      }
    }
  },
  "required": ["title", "fields"]
};
```

**JOE:** Nice! You marked the elements in the `fields` array as *enums* and not as *strings*. Where did you get the list of allowed values from?

**THEO:** Nancy gave me the list of fields that she wants to expose to the users.

Theo grabs his laptop and shows Joe the list he got from Nancy, as in [11.5](#).

### Listing 11.5 The important fields from OpenLibrary Books API

```
- title
- full_title
- subtitle
- publisher
- publish_date
- weight
- physical_dimensions
- number_of_pages
- subjects
- publishers
- genre
```

## 11.5 Represent a server response as a map

**JOE:** What does the Open Library Books API look like?

**THEO:** It's quite straightforward: you create a GET request with the book ISBN and it gives you back a JSON string with extended information about the book.

Theo executes a code snippet like the one in [11.6](#).

## Listing 11.6 Fetching data from OpenLibrary Books API

```

fetchAndLog("https://openlibrary.org/isbn/978-1982137274.json"); ①
//{
//  "authors": [
//    {
//      "key": "/authors/OL383159A",
//    },
//    {
//      "key": "/authors/OL30179A",
//    },
//    {
//      "key": "/authors/OL1802361A",
//    },
//  ],
//  "created": {
//    "type": "/type/datetime",
//    "value": "2020-08-17T14:26:27.274890",
//  },
//  "full_title": "7 Habits of Highly Effective
// People : Revised and Updated Powerful
// Lessons in Personal Change",
//  "isbn_13": [
//    "9781982137274",
//  ],
//  "key": "/books/OL28896586M",
//  "languages": [
//    {
//      "key": "/languages/eng",
//    },
//  ],
//  "last_modified": {
//    "type": "/type/datetime",
//    "value": "2021-09-08T19:07:57.049009",
//  },
//  "latest_revision": 3,
//  "lc_classifications": [
//    "",
//  ],
//  "number_of_pages": 432,
//  "publish_date": "2020",
//  "publishers": [
//    "Simon & Schuster, Incorporated",
//  ],
//  "revision": 3,
//  "source_records": [
//    "bwb:9781982137274",
//  ],
//  "subtitle": "Powerful Lessons in Personal Change",
//  "title": "7 Habits of Highly Effective
// People : Revised and Updated",
//  "type": {
//    "key": "/type/edition",
//  },
//  "works": [
//    {
//      "key": "/works/OL2629977W",
//    },
//  ],
// }
//}

```

- ① A utility function that fetches JSON and displays it to the console

**JOE:** Did Nancy ask for any special treatment of the fields returned by the API?

**THEO:** Nothing special beside keeping only the fields I showed you.

**JOE:** That's it?

**THEO:** Yes. Here's the JSON string returned by OpenLibrary Books API for "7 Habits of Highly Effective People", after having kept only the necessary fields.

Theo shows the JSON string in [11.7](#).

#### Listing 11.7 OpenLibrary Books API response for "7 Habits of Highly Effective People"

```
{
  "title": "7 Habits of Highly Effective People : Revised and Updated",
  "subtitle": "Powerful Lessons in Personal Change",
  "number_of_pages": 432,
  "full_title": "7 Habits of Highly Effective People : Revised and Updated
Powerful Lessons in Personal Change",
  "publish_date": "2020",
  "publishers": ["Simon & Schuster, Incorporated"]
}
```

**THEO:** Also, we need to keep only the fields that appear in the client request.

**JOE:** Do you know how to implement the double field filtering?

**THEO:** Yeah. I'll parse the JSON string from the API into a hash map, like we parsed a client request and then I'll use `_.pick` twice to keep only the required fields.

**JOE:** It sounds like a great plan to me. Can you code it, including validating data that is returned by the Open Books API?

**THEO:** Sure! Let me first write the JSON schema for the Open Books API response.

Theo needs to refresh his memory with the materials about schema composition in order to express the fact that either `isbn_10` or `isbn_13` is mandatory. The resulting schema is in [11.8](#).

### Listing 11.8 The JSON schema for the Open Books API response

```

var basicBookInfoSchema = {
    "type": "object",
    "required": ["title"],
    "properties": {
        "title": {"type": "string"},
        "publishers": {
            "type": "array",
            "items": {"type": "string"}
        },
        "number_of_pages": {"type": "integer"},
        "weight": {"type": "string"},
        "physical_format": {"type": "string"},
        "subjects": {
            "type": "array",
            "items": {"type": "string"}
        },
        "isbn_13": {
            "type": "array",
            "items": {"type": "string"}
        },
        "isbn_10": {
            "type": "array",
            "items": {"type": "string"}
        },
        "publish_date": {"type": "string"},
        "physical_dimensions": {"type": "string"}
    }
};

var mandatoryISBN13 = {
    "type": "object",
    "required": ["isbn_13"]
};

var mandatoryISBN10 = {
    "type": "object",
    "required": ["isbn_10"]
};

var bookInfoSchema = {
    "allOf": [
        basicBookInfoSchema,
        {
            "anyOf": [mandatoryISBN13, mandatoryISBN10]
        }
    ]
};

```

**THEO:** Now, assuming that I have a `fetchResponseBody` function that sends a request and retrieves the body of the response as a string, let me code up the how to do the retrieval. Give me a sec.

Theo types away in his IDE for several minutes and then shows the result ([11.9](#)) to Joe.

### Listing 11.9 Retrieving book information from OpenLibrary

```

var ajv = new Ajv({allErrors: true});
class OpenLibraryDataSource {
  static rawBookInfo(isbn) {
    var url = `https://openlibrary.org/isbn/${isbn}.json`;
    var jsonString = fetchResponseBody(url); ①
    return JSON.parse(jsonString);
  }

  static bookInfo(isbn, requestedFields) {
    var relevantFields = ["title", "full_title",
      "subtitle", "publisher",
      "publish_date", "weight",
      "physical_dimensions", "genre",
      "subjects", "number_of_pages"];
    var rawInfo = rawBookInfo(isbn);
    if(!ajv.validate(bookInfoSchema, rawInfo)) {
      var errors = ajv.errorsText(ajv.errors);
      throw "Internal error: Unexpected result from Open Books API: " + errors;
    }
    var relevantInfo = _.pick(_.pick(rawInfo, relevantFields), requestedFields);
    return _.set(relevantInfo, "isbn", isbn);
  }
}

```

- ① fetches JSON in the body of a response

**WARNING** The JavaScript snippets of this chapter are written as if JavaScript were dealing with I/O in a synchronous way. In real-life, we need to use `async` and `await` around I/O calls.

**JOE:** Looks good! But why did you add the `isbn` field in the map returned by `bookInfo`?

**THEO:** It will allow me to combine information from two sources about the same book.

**JOE:** I like it!

## 11.6 Passing information forward

**JOE:** If I understand it correctly, the program needs to combine two kinds of data: basic book information from the database and extended book information from OpenLibrary Books API. How are you going to combine them into a single piece of data in the response to be sent to the client?

**THEO:** In traditional OOP, I would create a specific class for each type of book information.

**JOE:** What do you mean?

**THEO:** You know, I'd have classes like `DBBook`, `OpenLibraryBook` and `CombinedBook`.

**JOE:** Hmm...

**THEO:** But it wouldn't work because we've decided to go with a dynamic approach where the client decides what fields should appear in the response.

**JOE:** True! And classes don't bring any added value here since we just need to pass data forward. Do you know the story of the guy who asked his friend to bring flowers to his fiancée?

**THEO:** No.

Joe takes a solemn position and tells Theo the following story as if reciting a poem.

**SIDE BAR**

**The story of the guy who asked his friend to bring flowers to his fiancée**

A few weeks before their wedding, Hugo wanted to offer flowers to Iris, his fiancée, who was on vacation with her family in a neighboring town. Unable to travel because of the wedding preparations, Hugo asked his friend Willy to make the trip and bring to his beloved a bouquet of flowers accompanied by an envelope containing a love letter that Hugo had written for his fiancée. Willy, having to make the trip anyway, kindly accepted.

Just before giving the flowers to Iris, Willy gave a phone call to his friend to inform him that he is about to do his duty. Hugo's joy was at its peak. During the conversation, Willy told Hugo about his admiration for the quality of his writing style.

Hugo was disappointed:

—"What? Did you read the letter I wrote to my fiancée?" Hugo asked

—"Of course! It was necessary in order to fulfill my duty faithfully." answered Willy.

**THEO:** Willy doesn't make any sense! Why would he have to read the letter in order to fulfill his duty?

**JOE:** That's exactly the point of the story! In a sense, traditional OOP is like Hugo's friend, Willy: in order to pass information forward OOP developers think they need to *open the letter* and represent information with specific classes.

**THEO:** Oh I see. And DOP developers are in the spirit of what Hugo expected from a delivery person: they just pass information forward, as generic data structures.

**JOE:** Exactly.

**THEO:** That's a funny analogy.

**JOE:** Let's get back to the question of combining data from the database with data from Books API. There are 2 ways to do it: nesting and merging.

**THEO:** How does nesting work?

**JOE:** In nesting, we add a field named `extendedInfo` to the information fetched from OpenLibrary.

**THEO:** I see. And what about merging?

**JOE:** In merging, we combine the fields of both maps into a single map.

**THEO:** And if there are fields with the same name in both maps?

**JOE:** Then, you have a merge conflict and you need to decide how to handle the conflict! That's the drawback of merging.

**Table 11.1 Two ways to combine hash maps**

	Advantages	Disadvantages
Nesting	No need to handle conflicts	Result is not flat
Merging	Result is flat	Need to handle conflicts

**WARNING** When merging maps, we need to think about the occurrences of conflicting fields.

**THEO:** Hopefully, in the context of extended search results, the maps don't have any field in common.

**JOE:** Let's merge them, then!

**THEO:** Would I need to write custom code for merging two maps?

**JOE:** No! As you might remember from one of our previous sessions, Lodash provides a handy `_.merge` function.

**TIP** `_.merge` was introduced in Chapter 5.

**THEO:** Could you refresh my memory?

**JOE:** Sure. Show me an example of maps with data from the database and data from OpenLibrary Books API and I'll show you how to merge.

**THEO:** From the database, we get only two fields: `isbn` and `available`. From the OpenLibrary API, we get 7 fields. Here's what they look like.

Theo show Joe the code snippets in [11.10](#) and [11.11](#)

### Listing 11.10 A map with book information from the database

```
var dataFromDb = {
  "available": true,
  "isbn": "978-1982137274"
};
```

### Listing 11.11 A map with book information from OpenLibrary Books API

```
var dataFromOpenLib = {
  "title": "7 Habits of Highly Effective People : Revised and Updated",
  "subtitle": "Powerful Lessons in Personal Change",
  "number_of_pages": 432,
  "full_title": "7 Habits of Highly Effective People : \
Revised and Updated Powerful Lessons in Personal Change",
  "publish_date": "2020",
  "publishers": ["Simon & Schuster, Incorporated"]
};
```

**JOE:** After calling `_.merge`, the result is a map with fields from both maps.

Joe shows Theo the merge as in [11.12](#).

### Listing 11.12 Merging two maps

```
_.merge(dataFromDb, dataFromOpenLib);
//{
//  "available": true,
//  "full_title": "7 Habits of Highly Effective People :\
//Revised and Updated Powerful Lessons in Personal Change",
//  "isbn": "978-1982137274",
//  "number_of_pages": 432,
//  "publish_date": "2020",
//  "publishers": ["Simon & Schuster, Incorporated"],
//  "subtitle": "Powerful Lessons in Personal Change",
//  "title": "7 Habits of Highly Effective People : Revised and Updated"
//}
```

**THEO:** Let me write down the JSON schema for the search books response.

Theo types into his IDE for a while producing the schema in [11.13](#).

### Listing 11.13 JSON schema for search books response

```
var searchBooksResponseSchema = {
  "type": "object",
  "required": ["title", "isbn", "available"],
  "properties": {
    "title": {"type": "string"},
    "available": {"type": "boolean"},
    "publishers": {
      "type": "array",
      "items": {"type": "string"}
    },
    "number_of_pages": {"type": "integer"},
    "weight": {"type": "string"},
    "physical_format": {"type": "string"},
    "subjects": {
      "type": "array",
      "items": {"type": "string"}
    },
    "isbn": {"type": "string"},
    "publish_date": {"type": "string"},
    "physical_dimensions": {"type": "string"}
  }
};
```

## 11.7 Search result enrichment in action

**THEO:** I think I have all the pieces in order to implement the enrichment of search results.

**JOE:** Could you write down the steps of the data flow?

**THEO:** Sure.

Theo steps to the whiteboard and writes a list of steps in the enrichment flow, as in the sidebar.

#### SIDE BAR

#### The steps of search result enrichment data flow

1. Receive a request from a client
2. Extract from the request the query and the fields to fetch from OpenLibrary
3. Retrieve from the database the books that match the query
4. Fetch information from OpenLibrary for each ISBN that match the query
5. Extract from the OpenLibrary responses the required fields
6. Combine book information from the database with information from OpenLibrary
7. Send the response to the client

**JOE:** Perfect! Would you like to try to implement it?

**THEO:** I think I'll start with the implementation of the book retrieval from the database. It's quite the same as what we did in the previous chapter.

**JOE:** Actually, it's even simpler since you don't need to join tables.

**THEO:** Right, I need only values for `isbn` and `available` column.

Theo works for a bit in his IDE, produces the code in [11.14](#), and shows it to Joe.

### Listing 11.14 Retrieving books whose title matches a query

```
var dbSearchResultSchema = {
    "type": "array",
    "items": {
        "type": "object",
        "required": ["isbn", "available"],
        "properties": {
            "isbn": {"type": "string"},
            "available": {"type": "boolean"}
        }
    }
};

class CatalogDB {
    static matchingBooks(title) {
        var matchingBooksQuery =
`SELECT isbn, available
FROM books
WHERE title = like '%$1%';
`;
        var books = dbClient.query(catalogDB, matchingBooksQuery, [title]);
        if(!ajv.validate(dbSearchResultSchema, books)) {
            var errors = ajv.errorsText(ajv.errors);
            throw "Internal error: Unexpected result from the database: " + errors;
        }
        return books;
    }
}
```

**JOE:** So far so good.

**THEO:** Next, I'll go with the implementation of the retrieval of book information from OpenLibrary for several books. Unfortunately, OpenLibrary API doesn't support querying several books at once. I'll need to send a request per book.

**JOE:** That's a bit annoying. Let's make our life easier and pretend that `_.map` works with asynchronous function. In real life, you'd need something like `Promise.all` in order to send the requests in parallel and combine the responses.

**THEO:** OK, then it's quite straightforward. I'll take the book retrieval code (see [11.9](#)) and add a `multipleBookInfo` function that maps over `bookInfo`.

Theo concentrates as he types once again into his IDE. When he's done he shows the result in [11.15](#) to Joe.

### Listing 11.15 Retrieving book information from OpenLibrary for several books

```

class OpenLibraryDataSource {
  static rawBookInfo(isbn) {
    var url = `https://openlibrary.org/isbn/${isbn}.json`;
    var jsonString = fetchResponseBody(url);
    return JSON.parse(jsonString);
  }

  static bookInfo(isbn, requestedFields) {
    var relevantFields = ["title", "full_title",
      "subtitle", "publisher",
      "publish_date", "weight",
      "physical_dimensions", "genre",
      "subjects", "number_of_pages"];
    var rawInfo = rawBookInfo(isbn);
    if(!ajv.validate(dbSearchResultSchema, bookInfoSchema)) {
      var errors = ajv.errorsText(ajv.errors);
      throw "Internal error: Unexpected result from Open Books API: " + errors;
    }
    var relevantInfo = _.pick(_.pick(rawInfo, relevantFields), requestedFields);
    return _.set(relevantInfo, "isbn", isbn);
  }

  static multipleBookInfo(isbns, fields) {
    return _.map(function(isbn) {
      return bookInfo(isbn, fields);
    }, isbns);
  }
}

```

**JOE:** Nice! Now comes the fun part: combining information from several data sources.

**THEO:** Yeah. I have two arrays in my hands: one with book information from the database and one with book information from OpenLibrary. I need somehow to join the arrays but I am not sure I can assume that the positions of book information are the same in both arrays.

**JOE:** What would you wish to have in your hands?

**THEO:** I wish I had two hash maps.

**JOE:** And what would the keys in the hash maps be?

**THEO:** Book ISBNs.

**JOE:** Well, I have good news for you: your wish is granted!

**THEO:** How?

**JOE:** Lodash provides a function named `_.keyBy` that transforms an array into a map.

**THEO:** I can't believe it. Can you show me an example?

**JOE:** Sure. Let's call `_.keyBy` on an array with two books.

Joe writes up a code snippet as in [11.16](#) and shows it to Theo.

### **Listing 11.16 Transforming an array into a map with `keyBy`**

```
var books = [
  {
    "title": "7 Habits of Highly Effective People",
    "isbn": "978-1982137274",
    "available": true
  },
  {
    "title": "The Power of Habit",
    "isbn": "978-0812981605",
    "available": false
  }
];

_.keyBy(books, "isbn");
```

When the result is displayed on the screen as in [11.17](#), Theo emits a sound of joy: `keyBy` is awesome!

**JOE:** Don't exaggerate, my friend. `_.keyBy` is quite similar to `_.groupBy`. The only difference is that `_.keyBy` assumes that there are only one element in each group.

### **Listing 11.17 The result of `keyBy`**

```
{
  "978-0812981605": {
    "available": false,
    "isbn": "978-0812981605",
    "title": "The Power of Habit"
  },
  "978-1982137274": {
    "available": true,
    "isbn": "978-1982137274",
    "title": "7 Habits of Highly Effective People"
  }
}
```

**THEO:** I think that with `_.keyBy` I'll be able to write a generic `joinArrays` function.

**JOE:** I'm glad to see you thinking in terms of implementing business logic through generic data manipulation functions.

<b>TIP</b>	<b>Many parts of business logic can be implemented through generic data manipulation functions.</b>
------------	---

**THEO:** The `joinArrays` function needs to receive the arrays and the field name by which we decide that two elements need to be combined together, for instance `isbn`.

**JOE:** In the general case, it's not necessarily the same field names for both arrays.

**THEO:** Right. Therefore, `joinArrays` needs to receive four arguments: two arrays and two field

names.

**JOE:** Go for it! And please don't forget to write a unit test for `joinArrays`.

**THEO:** Of course.

Theo works for a while and produces the code in [11.18](#) and the unit test in [11.19](#).

### Listing 11.18 A generic function for joining arrays

```
function joinArrays(a, b, keyA, keyB) {  
  var mapA = _.keyBy(a, keyA);  
  var mapB = _.keyBy(b, keyB);  
  var mapsMerged = _.merge(mapA, mapB);  
  return _.values(mapsMerged);  
}
```

### Listing 11.19 A unit test for joinArrays

```

var dbBookInfos = [
  {
    "isbn": "978-1982137274",
    "title": "7 Habits of Highly Effective People",
    "available": true
  },
  {
    "isbn": "978-0812981605",
    "title": "The Power of Habit",
    "available": false
  }
];

var openLibBookInfos = [
  {
    "isbn": "978-0812981605",
    "title": "7 Habits of Highly Effective People",
    "subtitle": "Powerful Lessons in Personal Change",
    "number_of_pages": 432,
  },
  {
    "isbn": "978-1982137274",
    "title": "The Power of Habit",
    "subtitle": "Why We Do What We Do in Life and Business",
    "subjects": [
      "Social aspects",
      "Habit",
      "Change (Psychology)"
    ],
  }
];

var joinedArrays = [
  {
    "available": true,
    "isbn": "978-1982137274",
    "subjects": [
      "Social aspects",
      "Habit",
      "Change (Psychology)",
    ],
    "subtitle": "Why We Do What We Do in Life and Business",
    "title": "The Power of Habit",
  },
  {
    "available": false,
    "isbn": "978-0812981605",
    "number_of_pages": 432,
    "subtitle": "Powerful Lessons in Personal Change",
    "title": "7 Habits of Highly Effective People",
  }
]

_.isEqual(joinedArrays,
  joinArrays(dbBookInfos, openLibBookInfos, "isbn", "isbn"));

```

**JOE:** Excellent! Now, you are ready to adjust the last piece of the extended search result endpoint.

**THEO:** That's quite easy: you fetch data from the database and from OpenLibrary and join them.

Theo works for a while, then shows Joe the code he's written as in [11.20.](#)

### **Listing 11.20 Search books and enrich book information**

```
class Catalog {
    static enrichedSearchBooksByTitle(searchPayload) {
        if(!ajv.validate(searchBooksRequestSchema, searchPayload)) {
            var errors = ajv.errorsText(ajv.errors);
            throw "Invalid request:" + errors;
        }
        var title = _.get(searchPayload, "title");
        var fields = _.get(searchPayload, "fields");

        var dbBookInfos = CatalogDataSource.matchingBooks(title);
        var isbns = _.map(dbBookInfos, "isbn");

        var openLibBookInfos = OpenLibraryDataSource.multipleBookInfo(isbns, fields);

        var res = joinArrays(dbBookInfos, openLibBookInfos);
        if(!ajv.validate(searchBooksResponseSchema, request)) {
            var errors = ajv.errorsText(ajv.errors);
            throw "Invalid response:" + errors;
        }
        return res;
    }
}
```

Theo takes a few moments to meditate about the simplicity of the code implementing the extended search endpoint as in the code snippets below. Classes are much less complex when we use them only to aggregate together stateless functions that operate on similar domain entities.

## Listing 11.21 Schema for the implementation of the extended search endpoint (Open Books API part)

```

var basicBookInfoSchema = {
  "type": "object",
  "required": ["title"],
  "properties": {
    "title": {"type": "string"},
    "publishers": {
      "type": "array",
      "items": {"type": "string"}
    },
    "number_of_pages": {"type": "integer"},
    "weight": {"type": "string"},
    "physical_format": {"type": "string"},
    "subjects": {
      "type": "array",
      "items": {"type": "string"}
    },
    "isbn_13": {
      "type": "array",
      "items": {"type": "string"}
    },
    "isbn_10": {
      "type": "array",
      "items": {"type": "string"}
    },
    "publish_date": {"type": "string"},
    "physical_dimensions": {"type": "string"}
  }
};

var mandatoryIsbn13 = {
  "type": "object",
  "required": ["isbn_13"]
};

var mandatoryIsbn10 = {
  "type": "object",
  "required": ["isbn_10"]
};

var bookInfoSchema = {
  "allOf": [
    basicBookInfoSchema,
    {
      "anyOf": [mandatoryIsbn13, mandatoryIsbn10]
    }
  ]
};

```

### Listing 11.22 Implementation of the extended search endpoint (Open Books API part)

```

var ajv = new Ajv({allErrors: true});

class OpenLibraryDataSource {
    static rawBookInfo(isbn) {
        var url = `https://openlibrary.org/isbn/${isbn}.json`;
        var jsonString = fetchResponseBody(url);
        return JSON.parse(jsonString);
    }

    static bookInfo(isbn, requestedFields) {
        var relevantFields = ["title", "full_title",
            "subtitle", "publisher",
            "publish_date", "weight",
            "physical_dimensions", "genre",
            "subjects", "number_of_pages"];
        var rawInfo = rawBookInfo(isbn);
        if(!ajv.validate(bookInfoSchema, rawInfo)) {
            var errors = ajv.errorsText(ajv.errors);
            throw "Internal error: Unexpected result from Open Books API: " + errors;
        }
        var relevantInfo = _.pick(_.pick(rawInfo, relevantFields), requestedFields);
        return _.set(relevantInfo, "isbn", isbn);
    }

    static multipleBookInfo(isbns, fields) {
        return _.map(function(isbn) {
            return bookInfo(isbn, fields);
        }, isbns);
    }
}

```

### Listing 11.23 Implementation of the extended search endpoint (Database part)

```

var dbClient;
var dbSearchResultSchema = {
    "type": "array",
    "items": {
        "type": "object",
        "required": ["isbn", "available"],
        "properties": {
            "isbn": {"type": "string"},
            "available": {"type": "boolean"}
        }
    }
};

class CatalogDB {
    static matchingBooks(title) {
        var matchingBooksQuery = `
SELECT isbn, available
FROM books
WHERE title = like '%$1%';
`;
        var books = dbClient.query(catalogDB, matchingBooksQuery, [title]);
        if(!ajv.validate(dbSearchResultSchema, books)) {
            var errors = ajv.errorsText(ajv.errors);
            throw "Internal error: Unexpected result from the database: " + errors;
        }
        return books;
    }
}

```

## Listing 11.24 Schema for the implementation of the extended search endpoint (Combining the pieces together)

```

var searchBooksRequestSchema = {
  "type": "object",
  "properties": {
    "title": {"type": "string"},
    "fields": {
      "type": "array",
      "items": {
        "type": [
          "title",
          "full_title",
          "subtitle",
          "publisher",
          "publish_date",
          "weight",
          "physical_dimensions",
          "number_of_pages",
          "subjects",
          "publishers",
          "genre"
        ]
      }
    }
  },
  "required": ["title", "fields"]
};

var searchBooksResponseSchema = {
  "type": "object",
  "required": ["title", "isbn", "available"],
  "properties": {
    "title": {"type": "string"},
    "available": {"type": "boolean"},
    "publishers": {
      "type": "array",
      "items": {"type": "string"}
    },
    "number_of_pages": {"type": "integer"},
    "weight": {"type": "string"},
    "physical_format": {"type": "string"},
    "subjects": {
      "type": "array",
      "items": {"type": "string"}
    },
    "isbn": {"type": "string"},
    "publish_date": {"type": "string"},
    "physical_dimensions": {"type": "string"}
  }
};

```

## Listing 11.25 Implementation of the extended search endpoint (Combining the pieces together)

```

class Catalog {
    static enrichedSearchBooksByTitle(request) {
        if(!ajv.validate(searchBooksRequestSchema, request)) {
            var errors = ajv.errorsText(ajv.errors);
            throw "Invalid request:" + errors;
        }

        var title = _.get(request, "title");
        var fields = _.get(request, "fields");

        var dbBookInfos = CatalogDataSource.matchingBooks(title);
        var isbnss = _.map(dbBookInfos, "isbn");

        var openLibBookInfos = OpenLibraryDataSource.multipleBookInfo(isbnss, fields);

        var response = joinArrays(dbBookInfos, openLibBookInfos);
        if(!ajv.validate(searchBooksResponseSchema, response)) {
            var errors = ajv.errorsText(ajv.errors);
            throw "Invalid response:" + errors;
        }
        return response;
    }
}

class Library {
    static searchBooksByTitle(payloadBody) {
        var payloadData = JSON.parse(payloadBody);
        var results = Catalog.searchBooksByTitle(payloadData);
        return JSON.stringify(results);
    }
}

```

**TIP** Classes are much less complex when we use them as a means to aggregate together stateless functions that operate on similar domain entities.

Joe interrupts Theo's meditation moment and congratulates him.

**JOE:** Excellent job my friend!

**THEO:** I was supposed to call Nancy later today with an ETA for the OpenLibraryBook milestone. I wonder what her reaction will be when I tell her the feature is ready!

**JOE:** Maybe you should tell her it'll be ready in a week, which would give you time to work in advance on the next milestone.

## 11.8 Delivering on time

Joe was right with his story about sharpening the saw. Theo was able to learn DOP and deliver the project on time. The Klaflim project is a success. Nancy is pleased. Theo's boss is very satisfied. Theo got promoted. What more can a person ask for?

Theo remembers his deal with Joe. It is with great pleasure that he strolls through the stores of the Westfield San Francisco Center to look for a gift for Joe's children, Neriah and Aurelia. He buys a DJI Mavic Air 2 drone for Neriah. And for Aurelia, the latest Apple Airpod Pros.

He also takes the opportunity to buy a necklace and a pair of earrings for his wife. A way for him to thank her for having endured his long days at work since the beginning of the Klaflim project.

**WARNING** The story continues in the opener of Part 3.

## 11.9 Summary

- We build the insides of our systems like we build the outsides.
- Components inside a program communicate via data that is represented as immutable data collections in the same way as components communicate via data over the wire.
- In Data-Oriented programming, the inner components of a program are loosely-coupled.
- Many parts of business logic can be implemented through generic data manipulation functions.
- We use generic functions to implement each step of the data flow inside a web service.
- We use generic functions to parse a request from a client
- We use generic functions to apply business logic to the request
- We use generic functions to fetch data from external sources (e.g. database and other web services)
- We use generic functions to apply business logic to the responses from external sources
- We use generic functions to serialize response to the client
- Classes are much less complex when we use them as a means to aggregate together stateless functions that operate on similar domain entities.

**Table 11.2 Lodash functions introduced in this chapter**

Function	Description
<code>keyBy(coll, f)</code>	Creates a map composed of keys generated from the results of running each element of <code>coll</code> through <code>f</code> ; the corresponding value for each key is the last element responsible for generating the key

# Part 3

*Maintainability*

## Innovation

After a month, the Klaufim project enters what Alabatross calls the maintenance phase. Small new features need to be added on a weekly basis. Bugs need to be fixed. Nothing dramatic.

Monica, Theo's boss, has decided to allocate Dave to the maintenance of the Klaufim project. It makes sense: over the last few months, he has demonstrated a great attitude of curiosity and seems to have solid programming skills. Theo sets up a meeting with Joe and Dave hoping that Joe will be willing to teach DOP to Dave so that he can leverage the good work he's done on Klaufim.

**THEO:** Will you have time over the next few weeks to teach Dave the principles of DOP?

**JOE:** Yes but I prefer not to.

**DAVE:** Why? Is it because I don't have enough experience in software development? I can guarantee you that I'm a fast learner.

**JOE:** It has nothing to do with your experience, Dave.

**THEO:** Why not then?

**JOE:** Theo, I think that *you* could be a great mentor for Dave.

**THEO:** But I don't even know all the parts of DOP!

**DAVE:** Come on! No false modesty between us!

**JOE:** Knowledge is never complete. As the great Socrates used to say "The more I know, the more I realize I know nothing." I'm confident you will be able to learn the missing parts by yourself and maybe even invent some.

**THEO:** How will I be able to invent missing parts?

**JOE:** You see, DOP is such a simple paradigm that it's fertile material for innovation. Part of the material I taught you, I learned from others. And part of it was an invention of mine. If you keep practicing DOP I'm quite sure you too will come up with some inventions of your own.

**THEO:** What do you say Dave? Are you willing to learn DOP from me?

**DAVE:** Definitely!

**THEO:** Joe, will you be continue to be available if we need your help from time to time?

**JOE:** Of course!

# 12

## *Advanced data validation*

### This chapter covers

- Validating function arguments
- Validating function return values
- Data validation beyond static types
- Automatic generation of data model diagrams
- Automatic generation of schema-based unit tests

## 12.1 A self-made gift

As the size of a codebase grows in a project that follows Data-Oriented Programming principles, it becomes harder to manipulate functions that receive and return only generic data. It is hard to figure out the expected shape of the function arguments and when we pass invalid data, we don't get meaningful errors.

Until now, we have illustrated how to validate data at system boundaries. In this chapter, we illustrate how to validate data when it flows inside the system by defining data schemas for function arguments and their return values. This allows us to make explicit the expected shape of function arguments and it eases development. We gain some additional benefits from this endeavor like automatic generation of data model diagrams and schema-based unit tests.

## 12.2 Function arguments validation

Dave's first task is to implement a couple of new HTTP endpoints: downloading the catalog as a CSV file, searching books by author, rating books. Once he is done with the task, Dave calls Theo for a code review.

**WARNING** The involvement of Dave in the Klapim project is explained in the opener for Part 3. Please take a moment to read the opener if you missed it.

**THEO:** Was it difficult to get your head around the DOP code?

**DAVE:** Not so much. I read your notes of the meetings with Joe and I must admit the code is quite simple to grasp.

**THEO:** Cool!

**DAVE:** But there is something that I couldn't get used to.

**THEO:** What's that?

**DAVE:** I'm struggling with the fact that all the functions receive and return generic data. In OOP I know the expected shape of the arguments for each and every function.

**THEO:** Do you validate data at system boundaries, like I have done?

**DAVE:** Absolutely, I have defined a data schema for every additional user request, database query and external service response.

**THEO:** Nice!

**DAVE:** Indeed, when the system runs on production, it works very well: When data is valid, the data flows well through the system and when data is invalid, we are able to display a meaningful error message to the user.

**THEO:** What's the problem then?

**DAVE:** The problem is that during development, it's hard to figure out the expected shape of the function arguments. And when I pass invalid data by mistake I don't get clear error messages.

**THEO:** I see. I remember that when Joe showed me how to validate data at system boundaries, I raised this concern about the development phase. Joe told me then that we validate data as it flows inside the system exactly like we validate data at system boundaries: We separate between data schema and data representation.

**DAVE:** Do we use the same schema language?

**THEO:** Yes.

**DAVE:** Cool. I like JSON schema.

**THEO:** The main purpose of data validation at system boundaries is to prevent invalid data from

getting into the system. While the main purpose of data validation inside the system is to make it easier to develop the system.

**Table 12.1 Two kinds of data validation**

Kind of data validation	Purpose	Environment
Boundaries	Guardian	Production
Inside	Ease of development	Dev

**DAVE:** By *making it easier to develop the system*, do you mean to help the developers understand the expected shape of function arguments, like in Object-Oriented programming?

**THEO:** Exactly.

**DAVE:** I'm impatient. Will you help me figure out how to validate the arguments of the function that implements book search?

**THEO:** Let me see the code of the implementation and I'll do my best.

**DAVE:** We have two implementations of book search: One from the prototype phase, where library data lives in memory and one where library data lives in the database.

**THEO:** I think that the schema for library data in memory is going to be more interesting than the schema for library data in the database, as the book search function receives catalog data in addition to the query.

**DAVE:** When you say *more interesting* data schemas, you mean more difficult to write?

**THEO:** More difficult to write but also more insightful.

**DAVE:** Let's go with library data is in memory then: the code for `Catalog.searchBooksByTitle` from the prototype phase is looks like this.

Dave pulls up some code on his laptop as in [12.1](#), and shows it to Theo.

### Listing 12.1 The implementation of search without data validation

```

class Catalog {
  static authorNames(catalogData, book) {
    var authorIds = _.get(book, "authorIds");
    var names = _.map(authorIds, function(authorId) {
      return _.get(catalogData, ["authorsById", authorId, "name"]);
    });
    return names;
  }

  static bookInfo(catalogData, book) {
    var bookInfo = {
      "title": _.get(book, "title"),
      "isbn": _.get(book, "isbn"),
      "authorNames": Catalog.authorNames(catalogData, book)
    };
    return bookInfo;
  }

  static searchBooksByTitle(catalogData, query) {
    var allBooks = _.get(catalogData, "booksByIsbn");
    var matchingBooks = _.filter(allBooks, function(book) {
      return _.get(book, "title").includes(query);
    });
    var bookInfos = _.map(matchingBooks, function(book) {
      return Catalog.bookInfo(catalogData, book);
    });
    return bookInfos;
  }
}

```

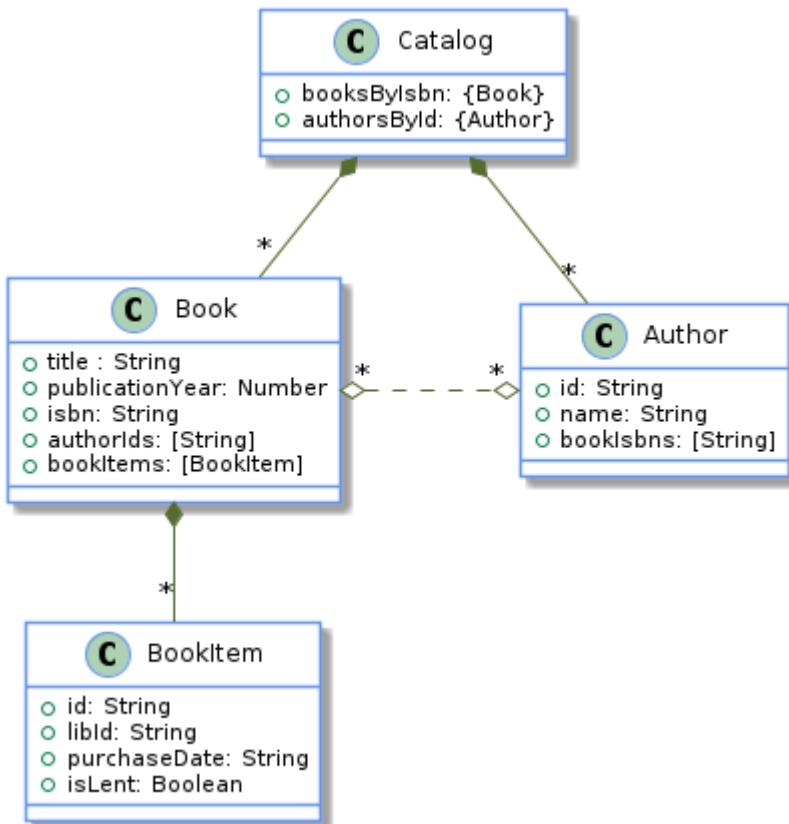
**THEO:** Please remind me of the expected shapes for catalogData and query.

**DAVE:** query should be a string and catalogData should be a map that conforms to the catalog data model.

**THEO:** What is the catalog data model?

**DAVE:** Let me see. I have seen a diagram of it somewhere... I found it!

Dave shows Theo the diagram in [12.1](#).



**Figure 12.1** The catalog data model

**THEO:** Can you try to write a JSON schema for the catalog data model?

**DAVE:** Am I allowed to use internal variables for book and author schema or do I have to nest all the schemas inside the catalog schema?

**THEO:** Why not? JSON schema is part of the code.

**DAVE:** OK. Now I need the JSON schema gift that Joe gave you.

Dave picks up a used piece of paper that is a bit torn. It contains the JSON schema cheatsheet in [12.2](#).

## Listing 12.2 JSON schema cheatsheet

```
{
  "type": "array",      ①
  "items": {
    "type": "object",   ②
    "properties": {      ③
      "myNumber": { "type": "number" },   ④
      "myString": { "type": "string" },   ⑤
      "myEnum": { "enum": [ "myVal", "yourVal" ] },  ⑥
      "myBool": { "type": "boolean" }     ⑦
    },
    "required": [ "myNumber", "myString" ],  ⑧
    "additionalProperties": false   ⑨
  }
}
```

- ① At the root level, data is an array
- ② Each element of the array is a map
- ③ The properties of each field in the map
- ④ myNumber is a number
- ⑤ myString is a string
- ⑥ myEnum is a enumeration value with two possibilities: "myVal" and "merVal"
- ⑦ myBool is a boolean
- ⑧ The mandatory fields in the map are myNumber and myString. Other fields are optional.
- ⑨ We don't allow fields that are not explicitly mentioned in the schema

**DAVE:** I think I'll start with the author schema. It seems simpler than the book schema. Here you go.

Dave shows Theo the author schema in [12.3](#).

## Listing 12.3 The author schema

```
var authorSchema = {
  "type": "object",
  "required": [ "id", "name", "bookIsbns" ],
  "properties": {
    "id": { "type": "string" },
    "name": { "type": "string" },
    "bookIsbns": {
      "type": "array",
      "items": { "type": "string" }
    }
  };
};
```

**THEO:** Well done. Move on to the book schema.

**DAVE:** I think I am going to store the book item schema in a variable.

Dave types on his laptop, and shows the result in [12.4](#) to Theo.

#### **Listing 12.4 The book schema**

```
var bookItemSchema = {
  "type": "object",
  "properties": {
    "id": {"type": "string"},
    "libId": {"type": "string"},
    "purchaseDate": {"type": "string"},
    "isLent": {"type": "boolean"}
  },
  "required": ["id", "libId", "purchaseDate", "isLent"]
};

var bookSchema = {
  "type": "object",
  "required": ["title", "isbn", "authorIds", "bookItems"],
  "properties": {
    "title": {"type": "string"},
    "publicationYear": {"type": "integer"},
    "isbn": {"type": "string"},
    "authorIds": {
      "type": "array",
      "items": {"type": "string"}
    },
    "bookItems": {
      "type": "array",
      "items": bookItemSchema
    }
  }
};
```

**TIP**

**When you define a complex data schema, it is advised to store nested schemas in variables to make the schemas easier to read.**

**THEO:** Why didn't you include publicationYear in the list of required fields on the book schema?

**DAVE:** Because, for some books the publication year is missing. Unlike in OOP, it will be easy to deal with nullable fields.

**THEO:** Excellent! And now, please tackle the final piece: the catalog schema.

**DAVE:** Here I have a problem: the catalog should be a map with two fields: booksByIsbn and authorsById. Both values should be indexes, represented in the model diagram with curly braces. I have no idea how to define the schema for an index.

**THEO:** Do you remember how we represent indexes in DOP?

**DAVE:** Yes, indexes are represented as maps.

**THEO:** Right. And what's the difference between those maps and the maps we use for records?

**DAVE:** For records, we use maps where the names of the fields are known and the values can have different shapes. While for indexes, we use maps where the names of the fields are unknown and the values have a common shape.

**THEO:** Right. We call the maps for records heterogeneous maps and the maps for indexes homogeneous maps.

**TIP**

In DOP, records are represented as heterogeneous maps while indexes are represented as homogeneous maps.

**DAVE:** How do we define the schema of an homogeneous map in JSON schema, then?

**THEO:** I don't know. Let's check the JSON schema online documentation.

After a couple of minutes of digging into the JSON schema online documentation, Theo finds a piece about `additionalProperties`.

**THEO:** I think we could use `additionalProperties`. Here's the the JSON schema for an homogeneous map where the values are numbers.

Theo shows Dave [12.5](#).

**Listing 12.5 The JSON schema for an homogeneous map where the values are numbers**

```
{
  "type": "object",
  "additionalProperties": {"type": "number"}
}
```

**DAVE:** I thought that `additionalProperties` was supposed to be a boolean and that it was used to allow or forbid properties not mentioned in the schema.

**THEO:** That's correct. Usually, `additionalProperties` is a boolean. But the documentation says it could also be a map that defines a schema. In that case, it means properties not mentioned in the schema should have a value of the schema associated to `additionalProperties`.

**DAVE:** I see. But what does that have to do with homogeneous maps?

**THEO:** Well. A homogeneous map could be seen as a map with no predefined properties where all the additional properties are of an expected type.

**DAVE:** Tricky!

**TIP**

In JSON schema, homogeneous string maps have `type: object` and with no properties and `additionalProperties` associated to a schema.

**THEO:** Indeed. Now, let me show you what the catalog schema looks like.

Theo types briefly on his laptop, then shows Dave the code in [12.6](#).

#### **Listing 12.6 The schema for catalog data**

```
var catalogSchema = {
  "type": "object",
  "properties": {
    "booksByIsbn": {
      "type": "object",
      "additionalProperties": bookSchema
    },
    "authorsById": {
      "type": "object",
      "additionalProperties": authorSchema
    }
  },
  "required": [ "booksByIsbn", "authorsById" ]
};
```

**DAVE:** Are we ready to plug the catalog and the query schema into the `Catalog.searchBooksByTitle` implementation?

**THEO:** We could, but I think we can do better by defining a single schema that combines catalog and query schemas.

**DAVE:** How would we combine two schemas into a single schema?

**THEO:** Do you know what a tuple is?

**DAVE:** I think I know but I couldn't define it formally.

**THEO:** A tuple is an array where the size is fixed and the elements could be of different shapes.

**DAVE:** OK. And how do we define tuples in JSON schema?

Once again, Theo explores the JSON schema online documentation.

**THEO:** I found it: We use `prefixItems`, as you can see for instance in the definition of a tuple made of a string and a number.

Theo shows Dave the code in [12.7](#).

#### **Listing 12.7 The schema for a tuple made of a string and a number**

```
{
  "type": "array",
  "prefixItems": [
    { "type": "string" },
    { "type": "number" }
  ]
}
```

**DAVE:** I see. And how would you define the schema for the arguments of `Catalog.searchBooksByTitle`?

**THEO:** Well, it's a tuple of size 2, where the first element is a catalog and the second element is a string.

**DAVE:** Something like this schema?

Dave brings up [12.8](#) on his laptop and shows it to Theo.

#### **Listing 12.8 The schema for the arguments of `Catalog.searchBooksByTitle`**

```
var searchBooksArgsSchema = {
  "type": "array",
  "prefixItems": [
    catalogSchema,
    { "type": "string" },
  ]
};
```

**THEO:** Exactly.

**DAVE:** Now that we have the schema for the arguments. How do we plug it into the implementation of search books?

**THEO:** It's very similar to the way we validate data at system boundaries. The main difference is that the data validation for data that flows inside the system should run only at development time and it should be disabled when the code runs in production.

**DAVE:** Why?

**THEO:** Because that data has been already validated up front at a system boundary. Validating it again on a function call is superfluous and it would impact performance.

**DAVE:** When you say development time, does that include testing and staging environments?

**THEO:** Yes. All the environments beside production.

**DAVE:** I see. It's like assertions in Java. They are disabled in production code.

**TIP**

**Data validation inside the system should be disabled in production.**

**THEO:** Exactly! For now, I am going to assume that we have a `dev` function that returns `true` when the code runs in development environment and `false` when it runs in production. Having said that, take a look at this.

Theo types some code on his laptop and shows the result in [12.9](#) to Dave.

### Listing 12.9 The implementation of search with validation of function arguments

```
Catalog.searchBooksByTitle = function(catalogData, query) {
  if(dev()) { ①
    var args = [catalogData, query];
    if(!ajv.validate(searchBooksArgsSchema, args)) {
      var errors = ajv.errorsText(ajv.errors);
      throw ("searchBooksByTitle called with invalid arguments: " + errors);
    }
  }

  var allBooks = _.get(catalogData, "booksByIsbn");
  var matchingBooks = _.filter(allBooks, function(book) {
    return _.get(book, "title").includes(query);
  });
  var bookInfos = _.map(matchingBooks, function(book) {
    return Catalog.bookInfo(catalogData, book);
  });

  return bookInfos;
};
```

- ① dev is a function whose implementation depends on the run-time environment: it returns true when the code runs in development environment and false when it runs in production

**DAVE:** Do you think we should validate the arguments of all the functions?

**THEO:** No. I think we should treat data validation like we treat unit tests. We should validate function arguments only for functions for whom we would write unit tests.

**TIP**

Treat data validation like unit tests.

## 12.3 Return value validation

**DAVE:** Do you think it would make sense to also validate the return value of functions?

**THEO:** Absolutely.

**DAVE:** Cool. Let me try to write the JSON schema for the return value of `Catalog.searchBooksByTitle`.

After a few minutes, Dave comes up with the schema in [12.10](#).

### Listing 12.10 The schema for the return value of `Catalog.searchBooksByTitle`

```
var searchBooksResponseSchema = {
    "type": "array",
    "items": {
        "type": "object",
        "required": ["title", "isbn", "authorNames"],
        "properties": {
            "title": {"type": "string"},
            "isbn": {"type": "string"},
            "authorNames": {
                "type": "array",
                "items": {"type": "string"}
            }
        }
    }
};
```

**THEO:** Well done! Now, would you like to try adding return value validation to the code of `Catalog.searchBooksByTitle`?

**DAVE:** Sure.

Dave works for a bit in his IDE, and shows the result in [12.11](#) to Theo.

### Listing 12.11 The implementation of search with data validation for both input and output

```
Catalog.searchBooksByTitle = function(catalogData, query) {
    if(dev()) {
        if(!ajv.validate(searchBooksArgsSchema, [catalogData, query])) {
            var errors = ajv.errorsText(ajv.errors);
            throw ("searchBooksByTitle called with invalid arguments: " + errors);
        }
    }

    var allBooks = _.get(catalogData, "booksByIsbn");
    var matchingBooks = _.filter(allBooks, function(book) {
        return _.get(book, "title").includes(query);
    });
    var bookInfos = _.map(matchingBooks, function(book) {
        return Catalog.bookInfo(catalogData, book);
    });

    if(dev()) {
        if(!ajv.validate(searchBooksResponseSchema, bookInfos)) {
            var errors = ajv.errorsText(ajv.errors);
            throw ("searchBooksByTitle returned a value that doesn't conform to schema: " + errors);
        }
    }
    return bookInfos;
};
```

**THEO:** Excellent! Now we need to figure out how to deal with advanced data validation.

## 12.4 Advanced data validation

**DAVE:** What do you mean by *advanced data validation*?

**THEO:** I mean going beyond static types.

**DAVE:** Could you give me an example?

**THEO:** Sure. Take for instance the publication year of a book. It's an integer. But what else could you say about this book?

**DAVE:** It has to be positive. It would say it's a positive integer.

**THEO:** Come on! Be courageous: go beyond types.

**DAVE:** I don't know. I would say it's a number that should be higher than 1900. I don't think it makes sense to have a book that is published before 1900.

**THEO:** Exactly. And what about the higher limit?

**DAVE:** I'd say that the publication year should be less than the current year.

**THEO:** Very good! I see that JSON schema supports number ranges. Here is how we write the schema for an integer that should be between 1900 and 2021:

Theo types for a bit, and shows the code in [12.12](#) to Dave.

### Listing 12.12 The schema for an integer between 1900 and 2021

```
var publicationYearSchema = {
  "type": "integer",
  "minimum": 1900,
  "maximum": 2021
};
```

**DAVE:** Why isn't this kind of data validation possible in Object-Oriented programming?

**THEO:** I'll let you think about it for a moment.

**DAVE:** I think have it! In Data-Oriented programming, data validation is executed at run-time while static type validation in Object-Oriented Programming is executed at compile-time. At compile-time, we have only information about static types while at run time we have the data itself. That's why in DOP data validation, it's possible to go beyond types!

**NOTE**

Of course, it's also possible in traditional OOP to write custom run-time data validation. Here, we are comparing data schema with static types.

**THEO:** You got it right! Now, let me show you how to write the schema for a string that should match a regular expression.<sup>9</sup> Let's take for example the book id. I am assuming it must be a UUID ([https://en.wikipedia.org/wiki/Universally\\_unique\\_identifier](https://en.wikipedia.org/wiki/Universally_unique_identifier)).

**DAVE:** Right.

**THEO:** Could you write down the regular expression for a valid UUID?

Dave googles "UUID regex" and he finds the beautiful regular expression that appears in [12.13](#).

### Listing 12.13 The regular expression for a valid UUID

```
[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}
```

**DAVE:** How do you plug a regular expression into a JSON Schema?

**THEO:** While you were looking for the UUID regular expression, I read about the `pattern` field. Here's how we can plug the UUID regular expression into a JSON schema.

Theo brings up [12.14](#) and shows it to Dave.

### Listing 12.14 The schema for a UUID

```
var uuidSchema = {
  "type": "string",
  "pattern": "[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}"
};
```

**DAVE:** Nice! Let me improve the catalog schema and refine the schema for `purchaseDate`, `isbn`, `libId` and `authorId` with regular expressions.

**THEO:** Before you do that, let me tell you something I read about regular expressions. Some of them are predefined. For instance, there is a predefined regular expression for dates.

**DAVE:** How does it work?

**THEO:** With the help of the `format` field.

Theo shows Dave the schema as in [12.15](#).

### Listing 12.15 The schema for a date

```
{
  "type": "string",
  "format": "date"
}
```

**TIP**

In Data-Oriented Programming, data validation goes beyond static types (e.g. number ranges, regular expressions).

**DAVE:** Very cool! Do I have all the information I need in order to refine the catalog schema?

**THEO:** Yes. Go ahead!

It takes Dave a bit of time to write the regular expressions for `isbn`, `authorId` and `libId`. But with the help of Google (again) and a bit of simplification, Dave comes up with the schema in [12.16](#) and in [12.17](#).

### **Listing 12.16 The refined schema of the catalog data (Part 1)**

```
var isbnSchema = {
  "type": "string",
  "pattern": "^[0-9-]{10,20}$"
};

var libIdSchema = {
  "type": "string",
  "pattern": "^[a-z0-9-]{3,20}$"
};

var authorIdSchema ={
  "type": "string",
  "pattern": "[a-z-]{2,50}"
};

var bookItemSchema = {
  "type": "object",
  "additionalProperties": {
    "id": uuidSchema,
    "libId": libIdSchema,
    "purchaseDate": {
      "type": "string",
      "format": "date"
    },
    "isLent": { "type": "boolean" }
  }
};
```

### Listing 12.17 The refined schema of the catalog data (Part 2)

```

var bookSchema = {
  "type": "object",
  "required": ["title", "isbn", "authorIds", "bookItems"],
  "properties": {
    "title": {"type": "string"},
    "publicationYear": publicationYearSchema,
    "isbn": isbnSchema,
    "publisher": {"type": "string"},
    "authorIds": {
      "type": "array",
      "items": authorIdSchema
    },
    "bookItems": bookItemSchema
  }
};

var authorSchema = {
  "type": "object",
  "required": ["id", "name", "bookIsbns"],
  "properties": {
    "id": {"type": "string"},
    "name": {"type": "string"},
    "bookIsbns": {
      "items": isbnSchema
    }
  }
};

var catalogSchema = {
  "type": "object",
  "properties": {
    "booksByIsbn": {
      "type": "object",
      "additionalProperties": bookSchema
    },
    "authorsById": {
      "type": "object",
      "additionalProperties": authorSchema
    }
  },
  "required": ["booksByIsbn", "authorsById"]
};

```

## 12.5 Automatic generation of data model diagrams

Before going home, Theo gives a phone call to Joe to tell him about how he and Dave used data validation inside the system. Joe tells Theo that it's exactly how he recommends doing it and suggests he come and visit Theo and Dave at the office tomorrow to show them some cool advanced stuff related to data validation.

**JOE:** Are you guys starting to feel the power of data validation à la Data-Oriented Programming?

**DAVE:** Yes. It's a bit less convenient to validate a JSON schema than to write down the class of function arguments. But this drawback is compensated by the fact that JSON schema supports conditions that go beyond static types.

**THEO:** We also realized that we don't have to validate data for each and every function.

**JOE:** Correct. Now, let me show you another cool thing that we can do with JSON schema.

**DAVE:** What's that?

**JOE:** Generate a data model diagram.

**DAVE:** Wow! How does that work?

**JOE:** There are tools that receive a JSON schema as an input and produce a data model diagram in a data model format.

**DAVE:** What is a data model format?

**JOE:** It's a format that allows you to define a data model in plain text. After that, you can generate an image from the text. My favorite data format in PlantUML (<https://plantuml.com/>).

**DAVE:** Do you know of any tools that generate data model diagrams?

**JOE:** I have used JSON Schema Viewer (<https://navneethg.github.io/jsonschemaviewer/>) and Malli (<https://github.com/matosin/malli>).

Joes shows shows Dave and Theo [12.18](#), the PlantUML diagram that Malli generated from the catalog schema in [12.16](#) and [12.17](#).

**Listing 12.18 A PlantUML diagram generated from the catalog data schema**

```

@startuml

Entity1      *-- Entity2
Entity1      *-- Entity4

Entity2      *-- Entity3

class Entity1 {
    + booksByIsbn: {Entity2}
    + authorsById: {Entity4}
}

class Entity2 {
    + title : String
    + publicationYear: Number
    + isbn: String
    + authorIds: [String]
    + bookItems: [Entity3]
}

class Entity3 {
    + id: String
    + libId: String
    + purchaseDate: String
    + isLent: Boolean
}

class Entity4 {
    + id: String
    + name: String
    + bookIsbns: [String]
}

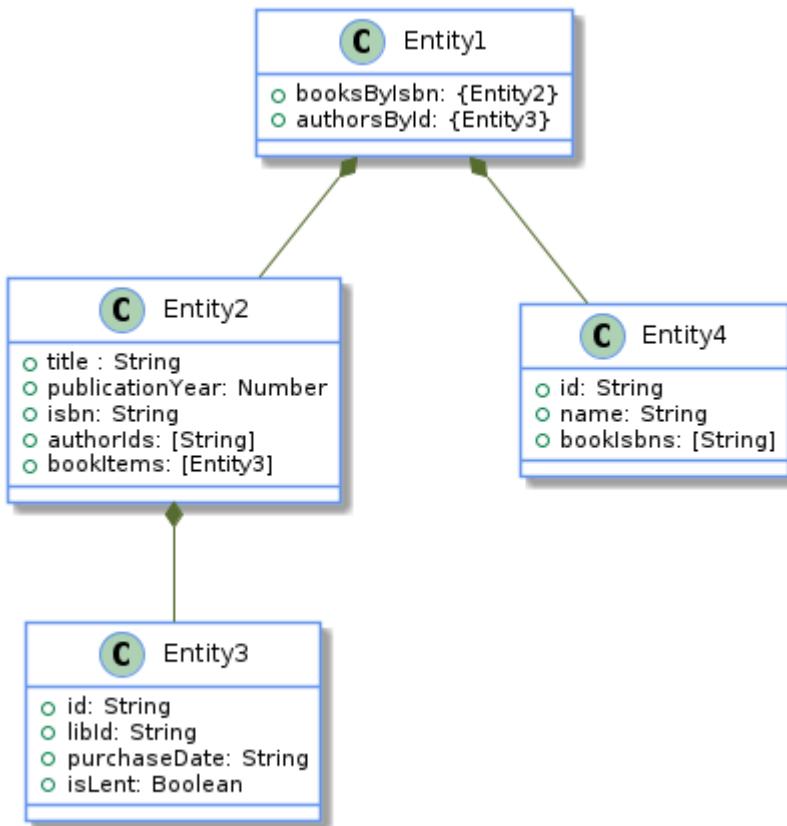
@enduml

```

**DAVE:** Is it possible to visualize this diagram?

**JOE:** Absolutely. Let me copy and paste the diagram text into PlantText online tool (<https://www.planttext.com/>).

In Dave's web browser, they see a diagram that looks like the image in [12.2](#).



**Figure 12.2 A visualization of the PlantUML diagram generated from the catalog data schema**

**DAVE:** That's cool! But why are the diagram entities named Entity1, Entity2 etc...?

**JOE:** Because in JSON schema, there's no way to give a name to a schema. Malli has to auto generate random names for you.

**THEO:** Also, I see that the extra information we have in the schema (like the number range for publicationYear and string regular expression for isbn) is missing from the diagram.

**JOE:** Right. That extra information is not part of the data model. That's why it's not included in the generated data model diagram.

**DAVE:** Anyway, it's very cool!

**JOE:** If you guys like the data model generation feature, I'm sure you're going to like the next feature.

**DAVE:** What's it about?

**JOE:** Automatic generation of unit tests.

**THEO:** Sounds exciting!

## 12.6 Automatic generation of schema-based unit tests

**JOE:** Once you've defined a data schema for function arguments and for its return value, it's quite simple to generate a unit test for this function.

**DAVE:** How?

**JOE:** Well. Think about it. What's the essence of a unit test for a function?

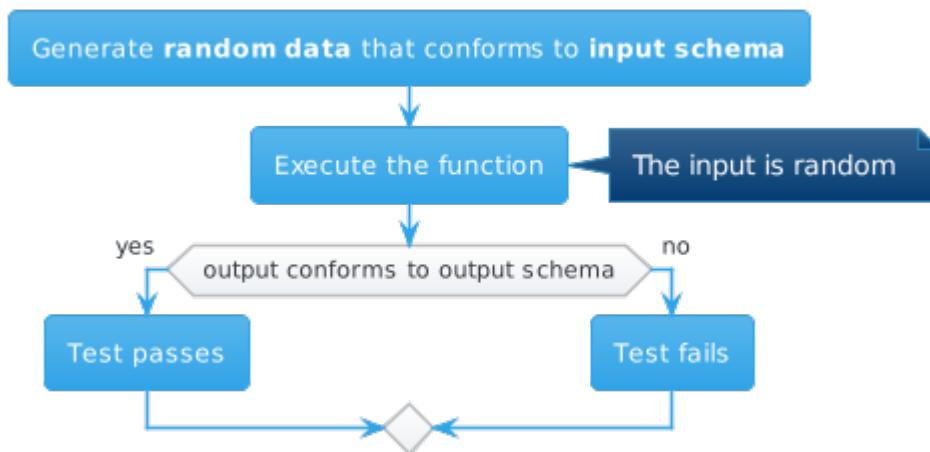
**DAVE:** A unit test calls a function with some arguments and check whether the function returns the expected value.

**JOE:** Exactly! Now, let's adapt it to the context of data schema and Data-Oriented programming. Let's say you have a function with a schema for their arguments and for their return value.

**DAVE:** OK.

**JOE:** Here's the flow of a schema-based unit test: we call the function with random arguments that conform to the schema of the function arguments. Then, we check whether the function returns a value that conforms to the schema of the return value. Here, let me diagram it out.

Joe goes to the whiteboard and draws the diagram in [12.3](#).



**Figure 12.3** The flow of a schema-based unit test

**DAVE:** How do you generate random data that conform to a schema?

**JOE:** Using a tool like JSON Schema Faker (<https://github.com/json-schema-faker/json-schema-faker>). Let's start with a simple schema: the schema for a UUID. Let me show you how to generate random data that conforms to the schema.

Joe types on keyboard for a bit, and shows the code as in [12.19](#) to Dave and Theo.

**Listing 12.19 Generating random data that conforms to a UUID schema**

```
var uuidSchema = {  
    "type": "string",  
    "pattern": "[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}"  
};  
  
JSONSchemaFaker.generate(uuidSchema);  
// "7aA8CdF3-14DF-9EF5-1A19-47dacdB16Fa9"
```

Dave executes the code snippet a couple of times and indeed, on each evaluation it returns a different UUID.

**DAVE:** Very cool! Let me see how it works with a more complex schemas like the catalog schema.

When Dave calls `JSONSchemaFaker.generate` with the catalog schema, he gets quite long random data as in [12.20](#).

## Listing 12.20 Generating random data that conforms to the catalog schema

```
{
  "booksByIsbn": {
    "Excepteur7": {
      "title": "elit veniam anim",
      "isbn": "5419903-3563-7",
      "authorIds": [
        "vfbzqahmuemgdegkzntfhzcjhjrbgfoljfzogfuqweggchum",
        "inxmqh-",
      ],
      "bookItems": {
        "ullamco5": {
          "id": "f7dac8c3-E59D-bc2E-7B33-C27F3794E2d6",
          "libId": "4jtbj7q7nrylfu14m",
          "purchaseDate": "2001-08-01",
          "isLent": false
        },
        "culpa_3e": {
          "id": "423DCdDF-CDAe-2CAa-f956-C6cd9dA8054b",
          "libId": "6wcxbh",
          "purchaseDate": "1970-06-24",
          "isLent": true
        }
      },
      "publicationYear": 1930,
      "publisher": "sunt do nisi"
    },
    "aliquip_d7": {
      "title": "aute",
      "isbn": "348782167518177",
      "authorIds": ["owfgtdxjbiidsobfgvjpjlxuabqpjhdcqmmmrjb-ezrsz-u"],
      "bookItems": {
        "ipsum_0b": {
          "id": "6DfE93ca-DB23-5856-56Fd-82Ab8CffEFF5",
          "libId": "bvjh0p2p2666vs7dd",
          "purchaseDate": "2018-03-30",
          "isLent": false
        }
      },
      "publisher": "ea anim ut ex id",
      "publicationYear": 1928
    }
  },
  "authorsById": {
    "labore_b88": {
      "id": "adipisicing nulla proident",
      "name": "culpa in minim",
      "bookIsbns": [
        "6243029--7",
        "5557199424742986"
      ]
    },
    "ut_dee": {
      "id": "Lorem officia culpa qui in",
      "name": "aliquip eiusmod",
      "bookIsbns": [
        "0661-8-5772"
      ]
    }
  }
}
```

**JOE:** I see that you have some bugs in your regular expressions.

**THEO:** How can you see that?

**JOE:** Some of the generated ISBNs don't seem to be valid ISBNs.

**DAVE:** You're right. I hate regular expressions!

**JOE:** Now, let me show you how to implement the flow of a schema-based unit test for `Catalog.searchBooksByTitle`.

Joe quickly types out the few lines of code as in [12.21](#).

### Listing 12.21 Implementation of the flow of a schema-based unit test

```
function searchBooksTest () {
  var catalogRandom = JSONSchemaFaker.generate(catalogSchema);
  var queryRandom = JSONSchemaFaker.generate({ "type": "string" });
  Catalog.searchBooksByTitle(catalogRandom, queryRandom);
}
```

**DAVE:** Wait a moment. I can't see where you check that `Catalog.searchBooksByTitle` returns a value that conforms to the return value schema.

**THEO:** If you look closer at the code you'll see it...

Dave takes a look at the code of `Catalog.searchBooksByTitle` in [12.22](#) and he sees it.

### Listing 12.22 The implementation of `Catalog.searchBooksByTitle`

```
Catalog.searchBooksByTitle = function(catalogData, query) {
  if(dev()) {
    if(!ajv.validate(searchBooksArgsSchema, [catalogData, query])) {
      var errors = ajv.errorsText(ajv.errors);
      throw ("searchBooksByTitle called with invalid arguments: " + errors);
    }
  }

  var allBooks = _.get(catalogData, "booksByIsbn");
  var matchingBooks = _.filter(allBooks, function(book) {
    return _.get(book, "title").includes(query);
  });
  var bookInfos = _.map(matchingBooks, function(book) {
    return Catalog.bookInfo(catalogData, book);
  });

  if(dev()) {
    if(!ajv.validate(searchBooksResponseSchema, bookInfos)) {
      var errors = ajv.errorsText(ajv.errors);
      throw ("searchBooksByTitle returned a value that doesn't conform to schema: " + errors);
    }
  }
  return bookInfos;
};
```

**DAVE:** Of course! It's in the code of `Catalog.searchBooksByTitle`. If the return value doesn't conform to the schema it throws an exception and the test will fail.

**JOE:** Correct. Now, let's improve the code of our unit test and return `false` when an exception occurs inside `Catalog.searchBooksByTitle`,

Joe edits the test code as in [12.23](#) and shows it to the others.

### Listing 12.23 A complete data schema based unit test for search books

```
function searchBooksTest () {
  var catalogRandom = JSONSchemaFaker.generate(catalogSchema);
  var queryRandom = JSONSchemaFaker.generate({ "type": "string" });
  try {
    Catalog.searchBooksByTitle(catalogRandom, queryRandom);
    return true;
  } catch (error) {
    return false;
  }
}
```

**DAVE:** Let me see what happens when I run the test.

**JOE:** Before we run it, we need to fix something in our unit test.

**DAVE:** What?

**JOE:** The catalog data and the query are random. There's a good chance that no books will match the query. We need to create a query that matches at least one book.

**DAVE:** How are we going to find a query that's guaranteed to match at least one book?

**JOE:** Our query will be the first letter of the first book from the catalog data that is generated.

Joe types for a bit and shows Theo and Dave his refined test, as in [12.24](#).

### Listing 12.24 A refined data schema based unit test for search books

```
function searchBooksTest () {
  var catalogRandom = JSONSchemaFaker.generate(catalogSchema);
  var queryRandom = JSONSchemaFaker.generate({ "type": "string" });
  try {
    var firstBook = _.values(_.get(catalogRandom, "booksByIsbn"))[0];
    var query = _.get(firstBook, "title").substring(0,1);
    Catalog.searchBooksByTitle(catalogRandom, query);
    return true;
  } catch (error) {
    return false;
  }
}
```

**DAVE:** I see. It's less complicated than what I thought. Does it happen often that you need to tweak the random data?

**JOE:** No, usually the random data is just fine.

**DAVE:** OK. Now I'm curious to see what happens when I execute the unit test.

When Dave executes the unit test in [12.25](#), it fails.

### **Listing 12.25 Running the schema-based unit test**

```
searchBooksTest();
//false
```

**DAVE:** I think something's wrong in the code of the unit test.

**THEO:** Maybe the unit test caught a bug in the implementation of `Catalog.searchBooksByTitle`.

**DAVE:** Let's check that. Is there a way to have the unit test display the return value of the function?

**JOE:** Yes. Here it is.

Joe once again shows the others his updated test as in [12.26](#).

### **Listing 12.26 Including the return value in the unit test output**

```
function searchBooksTest () {
  var catalogRandom = JSONSchemaFaker.generate(catalogSchema);
  var queryRandom = JSONSchemaFaker.generate({ "type": "string" });
  try {
    var firstBook = _.values(_.get(catalogRandom, "booksByIsbn"))[0];
    var query = _.get(firstBook, "title").substring(0,1);
    Catalog.searchBooksByTitle(catalogRandom, query);
    return true;
  } catch (error) {
    console.log(error);
    return false;
  }
}
```

**DAVE:** Now, let's see what's displayed when I run again the unit test.

Dave shows them the output as in [12.27](#).

### **Listing 12.27 Running again the schema-based unit test**

```
searchBooksTest();
//searchBooksByTitle returned a value that doesn't conform to schema:
//data[0].authorNames[0] should be string,
//data[0].authorNames[1] should be string,
//data[1].authorNames[0] should be string
```

**DAVE:** I think I understand what happened. In our random catalog data, the authors of the books are not present in the `authorByIds` index. That's why we have all those `undefined` in the value returned by `Catalog.searchBooksByTitle`, while in the schema we expect a string.

**THEO:** How do we fix that?

**DAVE:** By having Catalog.authorNames return the string Not available when an author doesn't exist in the catalog.

Dave shows his revised code as in [12.28](#).

### Listing 12.28 Fixing a bug in search books implementation

```
Catalog.authorNames = function(catalogData, book) {
  var authorIds = _.get(book, "authorIds");
  var names = _.map(authorIds, function(authorId) {
    return _.get(catalogData, ["authorsById", authorId, "name"], "Not available"); ①
  });
  return names;
};
```

- ① When no value is associated with the key ["authorsById", authorId, "name"], we return "Not available"

Dave executes the unit test again as in [12.29](#). And now, it passes.

### Listing 12.29 Running again the schema-based unit test

```
searchBooksTest();
// true
```

**JOE:** Well done Dave!

**DAVE:** You were right: the automatically generated unit tests were able to catch a bug in the implementation of Catalog.searchBooksByTitle.

**JOE:** Don't worry. It has happened to me so many times.

**DAVE:** Data validation à la DOP is really cool!

**JOE:** That's just the beginning, my friend. The more you use it, the more you love it!

**DAVE:** I must admit I still miss one cool IDE feature from Object-Oriented programming.

**JOE:** Which one?

**DAVE:** The autocompletion of field names in a class.

**JOE:** For the moment, field name autocompletion for data is only available in Clojure via clj-kondo (<https://github.com/clj-kondo/clj-kondo>) and the integration it provides with Malli (<https://github.com/metosin/malli>).

**DAVE:** Do you think that some day this functionality will be available in other programming languages?

**JOE:** Absolutely. IDEs like IntelliJ and Visual Studio Code already support JSON schema validation for JSON files. It's only a matter of time before they support JSON schema validation for function arguments and provide autocompletion of the field names in a map.

**DAVE:** I hope it won't take them too much time.

## 12.7 A new gift

When Joe leaves the office, Dave gets an interesting idea.

**DAVE:** Do you think we could make our own JSON schema gift with the advanced JSON schema features that we discovered today?

**THEO:** Excellent idea! But you'll have to do it on your own. I have to run for a meeting!

After his meeting, Theo comes back to Dave's desk station and as he sees Theo, Dave takes a small package out of his bag, like the one Joe gave Theo a few weeks ago. But this one is wrapped in a kind of light blue ribbon. Dave hands Theo the package with a solemn posture. When Theo undoes the ribbon, he discovers an elegant piece of paper decorated with pretty little designs. In the center of the paper, he manages to read the inscription "Advanced JSON schema cheatseet". Theo smiles while browsing the JSON schema (see [12.30](#)).

Then, he turns the paper over to find that the back is also filled with drawings. In the center of the paper, Theo reads the inscription "Example of valid data" (see [12.31](#)).

## Listing 12.30 Advanced JSON schema cheatsheet

```
{
  "type": "array",      ①
  "items": {
    "type": "object",   ②
    "properties": {      ③
      "myNumber": { "type": "number" },   ④
      "myString": { "type": "string" },   ⑤
      "myEnum": { "enum": [ "myVal", "yourVal" ] },  ⑥
      "myBool": { "type": "boolean" }     ⑦
      "myAge": {           ⑧
        "type": "integer",
        "minimum": 0,
        "maximum": 120
      },
      "myBirthday": {       ⑨
        "type": "string",
        "format": "date"
      },
      "myLetters": {        ⑩
        "type": "string",
        "pattern": "[a-zA-Z]*"
      }
    },
    "myNumberMap": {      ⑪
      "type": "object",
      "additionalProperties": { "type": "number" }
    },
    "myTuple": {          ⑫
      "type": "array",
      "prefixItems": [
        { "type": "string" },
        { "type": "number" }
      ]
    },
    "required": [ "myNumber", "myString" ],  ⑬
    "additionalProperties": false  ⑭
  }
}
```

- ① At the root level, data is an array
- ② Each element of the array is a map
- ③ The properties of each field in the map
- ④ `myNumber` is a number
- ⑤ `myString` is a string
- ⑥ `myEnum` is a enumeration value with two possibilities: `"myVal"` and `"merVal"`
- ⑦ `myBool` is a boolean
- ⑧ `myAge` is an integer between 0 and 120
- ⑨ `myBirthday` is a string conforming to the date format
- ⑩ `myLetters` is a string with letters only (lower case or upper case)
- ⑪ `myNumberMap` is an homogeneous string map where all the values are numbers

- ⑫ `myTuple` is a tuple where the first element is a string and the second element is a number
- ⑬ The mandatory fields in the map are `myNumber` and `myString`. Other fields are optional.
- ⑭ We don't allow fields that are not explicitly mentioned in the schema

**Listing 12.31 An example of valid data**

```
[
  {
    "myNumber": 42,
    "myString": "I-love-you",
    "myEnum": "myVal",
    "myBool": true,
    "myTuple": ["Hello", 42]
  },
  {
    "myNumber": 54,
    "myString": "Happy",
    "myAge": 42,
    "myBirthday": "1978-11-23",
    "myLetters": "Hello",
    "myNumberMap": {
      "banana": 23,
      "apple": 34
    }
  }
]
```

## 12.8 Summary

- We define data schemas using a language like JSON schema for function arguments and return values.
- Function argument schemas allow developers to figure out the expected shape of the function arguments they want to call.
- When invalid data is passed, data validation third-party libraries give meaningful errors with detailed information about the data parts that are not valid.
- Unlike data validation at system boundaries, data validation inside the system is supposed to run only at development time and to be disabled in production.
- We visualize a data schema by generating a data model diagram out of a JSON schema.
- For functions that have data schemas for their arguments and return values, we can automatically generate schema-based unit tests.
- Data validation is executed at run-time.
- We can define advanced data validation conditions that go beyond static types, like checking whether a number is within in a range or if a string matches a regular expression.
- Data validation inside the system should be disabled in production.
- Records are represented as heterogeneous maps while indexes are represented as homogeneous maps.
- When you define a complex data schema, it is advised to store nested schemas in variables to make the schemas easier to read.
- We treat data validation like unit tests.

# 13 Polymorphism

## This chapter covers

- Mimicking objects with multimethods (**Single dispatch**)
- Multimethods where implementation depends on several argument types (**Multiple dispatch**)
- Multimethods where implementation depends dynamically on several arguments (**Dynamic dispatch**)

### 13.1 Playing with the animals of the countryside

Object-Oriented Programming is well known for allowing different classes to be called with the same interface, via a mechanism called polymorphism. It may seem that the only way to have polymorphism in a program is with objects. In fact, as we are going to see in this chapter it is possible to have polymorphism without objects via multimethods.

Moreover, multimethods provide more advanced polymorphism than OOP polymorphism as they support cases where the chosen implementation depends on several argument types (multiple dispatch) and even on the dynamic value of the arguments (dynamic dispatch).

### 13.2 The essence of polymorphism

For today's session, Dave has invited Theo to come and visit him at his parent's house in the countryside. As Theo's drive takes him from the freeway to increasingly rural country roads, he lets himself be carried away by the beauty of the landscape, the smell of fresh earth, and the sounds of animals in nature. This 'nature bath' puts him in an excellent mood ...

Dave receives Theo in jeans and a T-shirt, a marked contrast with the elegant clothes he wears

at the office. A straw hat completes his country look.

Dave suggests that they go pick a few oranges in the field to squeeze for juice. After drinking a much more flavorful orange juice than they are used to in San Francisco, Theo and Dave get to work.

**DAVE:** When I was waiting for you, I thought of another thing I was missing from Object-Oriented programming.

**THEO:** What's that?

**DAVE:** Polymorphism.

**THEO:** What kind of polymorphism?

**DAVE:** You know, you define an interface and different classes implement the same interface in different ways.

**THEO:** I see. And why do you think polymorphism is valuable?

**DAVE:** Because it allows us to decouple an interface from its implementations.

**THEO:** Would you mind illustrating that with a concrete example?

**DAVE:** Sure. Since we're in the country, I'll use the classic OOP polymorphism example with animals.

**THEO:** Good idea!

**DAVE:** Let's say that each animal has its own greeting by making a sound and saying its name.

**THEO:** Oh cool. Like in anthropomorphic comics books.

**DAVE:** Anthro what?

**THEO:** You know, comics books where animals can walk, speak etc., like Mickey mouse?

**DAVE:** Of course, but I don't know that term. Where does it come from?

**THEO:** Anthropomorphism comes from the Greek *ánthrōpos* that means *human* and *morphe* that means *form*.

**DAVE:** I see. So an anthropomorphic book is a book where animals have human traits. The word sounds related to *polymorphism*.

**THEO:** Absolutely. Polymorphism comes from the Greek *polús* that means *many* and *morphe* that means *form*.

**Dave:** That makes sense. Polymorphism is the ability of different objects to implement the same method in different ways, which brings me back to my animal example. In OOP, I'd define an `IAnimal` interface with a `greet` method and each animal class would implement `greet` in its own way. Here, I happen to have an example.

Dave grabs his laptop and brings up the listing in [13.1](#).

### Listing 13.1 OOP polymorphism illustrated with animals

```
interface IAnimal {
    public void greet();
}

class Dog implements IAnimal {
    private String name;
    public void greet() {
        System.out.println("Woof woof! My name is " + animal.name);
    }
}

class Cat implements IAnimal {
    private String name;
    public void greet() {
        System.out.println("Meow! I am " + animal.name);
    }
}

class Cow implements IAnimal {
    private String name;
    public void greet() {
        System.out.println("Moo! Call me " + animal.name);
    }
}
```

**THEO:** Let me challenge you a bit: what is the fundamental difference between OOP polymorphism and a switch statement?

**DAVE:** What do you mean?

**THEO:** I could, for instance, represent an animal with a map having two fields `name` and `type` and calling a different piece of code depending on the value of `type`.

Theo pulls his laptop from its bag, and quickly types in the example as in [13.2](#).

### Listing 13.2 A switch case where behavior depends on type

```
function greet(animal) {
  switch (animal.type) {
    case "dog":
      console.log("Woof Woof! My name is: " + animal.name);
      break;
    case "cat":
      console.log("Meow! I am: " + animal.name);
      break;
    case "cow":
      console.log("Moo! Call me " + animal.name);
      break;
  }
}
```

**DAVE:** How would animals look exactly?

**THEO:** Like I just said, a map with two fields: `name` and `type`, like this.

After more typing, Theo shows Dave his code as in [13.3](#).

### Listing 13.3 Representing animals with maps

```
var myDog = {
  "type": "dog",
  "name": "Fido"
};

var myCat = {
  "type": "cat",
  "name": "Milo"
};

var myCow = {
  "type": "cow",
  "name": "Clarabelle"
};
```

**DAVE:** Could you have given another name to the field that holds the animal type?

**THEO:** Absolutely. It could be anything.

**DAVE:** I see. And you're asking me the fundamental difference between your code with a switch statement and my code with an interface and 3 classes?

**THEO:** Exactly.

**DAVE:** First of all, if you pass an invalid map to your `greet` function, bad things will happen.

**THEO:** You're right. Let me fix that and validate input data.

Theo shares his edits as in [13.4](#).

### Listing 13.4 Data validation

```

var animalSchema = {
  "type": "object",
  "properties": {
    "name": {"type": "string"},
    "type": {"type": "string"}
  },
  "required": ["name", "type"],
};

function greet(animal) {
  if(dev()) { ①
    if(!ajv.validate(animalSchema, animal)) {
      var errors = ajv.errorsText(ajv.errors);
      throw ("greet called with invalid arguments: " + errors);
    }
  }
  switch (animal.type) {
  case "dog":
    console.log("Woof Woof! My name is: " + animal.name);
    break;
  case "cat":
    console.log("Meow! I am: " + animal.name);
    break;
  case "cow":
    console.log("Moo! Call me " + animal.name);
    break;
  }
}

```

- ① See the chapter about data validation for details

**WARNING** You should not use `switch` statements like this in your production code. We use them here for didactic purpose only, as a step towards distilling the essence of polymorphism.

**DAVE:** Another drawback of your approach is that when you want to modify the implementation of `greet` for a specific animal, you have to change the code that deals with all the animals. While in my approach, you could change only a specific animal class.

**THEO:** I agree, and I could also fix that by having a separate function for each animal. Like this.

Theo codes [13.5](#) and shows it to Dave.

### Listing 13.5 Different implementations in different functions

```

function greetDog(animal) {
    console.log("Woof Woof! My name is: " + animal.name);
}

function greetCat(animal) {
    console.log("Meow! I am: " + animal.name);
}

function greetCow(animal) {
    console.log("Moo! Call me " + animal.name);
}

function greet(animal) {
    if(dev()) {
        if(!ajv.validate(animalSchema, animal)) {
            var errors = ajv.errorsText(ajv.errors);
            throw ("greet called with invalid arguments: " + errors);
        }
    }
    switch (animal.type) {
    case "dog":
        greetDog(animal);
        break;
    case "cat":
        greetCat(animal);
        break;
    case "cow":
        greetCow(animal);
        break;
    }
}

```

**DAVE:** But what if you want to extend the functionality of `greet` and add a new animal?

**THEO:** Now you got me. I admit that with a switch statement I can't add a new animal without modifying the original code, while in OOP you can add a new class without having to modify the original code.

**DAVE:** You helped me to realize that the main benefit of polymorphism is that it makes the code easily extensible.

**TIP**

The main benefit of polymorphism is extensibility.

**THEO:** I'm going to ask Joe if there's a way to benefit from polymorphism without objects.

Theo sends a message to Joe and asks him about polymorphism in DOP. He answers that he doesn't have time to get into a deep response as he is in a tech conference where he is about to give a talk about Data-Oriented programming. The only thing he has time to tell Theo is that he should take a look at multimethods.

Theo and Dave read some online material about multimethods. It doesn't look too complicated. They decide that after lunch they will give multimethods a try.

### 13.3 Multimethods with single dispatch

During lunch, Theo asks Dave how it feels to have grown up in the country. He starts with an enthusiastic description about being in direct contact with nature and living a simpler life than in the city. But he admits that country life can sometimes be hard without the conveniences of the city. But who said simple was easy?

After lunch, they decide to have coffee. Dave asks Theo if he'd like to grind the coffee beans himself. Theo accepts with joy. Next, Dave explains how to use a French press coffee maker to get the ideal trade-off between bitterness and rich taste.

While savoring their French press coffee in the garden, Theo and Dave continue their exploration of Polymorphism à la DOP.

**THEO:** It seems that multimethods are a software construct that provide polymorphism without the need for objects.

**DAVE:** I don't get how that's possible.

**THEO:** Multimethods have two parts: A dispatch function and a set of methods that provide an implementation for each dispatched value.

**DAVE:** Is a dispatch function like an interface?

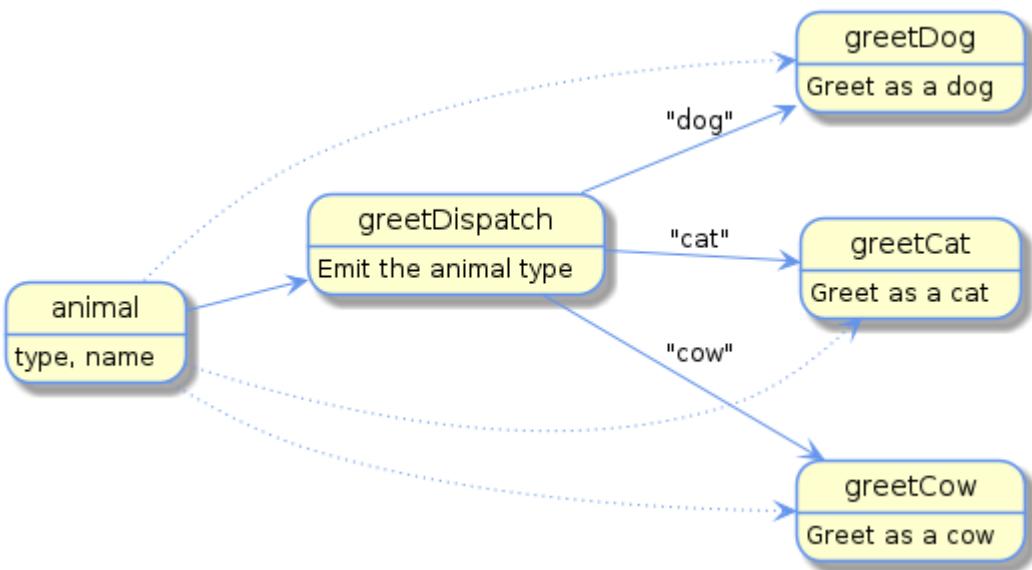
**THEO:** It's like an interface in the sense that it defines the way the function needs to be called. But it goes beyond that. It also dispatches a value that differentiates between the different implementations.

**DAVE:** That's a bit abstract for me.

After reading a couple of articles about multimethods...

**THEO:** I think I understand how to implement the animal greeting capabilities using a multimethod called `greet`. We need a dispatch function and 3 methods. Let's call the dispatch function `greetDispatch`: it dispatches the animal `type`, either "dog", "cat" or "cow". And each dispatch value is handled by a specific method: "dog" by `greetDog`, "cat" by `greetCat` and "cow" by `greetCow`.

Theo takes a napkin and draws a diagram like the one in [13.1](#).



**Figure 13.1** The logic flow of the `greet` multimethod

**DAVE:** Why is there an arrow between `animal` and the methods in addition to the arrow between `animal` and the dispatch function?

**THEO:** Because the arguments of a multimethod are passed to the dispatch function and to the methods.

<b>TIP</b>	The arguments of a multimethod are passed to the dispatch function and to the methods.
------------	--

**DAVE:** Arguments plural?! I see only a single argument.

**THEO:** You're right. Right now our multimethod only receives a single argument, but soon it will receive several arguments.

**DAVE:** I see. Could you show me how to write the code for the `greet` multimethod?

**THEO:** For that, we need a library. For instance, in JavaScript, the `arrows/multimethod` library (<https://github.com/caderek/arrows/tree/master/packages/multimethod>) provides an implementation of multimethods. Basically, we call `multi` to create a multimethod `method` to add a method.

**DAVE:** Where should we start?

**THEO:** We'll start with multimethod initialization by creating a dispatch function `greetDispatch` that defines the signature of the multimethod, validates the arguments, and emits the type of the animal. Then we'll pass `greetDispatch` to `multi` in order to create the `greet` multimethod, like this.

Theo codes the dispatch function as in [13.6](#).

### **Listing 13.6 The dispatch function for `greet` multimethod**

```
function greetDispatch(animal) {    ①
  if(dev()) {
    if(!ajv.validate(animalSchema, animal)) {    ②
      var errors = ajv.errorsText(ajv.errors);
      throw ("greet called with invalid arguments: " + errors);
    }
  }

  return animal.type;    ③
}

var greet = multi(greetDispatch);    ④
```

- ① Signature definition
- ② Argument validation
- ③ Dispatch value
- ④ Multimethod initialization

**TIP**

A multimethod dispatch function is responsible for 3 things: It defines the signature of the multimethod, it validates the arguments and it emits a dispatch value.

**DAVE:** What's next?

**THEO:** Now we need to implement a method for each dispatched value. Let's start with the method that deals with dogs. We create a `greetDog` function that receives an `animal` and add a dog method to the `greet` multimethod using the `method` function from `arrows/multimethod`. The `method` function receives two arguments: the dispatched value and a function that corresponds to the dispatch value. It looks like this.

Theo quickly codes the `greet` method, as in [13.7](#), and shows Dave.

### **Listing 13.7 Implementation of `greet` method for dogs**

```
function greetDog(animal) {    ①
  console.log("Woof woof! My name is " + animal.name);
}
greet = method("dog", greetDog)(greet);    ②
```

- ① method implementation
- ② method declaration

**DAVE:** Does the method implementation have to be in the same module as the multimethod initialization?

**THEO:** Not at all! Method declarations are decoupled from multimethod initialization, exactly like class definitions are decoupled from the interface definition. That's what make multimethods extensible!

**TIP**

Multimethods provides extensibility by decoupling between multimethod initialization and method implementations.

**DAVE:** What about cats and cows?

**THEO:** We add their method implementations like we did for dogs. Here.

Theo codes up 2 more `greet` methods, as in [13.8](#) and [13.9](#).

**Listing 13.8 Implementation of `greet` method for cats**

```
function greetCat(animal) {
  console.log("Meow! I am " + animal.name);
}

greet = method("cat", greetCat)(greet);
```

**Listing 13.9 Implementation of `greet` method for cows**

```
function greetCow(animal) {
  console.log("Moo! Call me " + animal.name);
}

greet = method("cow", greetCow)(greet);
```

**TIP**

In the context of multimethods, a method is a function that provides an implementation for a dispatch value.

**DAVE:** Are the names of dispatch functions and methods important?

**THEO:** Not really. But I like to follow a simple naming convention: use the name of the multimethod (e.g `greet`) as a prefix for the dispatch function (e.g. `greetDispatch`) and the methods, and have the `Dispatch` suffix for the dispatch function and a specific suffix for each method (e.g. `greetDog`, `greetCat` and `greetCow`).

**DAVE:** How does the multimethod mechanism work under the hood?

**THEO:** Internally, a multimethod maintains a hash map where the keys are the dispatched values and the values are the methods. When we add a method, an entry is added to the hash map and when we call the multimethod we query the hash map to find the implementation that corresponds to the dispatched value.

**DAVE:** I don't think you've told me yet how to call a multimethod.

**THEO:** We call it as a regular function, like this.

Theo types the code as in [13.10](#) and shows Dave.

### Listing 13.10 Calling a multimethod like a regular function

```
greet(myDog);
greet(myCat);
greet(myCow);
// "Woof woof! My name is Fido"
// "Meow! I am Milo"
// "Moo! Call me Clarabelle"
```

**TIP**

Multimethods are called like regular functions.

**DAVE:** You told me earlier that in the dispatch function we should validate the arguments. Is that mandatory or is it a best practice?

**THEO:** It's a best practice.

**DAVE:** What happens if the dispatch function doesn't validate the arguments and we pass an invalid argument?

**THEO:** Like when an animal has no corresponding method?

**DAVE:** Exactly!

**THEO:** In that case, you'll get an error. For instance, the `arrows/multimethods` library throws a `NoMethodError` exception.

**DAVE:** That's annoying. Is there a way to provide a default implementation?

**THEO:** Absolutely! In order to define a default implementation, you pass to `method`—as a single argument—the function that provides the default implementation. Like this.

Theo writes the code as in [13.11](#) and shows Dave. Dave tests it as in [13.12](#) and seems satisfied with the result.

### Listing 13.11 Defining a default implementation

```
function greetDefault(animal) {
  console.log("My name is " + animal.name);
}
greet = method(greetDefault)(greet);
```

### Listing 13.12 Calling a multimethod when no method corresponds to the dispatch value

```
var myHorse = {
    "type": "horse",
    "name": "Horace"
};
greet(myHorse);
// "My name is Horace"
```

**DAVE:** Cool!

**TIP**

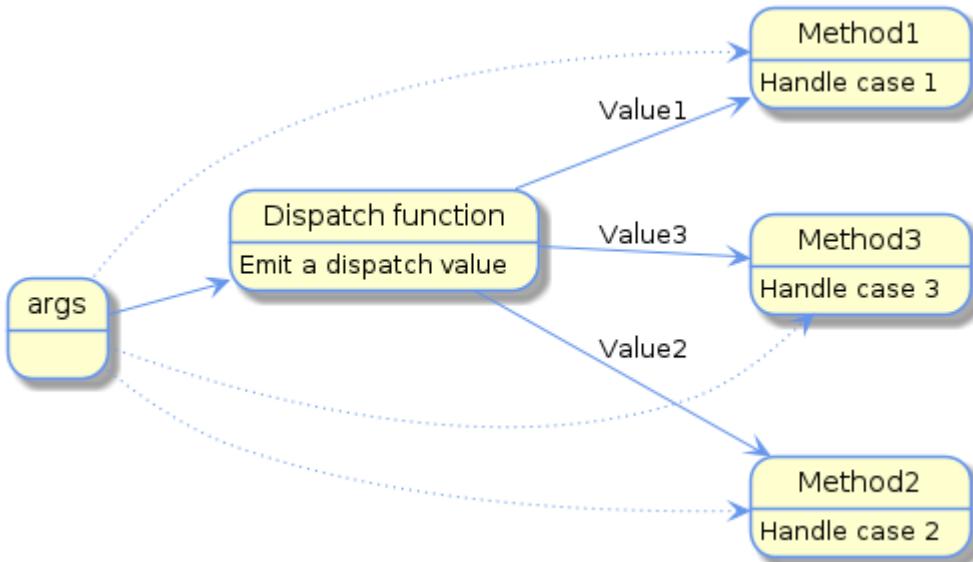
Multimethods support default implementations that are called when no method corresponds to the dispatch value.

## 13.4 Multimethods with multiple dispatch

**THEO:** So far, we've mimicked OOP by having the type of the multimethod argument as a dispatch value. But if you think again about the flow of a multimethod, you'll discover something interesting. Would you like to try and draw a diagram that describes the flow of a multimethod in general?

**DAVE:** Let me bring a fresh napkin. The one under my glass is a bit wet.

It takes Dave a few minutes to draw a diagram like the one in [13.2](#).



**Figure 13.2** The logic flow of multimethods

**THEO:** Excellent! I hope you see that the dispatch function can emit any value.

**DAVE:** Like what?

**THEO:** Like emitting the type of two arguments!

**DAVE:** What do you mean?

**THEO:** Imagine that our animals are polyglot.

**DAVE:** *Poly* what?

**THEO:** Polyglot comes from the Greek *polis* meaning *much* and *glōssa* meaning *language*. A polyglot is a person speaking many languages.

**DAVE:** What languages would our animals speak?

**THEO:** I don't know. Let's say English and French.

**DAVE:** OK. And how would we represent a language in our program?

**THEO:** With a map of course!

**DAVE:** What fields would we have in a language map?

**THEO:** Let's keep things simple and have two fields: `type` and `name`.

**DAVE:** Like an animal map?

**THEO:** Not exactly, in a language map, the `type` field must be either `fr` or `en` while in the animal map the `type` field is either `dog`, `cat` or `cow`.

**DAVE:** Let me write down the language map schema and two language maps.

Dave writes code as in [13.13](#) and [13.14](#) and shows Theo.

### Listing 13.13 The schema of a language map

```
var languageSchema = {
  "type": "object",
  "properties": {
    "name": {"type": "string"},
    "type": {"type": "string"}
  },
  "required": [ "name", "type" ],
};
```

### Listing 13.14 Two language maps

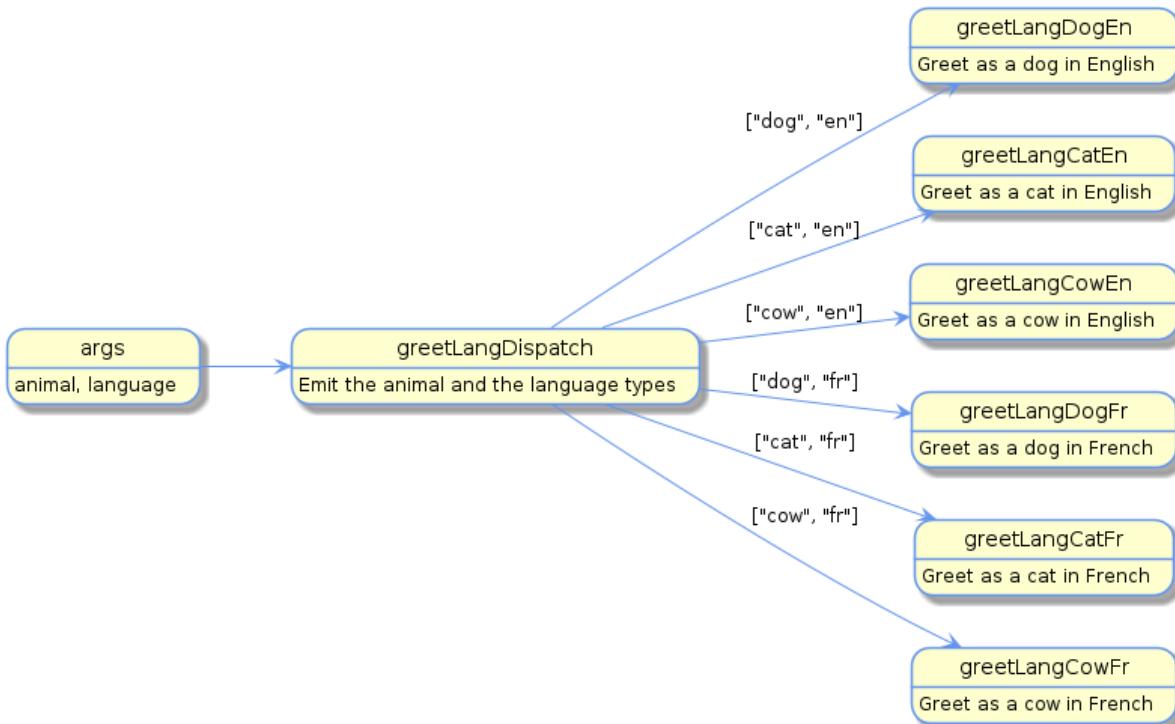
```
var french = {
    "type": "fr",
    "name": "Français"
};

var english = {
    "type": "en",
    "name": "English"
};
```

**THEO:** Excellent. Now, let's write the code for the dispatch function and the methods for our polyglot animals. Let's call our multimethod: `greetLang`. We have one dispatch function and 6 methods.

**DAVE:** Right. 3 animals (dog, cat, cow) times 2 languages (en, fr). But before the implementation I'd like to draw a flow diagram. It will help me to make things crystal clear.

Dave picks up another napkin and he draws a diagram like the one in [13.3](#).



**Figure 13.3** The logic flow of the `greetLang` multimethod

**THEO:** Why did you omit the arrow between the arguments and the methods?

**DAVE:** In order to keep the diagram readable. Otherwise there would be too many arrows.

**THEO:** OK. Are you ready for coding?

**DAVE:** Yes.

**THEO:** The dispatch function needs to validate its arguments and return an array with two elements: the type of the animal and the type of the language.

Theo types for a bit on his laptop and produces code as in [13.15](#). He shows Dave.

**Listing 13.15 Multimethod initialization with a dispatch function that return the type of its arguments**

```
var greetLangArgsSchema = {
  "type": "array",
  "prefixItems": [animalSchema, languageSchema]
};

function greetLangDispatch(animal, language) {
  if(dev()) {
    if(!ajv.validate(greetLangArgsSchema, [animal, language])) {
      throw ("greetLang called with invalid arguments: " + ajv.errorsText(ajv.errors));
    }
  }
  return [animal.type, language.type];
};

var greetLang = multi(greetLangDispatch);
```

**DAVE:** Does the order of the elements in the array matter?

**THEO:** It doesn't matter but it needs to be consistent with the wiring of the methods. The implementation of the `greetLang` looks like this.

Theo works a bit and produces the code as in [13.16](#).

### Listing 13.16 The implementation of greetLang methods

```

function greetLangDogEn(animal, language) {
  console.log("Woof woof! My name is " + animal.name + " and I speak " + language.name);
}

greetLang = method(["dog", "en"], greetLangDogEn)(greetLang);

function greetLangDogFr(animal, language) {
  console.log("Ouaf Ouaf! Je m'appelle " + animal.name + " et je parle " + language.name);
}

greetLang = method(["dog", "fr"], greetLangDogFr)(greetLang);

function greetLangCatEn(animal, language) {
  console.log("Meow! I am " + animal.name + " and I speak " + language.name);
}
greetLang = method(["cat", "en"], greetLangCatEn)(greetLang);

function greetLangCatFr(animal, language) {
  console.log("Miaou! Je m'appelle " + animal.name + " et je parle " + language.name);
}
greetLang = method(["cat", "fr"], greetLangCatFr)(greetLang);

function greetLangCowEn(animal, language) {
  console.log("Moo! Call me " + animal.name + " and I speak " + language.name);
}
greetLang = method(["cow", "en"], greetLangCowEn)(greetLang);

function greetLangCowFr(animal, language) {
  console.log("Meuh! Appelle moi " + animal.name + " et je parle " + language.name);
}
greetLang = method(["cow", "fr"], greetLangCowFr)(greetLang);

```

Dave looks at the code for the methods that deal with French and he is surprised to see `Ouaf` instead of `Woof` `Woof` for dogs, `Miaou` instead of `Meow` for cats and `Meuh` instead of `Moo` for cows.

**DAVE:** I didn't know that animal onomatopoeia were different in French than in English!

**THEO:** *Ono* what?

**DAVE:** Onomatopoeia from the Greek *ónoma* that means *name* and *poiéō* that means *to produce*. It is the property of words that sound like what they represent. For instance, *Woof*, *Meow* and *Moo*.

**THEO:** Yeah for some reason in French dogs make *Ouaf*, cats make *Miaou* and cows make *Meuh*.

**TIP**

**Multiple dispatch is when a dispatch function emits a value that depends on more than one argument.**

**DAVE:** I see that in the array the animal type is always before the language type.

**THEO:** Right. As I told you before, in a multimethod that features multiple dispatch, the order doesn't really matter, but it has to be consistent.

**TIP**

In a multimethod that features multiple dispatch, the order of the elements in the array emitted by the dispatch function has to be consistent with the order of the elements in the wiring of the methods.

**DAVE:** Now, let me see if I can figure out how to use a multimethod that features multiple dispatch.

Dave remembers that Theo told him earlier that multimethods are used like regular functions and he comes up with the code in [13.17](#).

**Listing 13.17 Calling a multimethod that features multiple dispatch**

```

greetLang(myDog, french);
greetLang(myDog, english);
greetLang(myCat, french);
greetLang(myCat, english);
greetLang(myCow, french);
greetLang(myCow, english);
// "Ouaf Ouaf! Je m\'appelle Fido et je parle Français"
// "Woof woof! My name is Fido and I speak English"
// "Miaou! Je m\'appelle Milo et je parle Français"
// "Meow! I am Milo and I speak English"
// "Meuh! Appelle moi Clarabelle et je parle Français"
// "Moo! Call me Clarabelle and I speak English"

```

**THEO:** Do you agree that multimethods with multiple dispatch offer a more powerful polymorphism than OOP polymorphism?

**DAVE:** Indeed.

**THEO:** Now, let me show you an even more powerful polymorphism called dynamic dispatch.

## 13.5 Multimethods with dynamic dispatch

**DAVE:** What is dynamic dispatch?

**THEO:** It's when the dispatch function of a multimethod returns a value that goes beyond the static type of its arguments.

**DAVE:** Like what?

**THEO:** Like for instance a number or a boolean.

**DAVE:** Why such a thing would be useful?

**THEO:** Imagine that instead of being polyglot our animals would suffer from dysmakrylexia.

**DAVE:** Suffering from what?

**THEO:** *Dysmakrylexia* comes from the Greek *dus* expressing the idea of *difficulty*, *makrys* meaning *long* and *léxis* that means *diction*. Therefore, Dysmakrilexia is difficulty pronouncing long words.

**DAVE:** I've never heard of that.

**THEO:** That's because I just invented it...

**DAVE:** Funny. What's considered a long word for our animals?

**THEO:** Let's say that when their name has more than 5 letters they're not able to say it.

**DAVE:** A bit weird, but OK.

**THEO:** Let's call our multimethod `dysGreet`. Its dispatch function returns an array with two elements: the animal type and a boolean about whether the name is long or not. Take a look.

Theo types for a bit and then shows Dave his code as in [13.18](#).

### Listing 13.18 Multimethod initialization with a dispatch function that does a dynamic dispatch

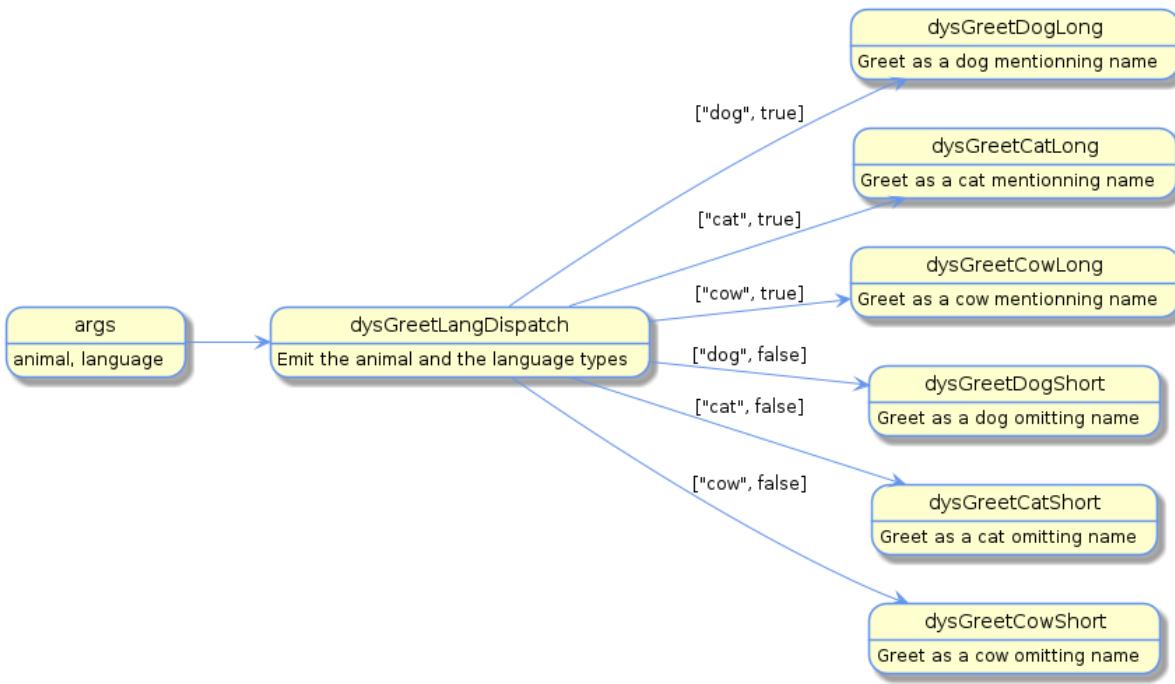
```
function dysGreetDispatch(animal) {
  if(dev()) {
    if(!ajv.validate(animalSchema, animal)) {
      var errors = ajv.errorsText(ajv.errors);
      throw ("dysGreet called with invalid arguments: " + errors);
    }
  }
  var hasLongName = animal.name.length > 5;

  return [animal.type, hasLongName];
};

var dysGreet = multi(dysGreetDispatch);
```

**DAVE:** Writing the `dysGreet` methods doesn't seem too complicated.

Dave finds one more clean napkin and draws a flow diagram as in [13.4](#). Then he grabs his laptop and writes the implementation as in [13.19](#).



**Figure 13.4** The logic flow of the `dysGreet` multimethod

### **Listing 13.19** The `dysGreet` methods

```

function dysGreetDogLong(animal) {
  console.log("Woof woof! My name is " + animal.name);
}
dysGreet = method(["dog", true], dysGreetDogLong)(dysGreet);

function dysGreetDogShort(animal) {
  console.log("Woof woof!");
}
dysGreet = method(["dog", false], dysGreetDogShort)(dysGreet);

function dysGreetCatLong(animal) {
  console.log("Meow! I am " + animal.name);
}
dysGreet = method(["cat", true], dysGreetCatLong)(dysGreet);

function dysGreetCatShort(animal) {
  console.log("Meow!");
}
dysGreet = method(["cat", false], dysGreetCatShort)(dysGreet);

function dysGreetCowLong(animal) {
  console.log("Moo! Call me " + animal.name);
}
dysGreet = method(["cow", true], dysGreetCowLong)(dysGreet);

function dysGreetCowShort(animal) {
  console.log("Moo!");
}
dysGreet = method(["cow", false], dysGreetCowShort)(dysGreet);
  
```

After checking that the code works as expected in [13.20](#), Theo compliments Dave.

**Listing 13.20 Testing dysGreet**

```
dysGreet(myDog);
dysGreet(myCow);
dysGreet(myCat);
// "Woof woof!"
// "Moo! Call me Clarabelle"
// "Meow!"
```

**THEO:** Well done, my friend! Our exploration of multimethods has come to an end. And I think it's time for me to drive back if I want to get home before night.

**DAVE:** Before you leave, let's check if multimethods are available in programming languages other than JavaScript.

**THEO:** That's a question for Joe.

**DAVE:** Do you think it's OK if I call him now?

**THEO:** I think it's better if you send him an email. Thank you for this beautiful day in the country.

**DAVE:** I enjoyed it also. Especially our discussions about etymology.

After a few minutes Dave receives an email from Joe with the subject: Support for multimethods in different language.

**SIDE BAR****Support for multimethods in different languages**

In Python, there is a library called multimethods (<https://github.com/weissjeffm/multimethods>) and in Ruby there is Ruby multimethods (<https://github.com/psantacruz/ruby-multimethods>). Both seem to work quite like JavaScript arrows/multimethod.

In Java, there is the Java Multimethod Framework (<http://igm.univ-mlv.fr/~forax/works/jmmf/>) and C# supports multimethods natively via the `dynamic` keyword. However, in both cases, it works only with static data types and not with generic data structures.

Language	URL	Generic data structure support
JavaScript	<a href="https://github.com/caderek/arrows/tree/master/packages/multimethod">https://github.com/caderek/arrows/tree/master/packages/multimethod</a>	Yes
Java	<a href="http://igm.univ-mlv.fr/~forax/works/jmmf/">http://igm.univ-mlv.fr/~forax/works/jmmf/</a>	No
C#	Native support	No
Python	<a href="https://github.com/weissjeffm/multimethods">https://github.com/weissjeffm/multimethods</a>	Yes
Ruby	<a href="https://github.com/psantacruz/ruby-multimethods">https://github.com/psantacruz/ruby-multimethods</a>	Yes

## 13.6 Integrating multimethods in a production system

While Theo is driving back home his thoughts take him back to the fresh air of the country. This pleasant moment is interrupted by a phone call from Nancy.

**NANCY:** How are you doing?

**THEO:** Fine. I'm driving back from the countryside.

**NANCY:** Cool. Are you available to talk about work?

**THEO:** Sure.

**NANCY:** I'd like to add a tiny feature to the catalog.

In the past, when Nancy qualified a feature as *tiny*, Theo was scared. What seemed easy for her always took him a surprising amount of time to develop. But after refactoring the system according to DOP principles, what seems tiny to Nancy usually is quite easy to implement.

**THEO:** What feature?

**NANCY:** I'd like to allow librarians to view the list of authors, ordered by last name, in two

formats: HTML and Markdown.

**THEO:** It doesn't sound too complicated.

**NANCY:** Also, I need a bit of text formatting.

**THEO:** What kind of text formatting?

**NANCY:** Depending on the number of books a author has written, their name should be in bold and italic fonts.

**THEO:** Could you send me an email with all the details. I'll take a look at it tomorrow morning.

**NANCY:** Perfect. Have a safe drive!

Before going to bed, Theo reflects about today's etymology lessons and he realizes that he never looked for the etymology of the word *etymology* itself! He searches for the term *etymology* in Wikipedia and he learns that the word *etymology* derives from the Greek *étumon*, meaning *true sense*, and the suffix *logia*, denoting *the study of*. During the night, Theo dreams of dogs, cats and cows programming in a field of grass on their laptops.

When Theo arrives at the office, he opens Nancy's email with the details about the text formatting feature. The details are summarized in the table in [13.1](#).

**Table 13.1 Text formatting for author names according to the number of books they have written**

Number of books	Italic	Bold
10 or fewer	Yes	No
Between 11 and 50	No	Yes
51 or more	Yes	Yes

Theo forwards Nancy's email to Dave and asks him to take care of this task.

The most difficult part of the feature lies in implementing an `Author.myName(author, format)` function that receives two arguments: the author data and the text format. Dave asks himself whether he could implement this function as a multimethod and leverage what he learned yesterday with Theo in the country. It seems that this feature is quite similar to the one that dealt with dysmakrilexia. Instead of checking the length of a string, he needs to check the length of an array.

First of all, Dave needs a data schema for the text format. He could represent a format as a map with a `type` field like Theo did yesterday for languages, but at the moment it seems simpler to represent a format as a string that could be either `markdown` or `html`. The text format schema is in [13.21](#). He already wrote the author schema with Theo last week. It's in [13.22](#).

### Listing 13.21 The text format schema

```
var textFormatSchema = {
  "name": {"type": "string"},
  "type": {"enum": ["markdown", "html"]}
};
```

### Listing 13.22 The author schema

```
var authorSchema = {
  "type": "object",
  "required": ["name", "bookIsbns"],
  "properties": {
    "name": {"type": "string"},
    "bookIsbns": {
      "type": "array",
      "items": {"type": "string"}
    }
  }
};
```

Now Dave needs to write a dispatch function and initialize the multimethod. Remembering that Theo had no qualms about creating the word "dysmakrylexia", he decides that he prefers his own neologism "prolificity" over the existing nominal form "prolificness", and so finds it useful to have an `Author.prolificityLevel` helper function that returns the level of prolificity of the author: either `low`, `medium` or `high`. The `authorNameDispatch` function is in [13.23](#).

### Listing 13.23 `Author.myName` multimethod initialization

```
Author.prolificityLevel = function(author) {
  var books = _.size(_.get(author, "bookIsbns"));
  if (books <= 10) {
    return "low";
  };
  if (books >= 51) {
    return "high";
  }
  return "medium";
};

var authorNameArgsSchema = {
  "type": "array",
  "prefixItems": [
    authorSchema,
    {"enum": ["markdown", "html"]}
  ]
};

function authorNameDispatch(author, format) {
  if(dev()) {
    if(!ajv.validate(authorNameArgsSchema, [author, format])) {
      throw ("Author.myName called with invalid arguments: " + ajv.errorsText(ajv.errors));
    }
  }

  return [Author.prolificityLevel(author), format];
};

Author.myName = multi(authorNameDispatch);
```

Then Dave works on the methods. First the HTML format methods. In HTML, bold text is wrapped inside a **<b>** tag and italic text is wrapped in a *<i>* tag. For instance, in HTML, three authors with different levels of prolificity would be written like in [13.24](#).

### **Listing 13.24 Examples of bold and italic in HTML**

```
<i>Yehonathan Sharvit<i>      ①
<b>Stephen Covey</b>          ②
<b><i>Isaac Asimov</i></b>    ③
```

- ① Italic formatting for minimally prolific authors
- ② Bold formatting for moderately prolific authors
- ③ Bold and italic formatting for highly prolific authors

With this information in mind, Dave writes the three methods that deal with HTML formatting. The result is in [13.25](#).

### **Listing 13.25 The methods that deal with HTML formatting**

```
function authorNameLowHtml(author, format) {
  return "<i>" + _.get(author, "name") + "</i>";
}

Author.myName = method(["low", "html"], authorNameLowHtml)(Author.myName);

function authorNameMediumHtml(author, format) {
  return "<b>" + _.get(author, "name") + "</b>";
}

Author.myName = method(["medium", "html"], authorNameMediumHtml)(Author.myName);

function authorNameHighHtml(author, format) {
  return "<b><i>" + _.get(author, "name") + "</i></b>";
}

Author.myName = method(["high", "html"], authorNameHighHtml)(Author.myName);
```

Then, Dave moves on to the three methods that deal with Markdown formatting. In Markdown, bold text is wrapped in two asterisks and italic text is wrapped in a single asterisk. For instance, in Markdown, three authors with different levels of prolificity would be written like in [13.26](#). The code of the markdown methods is in [13.27](#).

### **Listing 13.26 Examples of bold and italic in Markdown**

```
*Yehonathan Sharvit*      ①
**Stephen Covey**        ②
***Isaac Asimov***       ③
```

- ① Italic formatting for minimally prolific authors
- ② Bold formatting for moderately prolific authors
- ③ Bold and italic formatting for highly prolific authors

### Listing 13.27 The methods that deal with Markdown formatting

```
function authorNameLowMarkdown(author, format) {
    return "*" + _.get(author, "name") + "*";
}

Author.myName = method(["low", "markdown"], authorNameLowMarkdown)(Author.myName);

function authorNameMediumMarkdown(author, format) {
    return "***" + _.get(author, "name") + "***";
}

Author.myName = method(["medium", "markdown"], authorNameMediumMarkdown)(Author.myName);

function authorNameHighMarkdown(author, format) {
    return "****" + _.get(author, "name") + "****";
}

Author.myName = method(["high", "markdown"], authorNameHighMarkdown)(Author.myName);
```

Dave tests his code by involving a *mysterious* author as in [13.28](#) and [13.29](#).

### Listing 13.28 Testing HTML formatting

```
var yehonathan = {
    "name": "Yehonathan Sharvit",
    "bookIsbns": [ "9781617298578" ] // Real ISBN of "Data-Oriented Programming"
};

Author.myName(yehonathan, "html");
//<i>Yehonathan Sharvit</i>"
```

### Listing 13.29 Testing Markdown formatting

```
Author.myName(yehonathan, "markdown");
// *Yehonathan Sharvit*"
```

Dave sends Theo an email asking for a review of the pull request for the "list of authors" feature.

The only comment Theo makes about Dave's code relates to the author that appears in the test of `Author.myName` in [13.28](#).

**THEO:** Who is Yehonathan Sharvit?

**DAVE:** A character from a science-fiction book I am reading.

## 13.7 Summary

- The main benefit of polymorphism is extensibility.
- Multimethods make it possible to benefit from polymorphism when data is represented with generic maps.
- A multimethod is made of a dispatch function and multiple methods.
- The dispatch function of a multimethod emits a dispatch value
- Each of the methods used in a multimethod provides an implementations for a specific dispatch value.
- Multimethods can mimic OOP class inheritance via single dispatch.
- In single dispatch, a multimethod receives a single map that contains a `type` field and the dispatch function of the multimethod emits the value of the `type` field.
- In addition to single dispatch, multimethods provide two kinds of advanced polymorphisms: multiple dispatch and dynamic dispatch.
- Multiple dispatch is used when the behavior of the multimethod depends on multiple arguments
- Dynamic dispatch is used when the behavior of the multimethod depends on run-time arguments
- The arguments of a multimethod are passed to the dispatch function and to the methods.
- A multimethod dispatch function is responsible for: 1. Defining the signature 2. Validating the arguments 3. Emitting a dispatch value.
- Multimethods provides extensibility by decoupling between multimethod initialization and method implementations.
- Multimethods are called like regular functions.
- Multimethods support default implementations that are called when no method corresponds to the dispatch value.
- In a multimethod that features multiple dispatch, the order of the elements in the array emitted by the dispatch function has to be consistent with the order of the elements in the wiring of the methods.

**Table 13.2 Support for multimethods in different languages**

Language	URL	Generic data structure support
JavaScript	<a href="https://github.com/caderek/arrows/tree/master/packages/multimethod">https://github.com/caderek/arrows/tree/master/packages/multimethod</a>	Yes
Java	<a href="http://igm.univ-mlv.fr/~forax/works/jmmf/">http://igm.univ-mlv.fr/~forax/works/jmmf/</a>	No
C#	Native support	No
Python	<a href="https://github.com/weissjeffm/multimethods">https://github.com/weissjeffm/multimethods</a>	Yes
Ruby	<a href="https://github.com/psantac/ruby-multimethods">https://github.com/psantac/ruby-multimethods</a>	Yes

**Table 13.3 Lodash functions introduced in this chapter**

Function	Description
<code>size(coll)</code>	Gets the size of <code>coll</code>

# 14

## *Advanced data manipulation*

### This chapter covers

- Manipulating nested data
- Writing clear and concise business logic code
- Separation between business logic and generic data manipulation
- Building custom data manipulation tools
- Using the best tool for the job

### 14.1 Whatever is well conceived is clearly said

When our business logic involves advanced data processing, the generic data manipulation functions provided by the language run-time and by third-party libraries might not be sufficient. Instead of mixing the details of data manipulation with business logic, we can write our own generic data manipulation functions and implement our custom business logic using them.

Separating business logic from internal details of the data manipulation makes the business logic code concise and easy to read for other developers.

### 14.2 Updating a value in a map with eloquence

Dave is more and more autonomous on the Klafim project. He can implement most features on his own, typically turning to Theo only for code reviews. Dave's code quality standards are quite high. Even when his code is functionally solid, he tends to be unsatisfied with its readability. This time, he asks for Theo's help in improving the readability of the code that fixes a bug that Theo introduced a long time ago.

**DAVE:** I think I have found a bug in the code that returns book information from OpenLibrary

API.

**THEO:** What bug?

**DAVE:** Sometimes the API returns duplicate author names and we pass the duplicates through to the client.

**THEO:** It doesn't sound like a complicated bug to fix.

**DAVE:** Right. I fixed it but I'm not satisfied with the readability of the code I wrote.

**THEO:** Being able to be critical of our own code is an important quality for a developer to progress. What is it exactly that you don't like?

**DAVE:** Take a look at this code.

Dave points to code on his screen as in [14.1](#).

**Listing 14.1 The code that removes duplicates in a straightforward but tedious way.**

```
function removeAuthorDuplicates(book) {
  var authors = _.get(book, "authors");
  var uniqAuthors = _.uniq(authors);
  return _.set(book, "authors", uniqAuthors);
}
```

**DAVE:** I'm using `get` to retrieve the array with the author names, then `uniq` to create a duplicate-free version of the array, and finally `set` to create a new version of the book with no duplicate author names.

**THEO:** The code is tedious because the next value of `authorNames` needs to be based on its current value.

**DAVE:** But it's a common use case! Isn't there a simpler way to write this kind of code?

**THEO:** Your astonishment definitely honors you as a developer, Dave! I agree with you that there must be a simpler way. Let me call Joe.

*Over the phone.*

**JOE:** How's it going?

**THEO:** Great. Are you back from your tech conference?

**JOE:** I just landed. I'm on my way home in a taxi.

**THEO:** How was your talk about Data-Oriented Programming?

**JOE:** Pretty good. At the beginning people were a bit suspicious. But when I told them the story of Albatross and Klafim, it was quite convincing.

**THEO:** Yeah. Adults are like children in that way: they love stories.

**JOE:** What about you? Did you manage to achieve polymorphism with multimethods?

**THEO:** Yes. Dave even managed to implement a feature in Klafim with multimethods.

**JOE:** Cool!

**THEO:** Do you have time to help Dave with a question about programming?

**JOE:** Sure.

*You pass your phone to Dave.*

**DAVE:** Hi Joe. How are you doing?

**JOE:** Hello Dave. Not bad. What kind of help do you need?

**DAVE:** I'm wondering if there's a simpler way to remove duplicates inside an array value in a map. Using `get`, `uniq` and `set` looks quite tedious.

**JOE:** You should build your own data manipulation tools.

**DAVE:** What do you mean?

**JOE:** You should write a generic `update` function<sup>10</sup> that updates a value in a map, applying a calculation based on its current value.

**DAVE:** What would the arguments of `update`, be in your opinion?

**JOE:** Put the cart before the horse.

**DAVE:** What?!

**JOE:** Rewrite your business logic as if `update` were already implemented and you'll discover what the arguments of `update` should be.

**DAVE:** I see what you mean: The *horse* is the implementation of `update` and the *cart* is the usage of `update`.

**JOE:** Exactly. But remember: It's better if you keep your `update` function generic.

**DAVE:** How?

**JOE:** By not being limited to your specific use case.

**DAVE:** I see. The implementation of `update` should not deal with removing duplicate elements. Instead, it should receive the updating function (in my case `uniq`) as an argument.

**JOE:** Exactly! Uh, sorry Dave, I gotta go, I just got home. Good luck!

**DAVE:** Take care, Joe!

*Dave hangs up and gives Theo his phone back.*

**DAVE:** Joe advised me to write my own `update` function. For that purpose, he told me to start by rewriting `removeAuthorDuplicates` as if `update` were already implemented. It will allow us to make sure we get the signature of `update` right.

**THEO:** Sounds like a plan.

**DAVE:** Joe called it "putting the cart before the horse".

**TIP**

The best way to find the signature of a custom data manipulation function is to think about the most convenient way to use it.

**THEO:** Joe and his funny analogies...

**DAVE:** Anyway, the way I'd like to use `update` inside `removeAuthorDuplicates` is like this.

Dave types the code as in [14.2](#) and shows Theo.

**Listing 14.2 The code that remove duplicates in an elegant way.**

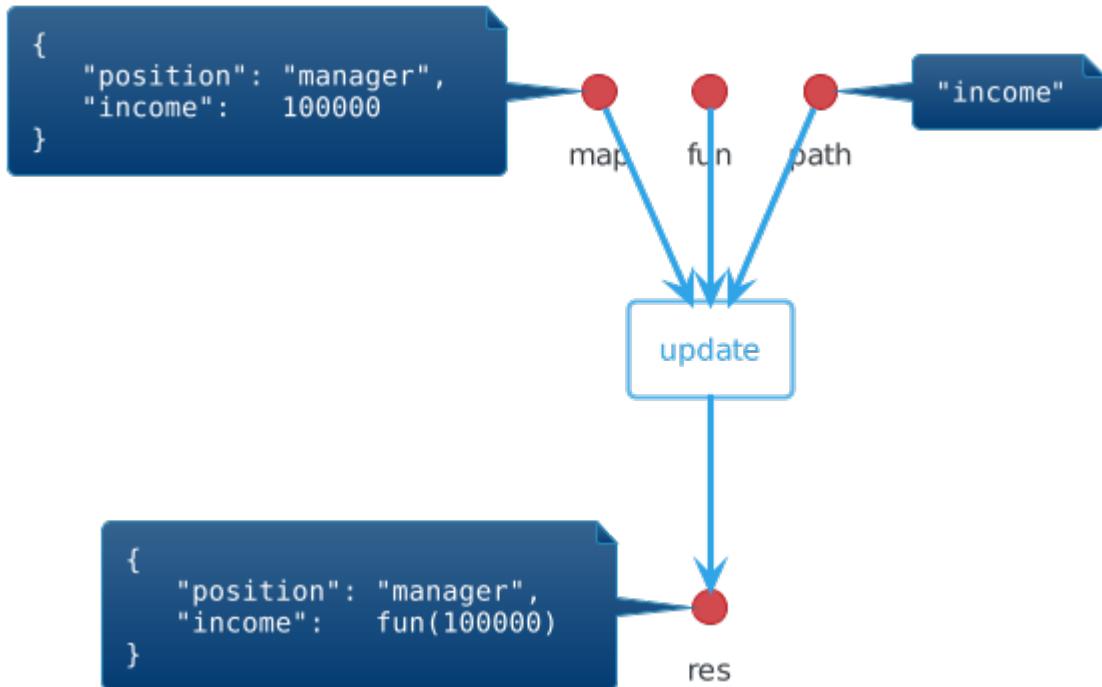
```
function removeAuthorDuplicates(book) {
  return update(book, "authors", _.uniq);
}
```

**THEO:** Looks good to me!

**DAVE:** Wow! The code with `update` is much more elegant than the code with `_.get` and `_.set`!

**THEO:** Before you implement `update`, I suggest that you write down in plain English exactly what the function does.

**DAVE:** It's quite easy: `update` receives a map `map`, a path `path` and a function `fun`. It returns a new version of the map `map` where `path` is associated with `fun(currentValue)`, and where `currentValue` is the value associated with `path` in `map`.



**Figure 14.1** The behavior of `update`

**TIP**

Before implementing a custom data manipulation function, formulate in plain English exactly what the function does.

**THEO:** With such a clear definition, it's going to be a piece of cake to implement `update`!

After a few minutes, Dave comes up with the code in [14.3](#).

**Listing 14.3** A generic `update` function

```
function update(map, path, fun) {
  var currentValue = _.get(map, path);
  var nextValue = fun(currentValue);
  return _.set(map, path, nextValue);
}
```

**THEO:** Why don't you see if it works with a simple case like incrementing a number in a map?

**DAVE:** Good idea! I'll try multiplying a value in a map by 2, with `update` code like this.

Dave types for a bit and shows the code in [14.4](#) to Theo.

### Listing 14.4 Multiplying a value in a map by 2.

```
var m = {
  "position": "manager",
  "income": 100000
};
update(m, "income", function(x) {
  return x * 2;
});
// {"position": "manager",
//  "income": 200000}
```

**THEO:** It seems to work!

## 14.3 Manipulating nested data

The next Monday, during Theo and Dave's weekly sync meeting, they discuss the upcoming features for Klaflim.

**THEO:** Recently, Nancy has been asking for more and more administrative features.

**DAVE:** Like what?

**THEO:** I'll give you a few examples. Let me find the email I got from Nancy yesterday.

**DAVE:** OK.

**THEO:** There are 3 feature requests for now: 1. Listing all the book author ids 2. Calculating the book lending ratio and 3. Grouping books by physical library.

**DAVE:** What feature should I tackle first?

**THEO:** It doesn't matter. But you should deliver the three of them before the end of the week. Good luck. Don't hesitate to call me if you need help.

On Tuesday, Dave asks for Theo's help on the admin features.

**DAVE:** I started to work on the three admin features, but I don't like the code I wrote. Let me show you the code for retrieving the list of author ids from the list of books returned from the database.

**THEO:** Can you remind me what an element in a book list returned from the database looks like?

**DAVE:** Each book is a map with an `authorIds` array field.

**THEO:** OK. So it sounds like a `map` over the books will do it.

**DAVE:** This is what I did but it doesn't work as expected.

Dave shares his code with Theo as in [14.5](#).

#### Listing 14.5 Retrieving the author ids in books as an array of arrays

```
function authorIdsInBooks(books) {
    return _.map(books, "authorIds");
}
```

**THEO:** What's the problem?

**DAVE:** The problem is that it returns an array of arrays of author ids instead of an array of author ids. For instance, when I run `authorIdsInBooks` on a catalog with two books, I get this.

Dave shows his results, as in [14.6](#).

#### Listing 14.6 The author ids in an array of arrays

```
[  
  ["sean-covey", "stephen-covey"],  
  ["alan-moore", "dave-gibbons"]  
]
```

**THEO:** That's not a big problem. You can flatten an array of arrays with `_.flatten` and get the result you expect.

**DAVE:** Nice! This is exactly what I need. Let me fix the code of `authorIdsInBooks` by flattening the result of the mapping. Here you go.

Dave shows Theo his code as in [14.7](#).

#### Listing 14.7 Retrieving the author ids in books as an array of strings

```
function authorIdsInBooks(books) {
    return _.flatten(_.map(books, "authorIds"));
}
```

**THEO:** Don't you think that mapping then flattening deserves a function of its own?

**DAVE:** Maybe. It's quite easy to implement a `flatMap`.<sup>11</sup> function. The code is here.

Dave shows Theo his code as in [14.8](#).

#### Listing 14.8 The implementation of `flatMap`

```
function flatMap(coll, f) {
    return _.flatten(_.map(coll, f));
}
```

**THEO:** Nice!

**DAVE:** I don't know. It's kind of weird to have such a small function.

**THEO:** I don't think that code size is what matters here.

**DAVE:** What do you mean?

**THEO:** See what happens when you rewrite `authorIdsInBooks`, using `flatMap`.

**DAVE:** OK. Here you go.

Dave shows Theo his code as in [14.9](#).

#### Listing 14.9 Retrieving the author ids in books as an array of strings using flatMap

```
function authorIdsInBooks(books) {
    return flatMap(books, "authorIds");
}
```

**THEO:** What implementation do you prefer: The one with `flatten` and `map` (in [14.7](#)) or the one with `flatMap` (in [14.9](#))?

**DAVE:** I don't know. To me, they look quite similar.

**THEO:** Right. But which implementation is more readable?

**DAVE:** Well. Assuming I know what `flatMap` does, I would say the implementation with `flatMap`—being more concise—is a bit more readable.

**THEO:** Again, it's not about the size of the code. It's about the clarity of intent and the power of naming things.

**DAVE:** I don't get that.

**THEO:** Let me give you an example from day to day language.

**DAVE:** OK.

**THEO:** Could you pass me that thing on your desk that's used for writing?

It takes Dave a few seconds to get that Theo has asked him to pass the pen on the desk. After he passes Theo the pen, he asks:

**DAVE:** Why didn't you simply ask for the pen?

**THEO:** I wanted you to experience how it feels when we use descriptions instead of names to convey our intent.

**DAVE:** Oh. I see. You mean that once we use a name for the operation that maps and flattens,

the code becomes clearer.

**THEO:** Exactly.

**DAVE:** Let's move on to the second admin feature.

**THEO:** Before that, I think we deserve a short period for rest and refreshments where we drink a beverage made by percolation from roasted and ground seeds.

**DAVE:** A coffee break!

## 14.4 Use the best tool for the job

After the coffee break, Dave shows Theo his implementation of the book lending ratio calculation. This time, he seems to like the code he wrote.

**DAVE:** I'm quite proud of the code I wrote to calculate the book lending ratio.

**THEO:** Show me the money!

**DAVE:** My function receives a list of books from the database like this.

Dave shows Theo the output as in [14.10](#).

### Listing 14.10 A list of 2 books with bookItems

```
[
  {
    "isbn": "978-1779501127",
    "title": "Watchmen",
    "bookItems": [
      {
        "id": "book-item-1",
        "libId": "nyc-central-lib",
        "isLent": true
      }
    ],
    "isbn": "978-1982137274",
    "title": "7 Habits of Highly Effective People",
    "bookItems": [
      {
        "id": "book-item-123",
        "libId": "hudson-park-lib",
        "isLent": true
      },
      {
        "id": "book-item-17",
        "libId": "nyc-central-lib",
        "isLent": false
      }
    ]
  }
]
```

**THEO:** Quite a nested piece of data!

**DAVE:** Yeah. But using `flatMap`, calculating the lending ratio is quite easy: I'm going over all the book items with `forEach` and incrementing either the `lent` or the `notLent` counter. At the end, I return the ratio between `lent` and `(lent + notLent)`. Here's how I do it.

Dave shows Theo his code as in [14.11](#).

#### Listing 14.11 Calculating the book lending ratio using `forEach`

```
function lendingRatio(books) {
  var bookItems = flatMap(books, "bookItems");
  var lent = 0;
  var notLent = 0;
  _.forEach(bookItems, function(item) {
    if(_.get(item, "isLent")) {
      lent = lent + 1;
    } else {
      notLent = notLent + 1;
    }
  });
  return lent/(lent + notLent);
}
```

**THEO:** Would you allow me to tell you frankly what I think of your code?

**DAVE:** If you are asking this question, it means that you don't like it. Right?

**THEO:** It's nothing against you: I don't like any piece of code with `forEach`.

**DAVE:** What's wrong with `forEach`?

**THEO:** It's too generic!

**DAVE:** I thought that genericity was a positive thing in programming.

**THEO:** It is when we build a utility function but when we use a utility function, we should use the least generic function that solves our problem.

**DAVE:** Why?

**THEO:** Because we ought to choose the right tool for the job, like in the real life.

**DAVE:** What do you mean?

**THEO:** Let me give you an example: yesterday, I had to clean up my drone from the inside. Do you think that I used a screwdriver or a Swiss army knife to unscrew the drone cover?

**DAVE:** A screwdriver of course! It's much more convenient to manipulate.

**THEO:** Right. Also, imagine that someone looks at me using a screwdriver. It's quite clear to

them that I am turning a screw. It conveys my intent clearly.

**DAVE:** Are you saying that `forEach` is like the Swiss army knife of data manipulation?

**THEO:** That's a good way to put it.

**TIP**

Pick the least generic utility function that solves your problem.

**DAVE:** What function should I use then, to iterate over the book item collection?

**THEO:** You could use `reduce`.

**DAVE:** I thought `reduce` was about returning data from a collection. Here I don't need to return data. I need to update two variables: `lent` and `notLent`.

**THEO:** You could represent those two values in a map with two keys.

**DAVE:** Could you show me how to rewrite my `lendingRatio` function using `reduce`?

**THEO:** Sure. The initial value passed to `reduce` is the map `{"lent": 0, "notLent": 0}` and inside each iteration, we update one of the two keys. Like this.

Theo shows Dave his code as in [14.12](#).

**Listing 14.12 Calculating the book lending ratio using `reduce`**

```
function lendingRatio(books) {
  var bookItems = flatMap(books, "bookItems");
  var stats = _.reduce(bookItems, function(res, item) {
    if(_.get(item, "isLent")) {
      res.lent = res.lent + 1;
    } else {
      res.notLent = res.notLent + 1;
    }
    return res;
  }, {notLent: 0, lent:0});
  return stats.lent/(stats.lent + stats.notLent);
}
```

**DAVE:** Instead of updating the variables `lent` and `notLent`, now we are updating `lent` and `notLent` map fields. What's the difference?

**THEO:** Dealing with map fields instead of variables allows us to get rid of `reduce` in our business logic code.

**DAVE:** How could you iterate over a collection without `forEach` and without `reduce`?

**THEO:** I can't avoid the iteration over a collection. But I can hide `reduce` behind a utility function. Take a look at the way `reduce` is used inside the code of `lendingRatio` and tell me:

what is the meaning of the `reduce` call? (see [14.12](#))

Dave looks at the code and thinks for a long moment.

**DAVE:** I think it's counting the number of times `isLent` is `true` and `false`.

**THEO:** Right. Now, let's leverage Joe's advice about building our own data manipulation tool.

**DAVE:** How exactly?

**THEO:** I suggest that you write a `countByBoolField` utility function that counts the number of times a field is `true` and `false`.

**DAVE:** OK. But before implementing this function, let me first rewrite the code of `lendingRatio`, assuming this function already exists.

**THEO:** You are definitely a fast learner, Dave!

**DAVE:** Thanks! I think that using `countByBoolField`, the code for calculating the lending ratio would be something like this.

Dave writes the code in [14.13](#).

### Listing 14.13 Calculating the book lending ratio using a custom utility function

```
function lendingRatio(books) {
  var bookItems = flatMap(books, "bookItems");
  var stats = countByBoolField(bookItems, "isLent", "lent", "notLent");
  return stats.lent/(stats.lent + stats.notLent);
}
```

**TIP**

**Don't use `reduce`, or any other low level data manipulation function inside code that deals with business logic. Instead, write a utility function—with a proper name—that hides `reduce`.**

**THEO:** Perfect. Don't you think that this code is clearer than the code using `reduce`?

**DAVE:** I do: The code is both more concise and the intent is clearer. Let me see if I can implement `countByBoolField` now.

**THEO:** I suggest that you write a unit test first.

**DAVE:** Good idea.

Dave writes types for a bit and shows Theo the result as in [14.14](#).

**Listing 14.14 A unit test for countByBoolField**

```
var input = [
  {"a": true},
  {"a": false},
  {"a": true},
  {"a": true}
];

var expectedRes = {
  "aTrue": 3,
  "aFalse": 1
};

_.isEqual(countByBoolField(input, "a", "aTrue", "aFalse"), expectedRes);
```

**THEO:** Looks good to me. For the implementation of `countByBoolField`, I think you are going to need our `update` function.

**DAVE:** I think you're right. On each iteration, I need to increment the value of either `aTrue` or `aFalse`, using `update` and a function that increments a number by 1.

After a few minutes of trial and error, Dave comes up with the piece of code in [14.15](#), using `reduce`, `update` and `inc`.

**Listing 14.15 The implementation of countByBoolField**

```
function inc (n) {
  return n + 1;
}

function countByBoolField(coll, field, keyTrue, keyFalse) {
  return _.reduce(coll, function(res, item) {
    if (_.get(item, field)) {
      return update(res, keyTrue, inc);
    }
    return update(res, keyFalse, inc);
  }, {[keyTrue]: 0, ①
        [keyFalse]: 0});
}
```

- ① Create a map with `keyTrue` and `keyFalse` associated to 0

**THEO:** Well done! Shall we move on and review the third admin feature?

**DAVE:** I'm not done yet. The third feature is more complicated. I would like to leverage the teachings from the first two features for the implementation of the third feature.

**THEO:** OK. Call me when you're ready for the code review.

## 14.5 Unwinding at ease

Dave really struggled with the implementation of the last admin feature. After a couple of hours of frustration, Dave calls Theo for a rescue.

**DAVE:** I really had a hard time implementing the grouping by lib feature.

**THEO:** I only have a couple of minutes before my next meeting but I could try to help you. What's the exact definition of grouping by library?

**DAVE:** Let me show you the unit test I wrote.

Dave shows Theo the code on his laptop, as in [14.16](#).

### Listing 14.16 Unit test for grouping books by library

```

var books = [
    {
        "isbn": "978-1779501127",
        "title": "Watchmen",
        "bookItems": [
            {
                "id": "book-item-1",
                "libId": "nyc-central-lib",
                "isLent": true
            }
        ],
        {
            "isbn": "978-1982137274",
            "title": "7 Habits of Highly Effective People",
            "bookItems": [
                {
                    "id": "book-item-123",
                    "libId": "hudson-park-lib",
                    "isLent": true
                },
                {
                    "id": "book-item-17",
                    "libId": "nyc-central-lib",
                    "isLent": false
                }
            ]
        }
    ];
}

var expectedRes =
{
    "hudson-park-lib": [
        {
            "bookItems": [
                {
                    "id": "book-item-123",
                    "isLent": true,
                    "libId": "hudson-park-lib",
                },
                {
                    "isbn": "978-1982137274",
                    "title": "7 Habits of Highly Effective People",
                }
            ],
            "isbn": "978-1982137274",
            "title": "7 Habits of Highly Effective People",
        },
        {
            "bookItems": [
                {
                    "id": "book-item-1",
                    "isLent": true,
                    "libId": "nyc-central-lib",
                },
                {
                    "isbn": "978-1779501127",
                    "title": "Watchmen",
                }
            ],
            "bookItems": [
                {
                    "id": "book-item-17",
                    "isLent": false,
                    "libId": "nyc-central-lib",
                },
                {
                    "isbn": "978-1982137274",
                    "title": "7 Habits of Highly Effective People",
                }
            ],
            "isbn": "978-1982137274",
            "title": "7 Habits of Highly Effective People",
        }
    ],
    ".isEqual(booksByRack(books) , expectedRes);
}

```

**THEO:** Cool. Writing unit tests before implementing complicated functions was also very helpful to me when I re-factored Klaflim from OOP to DOP.

**DAVE:** Writing unit tests for functions that receive and return data is much more fun than writing unit tests for the methods of stateful objects.

**TIP**

**Before implementing a complicated function, write a unit test for it.**

**THEO:** What was difficult about the implementation of `booksByLib`?

**DAVE:** I started with a complicated implementation involving `merge` and `reduce` before I remembered that you advised me to hide `reduce` behind a generic function. But I couldn't figure out what kind of generic function I needed in that case.

**THEO:** Indeed, it's not easy to implement.

**DAVE:** I'm glad to hear that. I thought I was doing something wrong.

**THEO:** The challenge here is that you need to work with book items but the book title and ISBN are not present in the book item map.

**DAVE:** Exactly!

**THEO:** It reminds me a query I had to write a year ago on MongoDB, where data was laid out in a similar way.

**DAVE:** And what did your query look like?

**THEO:** I used MongoDB's `$unwind` operator: Given a map `m` with a field `<arr, myArray>`, it returns an array where each element is a map corresponding to `m` without `arr` and with `item` associated to an element of `myArray`.

**DAVE:** That's a bit abstract for me. Could you give me an example?

Theo comes closer to the whiteboard and draws a diagram like the one in [14.2](#).

**THEO:** In my case, I was dealing with an online store where a customer cart was represented as a map with a `customer-id` field and an `items` array field. Each element in the array represented an item in the cart. I wrote a query with `unwind` that retrieved the cart items with the `customer-id` field.

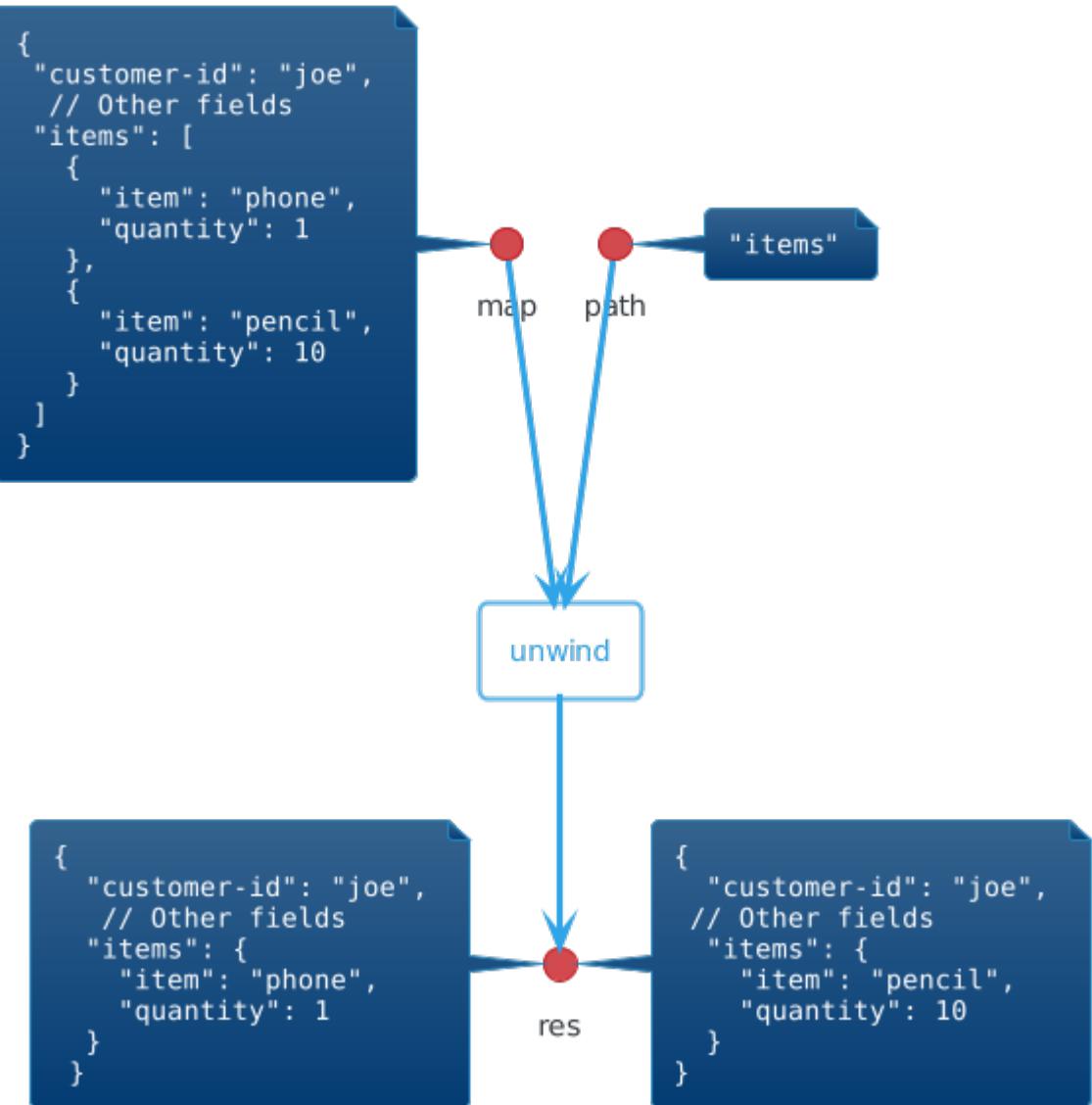


Figure 14.2 The behavior of `unwind`

**DAVE:** Amazing! That's exactly what we need. Let's write our own `unwind` function!

**THEO:** I'd be happy to pair program with you on this cool stuff. But I'm already running late for another meeting.

**DAVE:** I'm glad I'm not a manager!

When Theo leaves for your meeting, Dave prepares himself a long espresso and enjoys it while working on the implementation of `unwind`.

As Joe advised, Dave starts by writing the code for `booksByLib` as if `unwind` were already implemented. He needs to go over each book and unwind its book items using `flatMap` and `unwind`, then group the book items by their lib id using `_.groupBy`. The resulting code is in

### 14.17.

#### **Listing 14.17 Grouping books by library, using `unwind`**

```
function booksByRack(books) {
  var bookItems = flatMap(books, function(book) {
    return unwind(book, "bookItems");
  });
  return _.groupBy(bookItems, "bookItems.libId")
}
```

Dave could not believe that such a complicated function could be implemented so clearly and compactly. Dave says to himself that the complexity must reside in the implementation of `unwind`. But he's wrong!

He starts by writing a unit test for `unwind` similar to Theo's MongoDB customer cart scenario, as in [14.18.](#)

#### **Listing 14.18 A unit test for `unwind`**

```
var customer = {
  "customer-id": "joe",
  "items": [
    {
      "item": "phone",
      "quantity": 1
    },
    {
      "item": "pencil",
      "quantity": 10
    }
  ]
};

var expectedRes = [
  {
    "customer-id": "joe",
    "items": {
      "item": "phone",
      "quantity": 1
    }
  },
  {
    "customer-id": "joe",
    "items": {
      "item": "pencil",
      "quantity": 10
    }
  }
]

_.isEqual(unwind(customer, "items"), expectedRes)
```

The implementation of `unwind` is definitely not as complicated as Dave thought. It retrieves the array `arr` associated with `f` in `m` and creates for each element of `arr` a version of `m` where `f` is associated with `elem`. Dave is happy to remember that data being immutable, there is no need to clone `m`. Dave's code is in [14.19.](#)

### Listing 14.19 The implementation of `unwind`

```
function unwind(map, field) {
  var arr = _.get(map, field);
  return _.map(arr, function(elem) {
    return _.set(map, field, elem);
  });
}
```

After a few moments of contemplating his beautiful code, Dave sends Theo a message with link to the Pull Request that implements grouping books by library with `unwind`. After that he goes back home, by bike, tired but satisfied.

## 14.6 Summary

- Maintain a clear separation between the code that deals with business logic and the implementation of the data manipulation.
- Separating business logic from data manipulation makes our code, not only concise, but also easy to read as it conveys the intent in a clear manner.
- We design and implement custom data manipulation functions in a 4-step process:
- Step 1: Discover the function signature by using it before it is implemented
- Step 2: Write a unit test for the function
- Step 3: Formulate the behavior of the function in plain English
- Step 4: Implement the function
- The best way to find the signature of a custom data manipulation function is to think about the most convenient way to use it.
- Before implementing a custom data manipulation function, formulate in plain English exactly what the function does.
- Pick the least generic utility function that solves your problem.
- Don't use `reduce`, or any other low level data manipulation function inside code that deals with business logic. Instead, write a utility function—with a proper name—that hides `reduce`.
- Before implementing a complicated function, write a unit test for it.

**Table 14.1 Lodash functions introduced in this chapter**

Function	Description
<code>flatten(arr)</code>	<code>Flattens arr a single level deep</code>
<code>sum(arr)</code>	<code>Computes the sum of the values in arr</code>
<code>uniq(arr)</code>	<code>Creates an array of unique values from arr</code>
<code>every(coll, pred)</code>	<code>Checks if pred returns true for all elements of coll</code>
<code>forEach(coll, f)</code>	<code>Iterates over elements of coll and invokes f for each element</code>
<code>sortBy(coll, f)</code>	<code>Creates an array of elements, sorted in ascending order by the results of running each element in coll through f</code>

# 15 Debugging

## This chapter covers

- Reproducing a bug in code that involves primitive data types
- Reproducing a bug in code that involves aggregated data
- Replaying a scenario in the REPL
- Creating unit tests from bugs

### 15.1 Innovation at the museum

When our programs don't behave as expected, we need to investigate the source code. The traditional tool for code investigation is the debugger. The debugger allows us to run the code step by step until we find the line that causes the bug. However, a debugger doesn't allow us to reproduce the scenario that causes the problem.

In Data-Oriented programming, we can capture the context of a scenario that causes a bug and replay it in a separate process, like a REPL or a unit test. It allows us to benefit from a short feedback loop between our attempt to fix the code and the results of our attempt.

### 15.2 Determinism in programming

After a few months, Theo calls Dave to tell him that he's leaving Albatross. After Dave recovers from this first surprise, he's given another, more pleasant one. Theo informs him that after consulting with the management team, they have decided that Dave will be in charge of DOP at Albatross. In addition to the farewell at the office next week, Theo invites Dave for a last one-on-one work session, at the Exploratorium Museum of Science.

During their visit, Dave particularly enjoys the "Cells to self" exhibit in the Living systems

gallery and Theo is having fun with the "Colored shadows" exhibit in the Reflections gallery. After the visit, Theo and Dave settle in the auditorium of the Museum and open their laptops.

**DAVE:** Why did you want our last meeting to happen at the Museum of Science?

**THEO:** Remember when Joe told us that some day we'd be able to innovate in DOP?

**DAVE:** Yes.

**THEO:** Well, that day may have come. I think I have discovered an interesting connection between DOP and science. And it has implications in the way we can debug a program.

**DAVE:** I'm curious.

**THEO:** Do you believe in determinism?

**DAVE:** You mean that everything that happens in the universe is predestined and that free will is an illusion?

**THEO:** No, I don't want to get into a philosophy. This is more of a scientific question: Do you think that the same causes always produce the same effects?

**DAVE:** I think so. Otherwise, each time I use an elevator I'd be scared that the laws of physics had changed and the elevator might crash.

**THEO:** What about determinism in programming?

**DAVE:** How would you define causes and effects in programming?

**THEO:** Let's say for the sake of simplicity that in the context of programming, causes are function arguments and effects are return values.

**DAVE:** What about side effects?

**THEO:** Let's leave them aside for now.

**DAVE:** What about the program state? I mean, a function could return a different value for the same arguments if the program state has changed!

**THEO:** That's why we should avoid state as much as possible.

**DAVE:** But you can't avoid state in real-life applications!

**THEO:** Right. But we can minimize the number of modules that deal with state. In fact, that's exactly what DOP has encouraged us to do: only the `SystemState` module deals with state,

while all other modules deal with immutable data.

**DAVE:** Then I think that in modules that deal with immutable data, determinism—as you defined it—holds: for the same arguments, a function will always return the same value.

**TIP**

**In modules that deal with immutable data, function behavior is deterministic: the same arguments always lead to the same return values.**

**THEO:** Perfect. Let's give a name to the values of the function arguments that a function is called with: the function run-time context or in short the function context.

**DAVE:** I think I see what you mean. In general, the function context should involve both the function arguments and the program state. But in DOP, since we deal with immutable data, a function context is made only of the values of the function arguments.

**TIP**

**In DOP, the function context is made of the values of the function arguments.**

**THEO:** Exactly. Now, let's talk about reproducibility. Let's say that you want to capture a function context and reproduce it in another environment.

**DAVE:** Could you bit a bit more concrete about reproducing a function context in another environment?

**THEO:** Take for example a web service endpoint. You trigger the endpoint with some parameters. Inside the program—down the stack—a function `foo` is called. Now, you want to capture the context in which `foo` is called in order to reproduce later the exact same behavior of `foo`.

**DAVE:** We deal with immutable data. Therefore, if we call `foo` again with the same arguments, it will behave the same.

**THEO:** The problem is: How do you know the values of the function arguments? Remember that you didn't trigger `foo` directly. You triggered the endpoint.

**DAVE:** That's not a problem. You use a debugger and set a breakpoint inside the code of `foo` and you inspect the arguments when the program stops at the breakpoint.

**THEO:** Let's say `foo` receives three arguments: a number, a string, and a huge nested map. How do you capture the arguments and replay `foo` with the exact same arguments?

**DAVE:** I am not sure what you mean exactly by *replaying* `foo`?

**THEO:** I mean executing `foo` in the REPL.

**NOTE** The REPL (Read Eval Print Loop), sometimes called language shell, is a programming environment that takes pieces of code, executes them and displays the result. See [15.1](#) for a list of REPLs per programming language.

**Table 15.1 REPLs per programming language**

JavaScript (Browser)	Browser console
Node.js	Node CLI
Java	jshell
C#	C# REPL
Python	Python Interpreter
Ruby	Interactive Ruby

**DAVE:** Does the REPL have to be part of the process that I'm debugging?

**THEO:** It doesn't have to be. Think of the REPL as a scientific lab where developers perform experiments. Let's say you're using a separate process for the REPL.

**DAVE:** OK. For the number and the string, I can simply copy their value to the clipboard, paste them to the REPL and execute `foo` in the REPL with the same arguments.

**THEO:** That's the easy part. What about the nested map?

**DAVE:** I don't know. I don't think I can copy a nested map from a debugger to the clipboard!

**THEO:** In fact, JavaScript debuggers can. For instance, in Chrome, there is a Copy object option that appears when you right-click on data that is printed in the browser console.

**DAVE:** I never noticed it.

**THEO:** Even without that, you could serialize the nested map to a JSON string, copy the string to the clipboard and then paste the JSON string to the REPL. Finally, you deserialize the string into a hash map and call `foo` with it.

**DAVE:** Nice trick!

**THEO:** I don't think of it as a trick, but rather as a fundamental aspect of Data-Oriented programming: data is represented with generic data structures.

**DAVE:** I see. It's easy to serialize a generic data structure.

**TIP** In order to copy and paste a generic data structure, we serialize and deserialize it.

**THEO:** You just discovered the two conditions for reproducibility in programming.

**DAVE:** The first one is that data should be immutable.

**THEO:** Right. And the second one?

**DAVE:** It should be easy to serialize and deserialize any data.

**TIP**

The two conditions for reproducibility in programming are: immutability and ease of (de)serialization.

## 15.3 Reproducibility with numbers and strings

**THEO:** In fact, we don't even need a debugger in order to capture a function context.

**DAVE:** But the function context is basically made of its arguments. How can you inspect the arguments of a function without a debugger?

**THEO:** By modifying the code of the function under investigation and printing the serialization of the arguments to the console.

**DAVE:** I don't get that.

**THEO:** Let me show you what I mean with a function that deals with numbers.

**DAVE:** OK.

**THEO:** Take for instance a function that returns the nth digit of a number.

**DAVE:** Oh no. I hate digit arithmetic.

**THEO:** Don't worry, we'll find some code for it on the web.

Theo googles "nth digit of a number in JavaScript" and takes a piece of code from StackOverflow that seems to work. It is in [15.1](#).

### Listing 15.1 Calculate the nth digit of a number

```
function nthDigit(a, n) {
    return Math.floor((a / (Math.pow(10, n - 1)))) % 10;
}
```

**DAVE:** Do you understand how it works?

**THEO:** Let's see. Dividing a by  $10^{n-1}$  is like right-shifting it  $n-1$  places. Then we just need to get the rightmost digit.

**DAVE:** And the last digit of a number is obtained by the `modulo 10` operation.

**THEO:** Right. Now, imagine that this function is called down the stack when some endpoint is triggered. I'm going to modify it by adding context-capturing code.

**DAVE:** What's that?

**THEO:** Context-capturing code is code that we insert at the beginning of a function body to print the values of the arguments. Like this.

Theo edits the `nthDigit` code as in [15.2](#).

### Listing 15.2 Capturing a context made of numbers

```
function nthDigit(a, n) {
  console.log(a);
  console.log(n);
  return Math.floor((a / (Math.pow(10, n - 1)))) % 10;
}
```

**DAVE:** It looks trivial.

**THEO:** It is trivial for now but it will get less trivial in a moment. Now, please tell me what will happen when I trigger the endpoint?

**DAVE:** When the endpoint is triggered, the program will display the two numbers `a` and `n` in the console.

**THEO:** Exactly. And what would you have to do in order to replay the function in the same context as when the endpoint was triggered?

**DAVE:** I would need to copy the values of `a` and `n` from the console, paste them into the REPL, and call `nthDigit` with those two values.

**THEO:** What makes you confident that when we run `nthDigit` in the REPL, it will reproduce exactly what happened when the endpoint was triggered? Remember that the REPL might run in a separate process.

**DAVE:** Because I know that `nthDigit` depends only on its arguments.

**THEO:** Right. And how can you be sure that the arguments you pass are the same as the arguments that were passed?

**DAVE:** A number is a number!

**THEO:** I agree with you. Now, let's move on and see what happens with strings.

**DAVE:** I expect it to be *exactly* the same.

**THEO:** It's going to be *almost* the same. Let's write a function that receives a sentence and a prefix and returns `true` when the sentence contains a word that starts with the prefix.

**DAVE:** Why would anyone ever need such a weird function?

**THEO:** It could be useful for your library management system when a user wants to find books whose title contains a prefix.

**DAVE:** Interesting. I'll talk about that with Nancy. Anyway, coding such a function seems quite obvious. I need to split the sentence string into an array of words and then check whether a word in the array starts with the prefix.

**THEO:** How are you going to check whether any element of the array satisfies the condition?

**DAVE:** I think I'll use Lodash `filter` and check the length of the returned array.

**THEO:** That would work but it might have a performance issue.

**DAVE:** Why?

**THEO:** Think about it for a minute.

**DAVE:** I got it. `filter` processes all the elements in the array rather than stopping after the first match. Is there a function in Lodash that stops after the first match?

**THEO:** Yes. It's called `find`.

**DAVE:** Cool. I'll use that. Hang on.

Dave codes briefly and shows Theo his implementation of `hasWordStartingWith` using `_.find` as in [15.3](#).

### Listing 15.3 Checking whether a sentence contains a word that starts with a prefix

```
function hasWordStartingWith(sentence, prefix) {
  var words = sentence.split(" ");
  return _.find(words, function(word) {
    return word.startsWith(prefix);
  }) != null;
}
```

**THEO:** OK. Now, please add the context capturing code at the beginning of the function.

**DAVE:** Sure.

Dave does some edits and produces the code in [15.4](#).

### Listing 15.4 Capturing a context made of strings

```
function hasWordStartingWith(sentence, prefix) {
  console.log(sentence);
  console.log(prefix);
  var words = sentence.split(" ");
  return _.find(words, function(word) {
    return word.startsWith(prefix);
  }) != null;
}
```

**THEO:** Let me check your code and see what happens when I check whether the sentence *I like the word "reproducibility"* contains a word that starts with `li`.

Theo comes closer to Dave's laptop and executes the code in [15.5](#). It returns `true` as expected but doesn't display to the console the text that Dave expected. He shares his surprise with Theo.

### Listing 15.5 Testing `hasWordStartingWith`

```
hasWordStartingWith("I like the word \"reproducibility\"", "li");
// It returns true
// It displays the following two lines:
// I like the word "reproducibility"
// li
```

**DAVE:** Where are the quotes around the strings? And where are the backslashes near the quote marks surrounding "reproducibility"?

**THEO:** They disappeared!

**DAVE:** Why?

**THEO:** When you print a string to the console, the content of the string is displayed without quotes. It's more human readable.

**DAVE:** Bummer! It's not good for reproducibility. After I copy and paste a string I have to manually wrap it with quotes and backslashes.

**THEO:** Fortunately, there is a simpler solution. If you serialize your string to JSON then it has the quotes and the backslashes. For instance, this code displays to the string you expected.

Theo shows dave the code in [15.6](#)

### Listing 15.6 Displaying to the console the serialization of a string

```
console.log(JSON.stringify("I like the word \"reproducibility\""));
// It displays:
// "I like the word \"reproducibility\""
```

**DAVE:** I didn't know that strings were considered valid JSON data. I thought only objects and

arrays were valid.

**THEO:** Both compound data types and primitive data types are valid JSON data.

**DAVE:** Cool! I'll fix the code in `hasWordStartingWith` that captures the string arguments. Here you go.

Dave shares his revisions, as in [15.7](#).

### Listing 15.7 Capturing a context made of strings using JSON serialization

```
function hasWordStartingWith(sentence, prefix) {
  console.log(JSON.stringify(sentence));
  console.log(JSON.stringify(prefix));
  var words = sentence.split(" ");
  return _.find(words, function(word) {
    return word.startsWith(prefix);
  }) != null;
}
```

**THEO:** Great! Capturing Strings takes a bit more work than with numbers, but with JSON they're not too bad.

**DAVE:** Right. Now, I'm curious to see if using JSON serialization for context capturing works well with numbers.

**THEO:** It works. In fact, it works well with any data whether it's a primitive data type or a collection.

**DAVE:** Nice!

**THEO:** Now, I'm going to show you how to leverage this approach in order to reproduce a real scenario that happens in the context of your library management system.

**DAVE:** No more digit arithmetic?

**THEO:** No more!

## 15.4 Reproducibility with any data

The essence of DOP is that it treats data as a first-class citizen. As a consequence, we can reproduce any scenario that deals with data with the same simplicity as we reproduce a scenario that deals with numbers and strings.

**DAVE:** I just called Nancy to tell her about the improved version of the book search, where a prefix could match any word in the book title.

**THEO:** And?

**DAVE:** She likes the idea.

**THEO:** Great! Let's use this feature as an opportunity to exercise reproducibility with any data.

**DAVE:** Where should we start?

**THEO:** First, we need to add context-capturing code inside the function that does the book matching.

**DAVE:** The function is `Catalog.searchBooksByTitle`.

**THEO:** What are the arguments of `Catalog.searchBooksByTitle`?

**DAVE:** It has two arguments: `catalogData`, a big nested hash map and `query`, a string.

**THEO:** Could you please edit the code and add the context capturing piece?

**DAVE:** Sure.

Dave writes the content capturing code as in [15.8](#) and shows it to Theo.

#### **Listing 15.8 Capturing the arguments of `Catalog.searchBooksByTitle`**

```
Catalog.searchBooksByTitle = function(catalogData, query) {
  console.log(JSON.stringify(catalogData));
  console.log(JSON.stringify(query));
  var allBooks = _.get(catalogData, "booksByIsbn");
  var queryLowerCased = query.toLowerCase();
  var matchingBooks = _.filter(allBooks, function(book) {
    return _.get(book, "title")
      .toLowerCase()
      .startsWith(queryLowerCased);
  });
  var bookInfos = _.map(matchingBooks, function(book) {
    return Catalog.bookInfo(catalogData, book);
  });
  return bookInfos;
};
```

**THEO:** Perfect. Now let's trigger the search endpoint.

Theo triggers the search endpoint with the query "Watch" hoping to get details about Watchmen. When the endpoint returns, Theo opens the console and Dave can see the two lines of output in [15.9](#).

### Listing 15.9 Console output when triggering the search endpoint

```
{
  "booksByIsbn": {
    "978-1982137274": {
      "isbn": "978-1982137274",
      "title": "7 Habits of Highly Effective People",
      "authorIds": [
        "sean-covey", "stephen-covey"
      ],
      "978-1779501127": {
        "isbn": "978-1779501127",
        "title": "Watchmen",
        "publicationYear": 1987,
        "authorIds": [
          "alan-moore", "dave-gibbons"
        ]
      }
    },
    "authorsById": {
      "stephen-covey": {
        "name": "Stephen Covey"
      },
      "sean-covey": {
        "name": "Sean Covey"
      },
      "alan-moore": {
        "name": "Alan Moore"
      },
      "dave-gibbons": {
        "name": "Dave Gibbons"
      }
    },
    "bookIsbns": [
      "978-1982137274", "978-1779501127"
    ],
    "bookIsbns": [
      "978-1982137274", "978-1779501127"
    ],
    "bookIsbns": [
      "978-1779501127"
    ],
    "bookIsbns": [
      "978-1779501127"
    ]
  }
}
"Watch"
```

**DAVE:** I know that the first line contains the catalog data. But it's really hard to read!

**THEO:** It doesn't matter too much. You only need to copy and paste it in order to reproduce the `Catalog.searchBooksByTitle` call.

**DAVE:** Let me do that.

Dave writes code that reproduces the `Catalog.searchBooksByTitle` as in [15.10](#).

### Listing 15.10 Reproducing a function call

```
var catalogData = {
  "booksByIsbn": {
    "978-1982137274": {
      "isbn": "978-1982137274",
      "title": "7 Habits of Highly Effective People",
      "authorIds": [
        "sean-covey", "stephen-covey"
      ],
      "978-1779501127": {
        "isbn": "978-1779501127",
        "title": "Watchmen",
        "publicationYear": 1987,
        "authorIds": [
          "alan-moore", "dave-gibbons"
        ]
      }
    },
    "authorsById": {
      "stephen-covey": {
        "name": "Stephen Covey"
      },
      "sean-covey": {
        "name": "Sean Covey"
      },
      "alan-moore": {
        "name": "Alan Moore"
      },
      "dave-gibbons": {
        "name": "Dave Gibbons"
      }
    },
    "bookIsbns": [
      "978-1982137274", "978-1779501127"
    ],
    "bookIsbns": [
      "978-1982137274", "978-1779501127"
    ],
    "bookIsbns": [
      "978-1779501127"
    ],
    "bookIsbns": [
      "978-1779501127"
    ]
  }
}
"Watch";  

Catalog.searchBooksByTitle(catalogData, query);
```

**THEO:** Now that we have real catalog data in hand, we can do some interesting things in the REPL.

**DAVE:** Like what?

**THEO:** Like implementing the improved search feature without having to leave the REPL.

**TIP**

**Reproducibility allows us to reproduce a scenario in a pristine environment.**

**DAVE:** Without triggering the search endpoint?

**THEO:** Exactly! We are going to improve our code until it works as desired, leveraging the short feedback loop that the console provides.

**DAVE:** Cool! In the catalog we have the book "7 Habits of Highly Effective People". Let's see what happens when we search books that match `Habit`.

Theo replaces the value of the query in [15.10](#) with "Habit" and the code returns an empty array as in [15.11](#). It is expected because the current implementation only searches for books whose title starts with the query, while the title starts with 7 Habits.

### Listing 15.11 Testing `searchBooksByTitle`

```
Catalog.searchBooksByTitle(catalogData, 'Habit');
// []
```

**THEO:** Would you like to implement the improved search?

**DAVE:** It's not so hard as we have already implemented `hasWordStartingWith`. Here's the improved search.

Dave implements it as in [15.12](#).

### Listing 15.12 An improved version of book search

```
Catalog.searchBooksByTitle = function(catalogData, query) {
  console.log(JSON.stringify(catalogData));
  console.log(JSON.stringify(query));
  var allBooks = _.get(catalogData, "booksByIsbn");
  var matchingBooks = _.filter(allBooks, function(book) {
    return hasWordStartingWith(_.get(book, "title"), query);
  });
  var bookInfos = _.map(matchingBooks, function(book) {
    return Catalog.bookInfo(catalogData, book);
  });
  return bookInfos;
};
```

**THEO:** I like it. Let's see if it works as expected.

Dave is about to trigger the search endpoint when suddenly Theo stops him and says with an authoritative tone.

**THEO:** Don't do that!

**DAVE:** Don't do what?

**THEO:** Don't trigger an endpoint to test your code.

**DAVE:** Why?

**THEO:** Because the REPL environment gives you a much quicker feedback than triggering the endpoint. The main benefit of reproducibility is to be able to reproduce the real life conditions in a more effective environment.

Once again, Dave executes the code in [15.13](#) with "Habit", but this time it returns the details about "7 Habits of Highly Effective People".

### **Listing 15.13 Testing `searchBooksByTitle` again**

```
Catalog.searchBooksByTitle(catalogData, 'Habit');
// [ { "title": "7 Habits of Highly Effective People", ...} ]
```

**DAVE:** It works!

**THEO:** Let's try more queries: `abit` and `bit` should not return any book while `habit` and `7 Habits` should return one book.

Dave tries in the REPL the four queries that Theo suggested. For `abit` and `bit`, the code works as expected while for `habit` and `7 Habits` it fails.

**DAVE:** Let me try to fix that code.

**THEO:** I suggest that you write a couple of unit tests that check the various inputs.

**DAVE:** Good idea. Is there a way to leverage reproducibility in the context of unit tests?

**THEO:** Absolutely!

## **15.5 Unit tests**

**DAVE:** How do we leverage reproducibility in a unit test?

**THEO:** As Joe told showed me so many times, in Data-Oriented Programming unit tests are really simple. They call a function with some data and they check that the data returned by the function is the same as we expect.

**DAVE:** I remember and I have written many unit tests for the library management system following this approach. But sometimes, I struggled to provide input data for the functions under test. For instance, building catalog data with all its nested fields was not a pleasure.

**THEO:** Here's where reproducibility can help. Instead of building data manually, you put the system under the conditions you'd like to test, and capture data inside the function under test. Once data is captured, you use it in your unit test.

**DAVE:** Nice! Let me write a unit test for `Catalog.searchBooksByTitle` following this approach.

Dave triggers the search endpoint once again. Then, he opens the console and copies the line with the captured catalog data to the clipboard. Finally, he pastes it inside the code of the unit test in [15.14](#).

### Listing 15.14 A unit test with captured data

```

var catalogData = {"booksByIsbn": {"978-1982137274": {"isbn": "978-1982137274",
  "title": "7 Habits of Highly Effective People", "authorIds": ["sean-covey",
  "stephen-covey"]}, "978-1779501127": {"isbn": "978-1779501127", "title":
  "Watchmen", "publicationYear": 1987, "authorIds": ["alan-moore",
  "dave-gibbons"]}}, "authorsById": {"stephen-covey": {"name":
  "Stephen Covey"}, "bookIsbns": ["978-1982137274"]}, "sean-covey":
  {"name": "Sean Covey"}, "bookIsbns": ["978-1982137274"]}, "dave-gibbons":
  {"name": "Dave Gibbons"}, "bookIsbns": ["978-1779501127"]}, "alan-moore":
  {"name": "Alan Moore"}, "bookIsbns": ["978-1779501127"]}}};
var query = "Habit";

var result = Catalog.searchBooksByTitle(catalogData, query);
var expectedResult = [
  {
    "authorNames": [
      "Sean Covey",
      "Stephen Covey",
    ],
    "isbn": "978-1982137274",
    "title": "7 Habits of Highly Effective People",
  }
];
_.isEqual(result, expectedResult);
// true

```

**THEO:** Well done! Now, would you like me to show you how to do the same without copying and pasting?

**DAVE:** Definitely.

**THEO:** Instead of displaying the captured data to the console, we're going to write it to a file and read data from that file inside the unit test.

**DAVE:** Where are you going to save the files that store captured data?

**THEO:** Those files are part of the unit tests. They need to be under the same file tree as the unit tests.

**DAVE:** There are many files! How do we make sure a file doesn't override an existing file?

**THEO:** By following a simple file naming convention. A name for a file that stores captured data is made of two parts: a context (e.g. the name of the function where data was captured) and a universal unique identifier (UUID).

**DAVE:** How do you generate a UUID?

**THEO:** In some languages it's part of the language, while in other languages like JavaScript you need a third-party library like `uuid` (<https://github.com/uuidjs/uuid>). I happen to have a list of libraries for UUID.

Theo finds his list, as in [15.2](#) and shows Dave.

**Table 15.2 Libraries for UUID generation**

Language	UUID library
JavaScript	<a href="https://github.com/uuidjs/uuid">https://github.com/uuidjs/uuid</a>
Java	java.util.UUID
C#	Guid.NewGuid
Python	uuid
Ruby	SecureRandom

**THEO:** The code for the `dataFilePath` function that receives a context and returns a file path is pretty simple. Like this.

Theo codes up `dataFilePath` as in [15.15](#).

### Listing 15.15 Computing the file path for storing captured data

```
var capturedDataFolder = "test-data";      ①
function dataFilePath(context) {
    var uuid = generateUUID();            ②
    return capturedDataFolder + "/" + context + "-" + ".json"; ③
}
```

- ① The root folder for captured data
- ② UUID generation is language dependent (see [15.2](#))
- ③ We use `.json` file extension because we serialize data to JSON

**DAVE:** And how do we store a piece of data in a JSON file?

**THEO:** We serialize it and write it to disk.

**DAVE:** Synchronously or asynchronously?

**THEO:** I prefer to write to the disk asynchronously (or in a separate thread in run times that support multithreading) in order to avoid slowing down the real work. My implementation of `dumpData` looks like this.

Theo brings up his code for `dumpData` as in [15.16](#) and shares it with Dave.

### Listing 15.16 Dumping data in JSON format

```
function dumpData(data, context) {
    var path = dataFilePath(context);
    var content = JSON.stringify(data);
    fs.writeFile(path, content, function () { ①
        console.log("Data for " + context + " stored in: " + path); ②
    });
}
```

- ① Write asynchronously to prevent blocking the real work. The 3rd argument is a callback function that is called once write completes
- ② Display a message once data is written to the file

**DAVE:** Let me see if I can use `dumpData` inside `Catalog.searchBooksByTitle` and capture the context into a file. I think that something like this should work.

Dave uses `dumpData` in his book search code as in [15.17](#).

### Listing 15.17 Capturing the context into a file

```
Catalog.searchBooksByTitle = function(catalogData, query) {
  dumpData([catalogData, query], 'searchBooksByTitle');
  var allBooks = _.get(catalogData, "booksByIsbn");
  var queryLowerCased = query.toLowerCase();
  var matchingBooks = _.filter(allBooks, function(book) {
    return _.get(book, "title")
      .toLowerCase()
      .startsWith(queryLowerCased);
  });
  var bookInfos = _.map(matchingBooks, function(book) {
    return Catalog.bookInfo(catalogData, book);
  });
  return bookInfos;
};
```

**THEO:** Trigger the endpoint and you'll see if it works.

Once again, Dave triggers the search endpoint and sees the message in [15.18](#) in the console. When he opens the file mentioned in the log message, he sees a single line that is hard to decipher, as in [15.19](#).

### Listing 15.18 Console output when triggering the search endpoint

```
Data for searchBooksByTitle stored in
test-data/searchBooksByTitle-68e57c85-2213-471a-8442-c4516e83d786.json
```

### Listing 15.19 The content of the JSON file that captured the context

```
[{"booksByIsbn": {"978-1982137274": {"isbn": "978-1982137274",
  "title": "7 Habits of Highly Effective People", "authorIds": ["sean-covey", "stephen-covey"]}, "978-1779501127": {"isbn": "978-1779501127", "title": "Watchmen", "publicationYear": 1987, "authorIds": ["alan-moore", "dave-gibbons"]}}, "authorsById": {"stephen-covey": {"name": "Stephen Covey", "bookIsbns": ["978-1982137274"]}, "sean-covey": {"name": "Sean Covey", "bookIsbns": ["978-1982137274"]}, "dave-gibbons": {"name": "Dave Gibbons", "bookIsbns": ["978-1779501127"]}, "alan-moore": {"name": "Alan Moore", "bookIsbns": ["978-1779501127"]}}, "Habit"]}
```

**DAVE:** It's very hard to read this JSON file!

**THEO:** We can beautify the JSON string if you want.

**DAVE:** How?

**THEO:** By passing to `JSON.stringify` the number of space characters to use for indentation. How many characters do you like to use for indentation?

**DAVE:** 2.

After adding the number of indentation characters to the code of `dumpData` (as in [15.20](#)), Dave open the JSON file mentioned in the log message (it's a different file name!) and he sees a beautiful JSON array with two elements as in [15.21](#).

#### Listing 15.20 Dumping data in JSON format with indentation

```
function dumpData(data, context) {
  var path = dataFilePath(context);
  var content = JSON.stringify(data, null, 2); ①
  fs.writeFile(path, content, function () {
    console.log("Data for " + context + " stored in: " + path);
  });
}
```

- ① 2nd argument to `JSON.stringify` is ignored. 3rd argument to `JSON.stringify` specifies the number of characters to use for indentation.

**Listing 15.21 The content of the JSON file that captured the context, with indentation**

```
[
  {
    "booksByIsbn": {
      "978-1982137274": {
        "isbn": "978-1982137274",
        "title": "7 Habits of Highly Effective People",
        "authorIds": [
          "sean-covey",
          "stephen-covey"
        ]
      },
      "978-1779501127": {
        "isbn": "978-1779501127",
        "title": "Watchmen",
        "publicationYear": 1987,
        "authorIds": [
          "alan-moore",
          "dave-gibbons"
        ]
      }
    },
    "authorsById": {
      "stephen-covey": {
        "name": "Stephen Covey",
        "bookIsbns": [
          "978-1982137274"
        ]
      },
      "sean-covey": {
        "name": "Sean Covey",
        "bookIsbns": [
          "978-1982137274"
        ]
      },
      "dave-gibbons": {
        "name": "Dave Gibbons",
        "bookIsbns": [
          "978-1779501127"
        ]
      },
      "alan-moore": {
        "name": "Alan Moore",
        "bookIsbns": [
          "978-1779501127"
        ]
      }
    },
    "Habit"
  ]
]
```

**DAVE:** While looking at the contents of the JSON file, I thought about the fact that we write data to the file in an asynchronous way. It means that data is written concurrently to the execution of the function code. Right?

**THEO:** Right. As I told you, we don't want to slow down the real work.

**DAVE:** I get that. But what happens if the code of the function modifies the data that we are writing. Will we write the original data to the file, or the modified data?

**THEO:** I'll let you think about that while I get a cup of coffee at the museum Coffee shop. Would you like one?

**DAVE:** Yes, an espresso please.

While Theo goes to the coffee shop, Dave explores the exhibit outside called "Wind Arrows", hoping that his mind will be inspired by the beauty of science. He takes a few breaths to relax and after a couple of minutes, he gets an answer to his question. When Theo comes back with the two espressos, he finds Dave sitting on a chair in the middle of the last row of the auditorium.

Dave smiles at Theo and says:

**DAVE:** In DOP, we never mutate data. Therefore, my question is not a question: The code of the function cannot modify the data while we are writing to the file.

**THEO:** Exactly. Now, let me show you how to use data from the JSON file in a unit test. First, we need a function that reads data from a JSON file and deserializes it. Here.

Theo types in `readData` as in [15.22](#).

#### **Listing 15.22** Reading data from a JSON file

```
function readData(path) {
  return JSON.parse(fs.readFileSync(path));
}
```

**DAVE:** Why are you reading synchronously and not asynchronously like when we capture data?

**THEO:** Because `readData` is meant to be used inside a unit test and we cannot run the test before the data is read from the file.

**DAVE:** Makes sense. Using `readData` inside a unit test seems straightforward. Let me use it.

Dave incorporates `readData` into his unit test code as in [15.23](#).

### Listing 15.23 A unit test that reads captured data from a file

```

var data = readData('test-data/searchBooksByTitle-68e57c85-2213-471a-8442-c4516e83d786.json');
var catalogData = data[0];
var query = data[1];

var result = Catalog.searchBooksByTitle(catalogData, query);
var expectedResult = [
{
  "authorNames": [
    "Sean Covey",
    "Stephen Covey",
  ],
  "isbn": "978-1982137274",
  "title": "7 Habits of Highly Effective People",
}
];
_.isEqual(result, expectedResult);
// false

```

**THEO:** Do you prefer the version of the unit test with the inline data or with the data read from the file?

**DAVE:** It depends. When data is small, I prefer to have the inline data as it allows me to see the data. But when data is big, like the catalog data, having the data inline makes the code hard to read.

**THEO:** Now let's fix the code of the improved search so that it works with the 2 queries that return an empty result.

**DAVE:** I completely forgot about that. Do you remember those 2 queries what were?

**THEO:** Yes. It was habit and 7 Habits.

**DAVE:** The first query doesn't work because the code leaves the strings in their original case. I can easily fix that by converting both the book title and the query to lower case.

**THEO:** And what about the second query?

**DAVE:** It's much harder to deal with, as it's made of two words. Somehow I need to check whether the title contains those two prefixes subsequently.

**THEO:** Are you familiar with the \b regular expression meta character?

**DAVE:** No.

**THEO:** \b matches a position that is called a *word boundary*. It allows to perform prefix matching.

**DAVE:** Cool. Could you give me an example?

**THEO:** For instance \bHabits and \b7 Habits match 7 Habits of Highly Effective People. While abits won't match.

**DAVE:** What about \bHabits of?

**THEO:** It also matches.

**DAVE:** Excellent. This is exactly what I need! Let me fix the code of hasWordStartingWith so that it does a case-insensitive prefix match.

Dave types for a bit and shows Theo his code as in [15.24](#).

#### Listing 15.24 A revised version of haswordstartingWith

```
function hasWordStartingWith(sentence, prefix) {
  var sentenceLowerCase = sentence.toLowerCase();
  var prefixLowerCase = prefix.toLowerCase();
  var prefixRegExp = new RegExp("\\b" + prefixLowerCase); ①
  return sentenceLowerCase.match(prefixRegExp) != null;
}
```

- ① When passing \b to the RegExp constructor, we need an extra backslash

**THEO:** Now, please write unit tests for all the cases.

**DAVE:** One test per query?

**THEO:** You could but it's more efficient to have a unit test for all the queries that should return a book, and another one for all the queries that should return no books. Give me a minute.

Theo codes for a while and produces two unit tests as in [15.25](#) and [15.26](#).

#### Listing 15.25 A unit test for several queries that should return a book

```
var data = readData('test-data/searchBooksByTitle-68e57c85-2213-471a-8442-c4516e83d786.json');
var catalogData = data[0];
var queries = ["Habit", "habit", "7 Habit", "habits of"];
var expectedResult = [
  {
    "authorNames": [
      "Sean Covey",
      "Stephen Covey",
    ],
    "isbn": "978-1982137274",
    "title": "7 Habits of Highly Effective People",
  }
];

_.every(queries, function(query) {
  var result = Catalog.searchBooksByTitle(catalogData, query);
  return _.isEqual(result, expectedResult);
});
// [true, true, true, true]
```

### Listing 15.26 A unit test for several queries that should return no book

```
var data = readData('test-data/searchBooksByTitle-68e57c85-2213-471a-8442-c4516e83d786.json');
var catalogData = data[0];
var queries = ["abit", "bit", "7 abit", "habit of"];
var expectedResult = [ ];

_.every(queries, function(query) {
  var result = Catalog.searchBooksByTitle(catalogData, query);
  return _.isEqual(result, expectedResult);
});
// [true, true, true, true]
```

**DAVE:** What is `_.every`?

**THEO:** It's a Lodash function that receives a collection and a predicate and returns `true` if the predicate returns `true` for every element of the collection.

**DAVE:** Nice!

Dave executes the unit tests and they pass.

**DAVE:** Now am I allowed to trigger the search endpoint with `7 Habit` in order to confirm that the improved search works as expected?

**THEO:** Of course. It's only during the multiple iterations of code improvements that I advise you not to trigger the system from the outside, in order to benefit from a shorter feedback loop. Once you're done with the debugging and fixing, you *must* test the system from end to end.

Dave triggers the search endpoint with `7 Habit` and it returns the details about `7 Habits of Highly Effective People`.

## 15.6 Dealing with external data sources

**DAVE:** Could we also leverage reproducibility when the code involves fetching data from an external data source like a database or an external service?

**THEO:** Why not?

**DAVE:** The function context might be exactly the same but the behavior could be different if the function fetches data from a data source that returns a different response for the same query.

**THEO:** Well, it depends on the data source. Some databases are immutable in the sense that the same query will always return the same response.

**DAVE:** I have never heard about immutable databases.

**THEO:** Sometimes, they are called functional databases or append-only databases.

**DAVE:** Never heard about them either. Did you mean read-only databases?

**THEO:** For sure read-only databases are immutable. But they are not useful for storing the state of an application.

**DAVE:** How could a database be both writable and immutable?

**THEO:** By embracing time.

**DAVE:** What does *time* have to do with immutability?

**THEO:** In an immutable database, a record has an automatically generated timestamp and instead of updating a record, we create a new version of it with a new timestamp. Moreover, a query always has a time range in addition to the query parameters.

**DAVE:** Why does it guarantee that the same query will always return the same response?

**THEO:** In an immutable database, queries don't operate on the database itself. Instead, they operate on a database snapshot, which never changes. Therefore, queries with the same parameters are guaranteed to return the same response.

**DAVE:** Are there databases like that for real?

**THEO:** Yes. For instance, Datomic (<https://www.datomic.com>) is an immutable database. It is used by some digital banks.

**DAVE:** But most databases don't provide such a guarantee!

**THEO:** Right. But, in practice, when we're debugging an issue in our local environment, data usually doesn't change.

**DAVE:** What do you mean?

**THEO:** Take for instance the database of Klafim. In theory, between the time you trigger the search endpoint and the time you replay the search code from the REPL with the same context, a book might have been borrowed and its availability state in the database has changed, leading to a difference response to the search query.

**DAVE:** Exactly.

**THEO:** But in practice, in your local environment, you are the only one that interacts with the system. Therefore, it should not happen.

**DAVE:** I see. Would you allow me an analogy with science?

**THEO:** Of course!

**DAVE:** In a sense, external data sources are like hidden variables in quantum physics: in theory, they could alter the result of an experiment for no obvious reason. But in practice, at the macro level, our physical world looks stable.

Theo searches his bag to find a parcel wrapped with gift wrap from the museum's souvenir shop, which he hands to Dave with a smile. Dave opens the gift package to find a T-shirt. On one side there is an Albert Einstein avatar and his famous quote: "God does not play dice with the universe"; on the other side, an avatar of Alan Kay and his quote: "The last thing you want to do is to mess with internal state".

Dave thanks Theo for his gift and Theo can feel a touch of emotion at the back of his throat...

## 15.7 Farewell

A week after the meeting with Dave at the museum, Theo invites Joe and Nancy for his farewell drink at Albatross' offices. This is the first time that Joe meets Nancy and Theo takes the opportunity to tell Nancy that if the Klafim project meets its deadlines, it is thanks to Joe.

Everyone is curious about the name of the company Theo is going to work for, but no one dares to ask him. Finally, it's Dave who gets up the courage to ask.

**DAVE:** May I ask you what company are you going to work for?

**THEO:** I'm going to take a break.

**DAVE:** Really?

**THEO:** Yes. I'll be traveling around the world for a couple of months.

**DAVE:** And after that, will you go back to work in programming?

**THEO:** I'm not sure.

**DAVE:** Do you have other projects in mind?

**THEO:** I'm thinking of writing a book.

**DAVE:** A book?

**THEO:** Yes. Data-Oriented programming has been a meaningful journey for me. I have learned some interesting lessons about reducing complexity in programming and I would like to share my story with the community of developers.

**DAVE:** Well. If you are as good of a story teller as you are as a teacher, I am sure your book will be a success.

**THEO:** Thank you, Dave!

Monica, Dave, Nancy, Joe and the other Albatross employees raise their glasses to Theo's health and exclaim together: Cheers!

## 15.8 Summary

- We reproduce a scenario by capturing the context in which a function is called and replaying it either in the REPL or in a unit test.
- In Data-Oriented Programming, a function context is made only of data.
- There are various locations to capture a function context: the clipboard, the console, a file.
- We are able to capture a function context because data is represented with generic data structure and therefore it is easily serializable.
- Replaying a scenario in the REPL provides a short feedback loop that makes us effective when we try to fix our code.
- When we execute a function with a captured context, the behavior of the function is guaranteed to be the same as long as it only manipulates immutable data, as specified by DOP.
- In modules that deal with immutable data, function behavior is deterministic: the same arguments always lead to the same return values.
- The function context is made of the values of the function arguments.
- The REPL (Read Eval Print Loop), sometimes called language shell, is a programming environment that takes pieces of code, executes them and displays the result.
- In order to copy and paste a generic data structure, we serialize and deserialize it.
- The two conditions for reproducibility in programming are: immutability and ease of (de)serialization.
- Reproducibility allows us to reproduce a scenario in a pristine environment.

**Table 15.3 REPLs per programming language**

JavaScript (Browser)	Browser console
Node.js	Node CLI
Java	jshell
C#	C# REPL
Python	Python Interpreter
Ruby	Interactive Ruby

**Table 15.4 Libraries for UUID generation**

Language	UUID library
JavaScript	<a href="https://github.com/uuidjs/uuid">https://github.com/uuidjs/uuid</a>
Java	java.util.UUID
C#	Guid.NewGuid
Python	uuid
Ruby	SecureRandom

**Table 15.5 Lodash functions introduced in this chapter**

Function	Description
find(coll, pred)	Iterates over elements of coll, returning the first element for which pred returns true

# *Principles of Data-Oriented Programming*



Data-Oriented programming (DOP) is a programming paradigm aimed at simplifying the design and implementation of software systems where information is at the center: systems such as front-end or back-end web applications and web services.

Instead of designing information systems around software constructs that combine code and data (e.g. objects instantiated from classes), DOP encourages the *separation* of code from data. Moreover, DOP provides guidelines about how to represent and manipulate data.

The essence of DOP is that it treats data a first-class citizen; it gives developers the ability to manipulate data inside a program with the same simplicity as they manipulate numbers or strings.

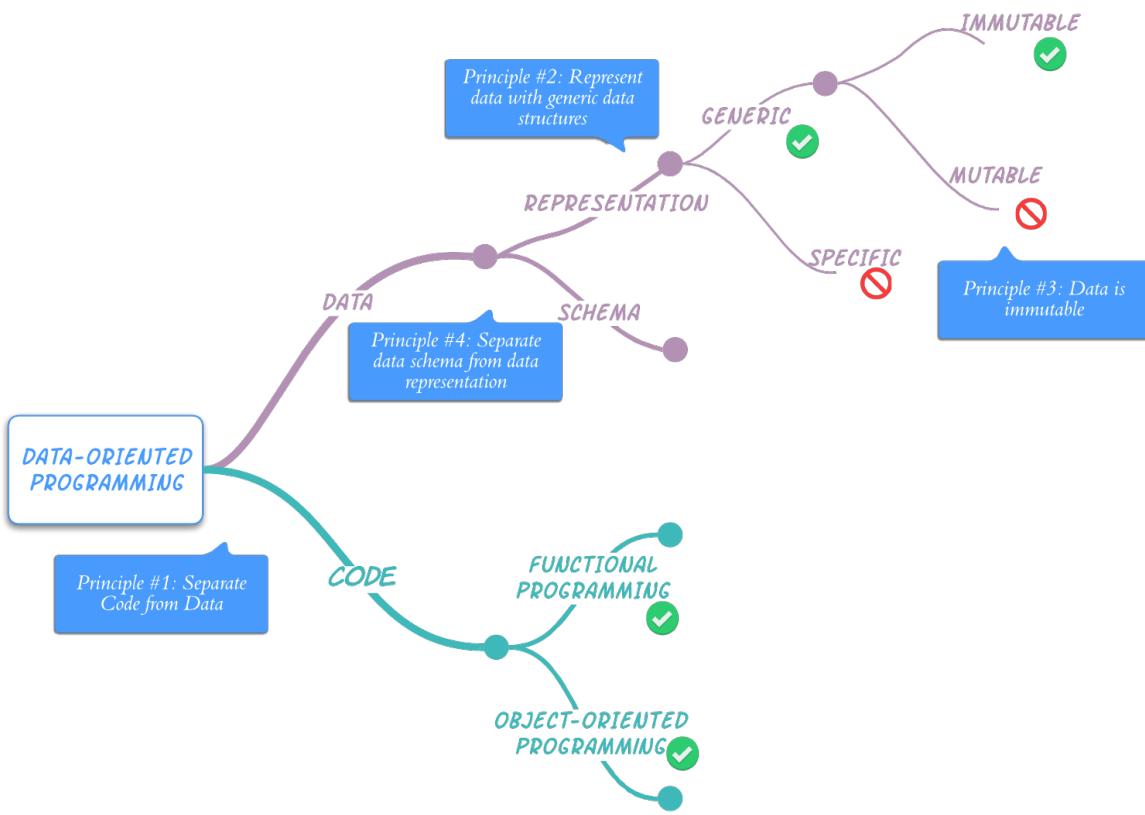
**TIP**

In Data-Oriented programming, data is treated as a first-class citizen.

Treating data as a first-class citizen is made possible by adhering to four core principles:

1. Separate code (behavior) from data
2. Represent data with generic data structures
3. Data is immutable
4. Separate data schema from data representation

When these 4 principles are combined, they form a cohesive whole as shown in [A.1](#). Systems built using DOP are simpler, easier to understand, and the developer experience is significantly improved.



**Figure A.1** The principles of Data-Oriented programming

**TIP**

In a Data-Oriented system, code is separated from data. Data is represented with generic data structures that are immutable and have a separate schema.

Notice that DOP principles are language agnostic: They can be adhered to (or broken) in:

- Object-Oriented programming (OOP) languages: Java, C#, C++...
- Functional Programming (FP) languages: Clojure, Ocaml, Haskell...
- Languages that support both OOP and FP: JavaScript, Python, Ruby, Scala...

**TIP**

DOP Principles are language agnostic.

**WARNING**

For OOP developers, the transition to DOP might require more of a mind shift than for FP developers, as DOP prohibits the encapsulation of data in stateful classes.

This appendix, illustrates in a succinct way, how those principles could be applied or broken in JavaScript.

Mentioned briefly, are the benefits of adherence to each principle, and the costs paid to enjoy those benefits.

This appendix illustrates the principles of DOP via simple code snippets. Throughout the book, the application of DOP principles to production information systems is explored in depth.

## A.1 DOP Principle #1: Separate code from data

### A.1.1 The principle in a nutshell

Principle #1 is a design principle that recommends a clear separation between code (behavior) and data.

**NOTE**

**Principle #1: Separate code from data in a way that the code resides in functions whose behavior does not depend on data that is encapsulated in the function's context.**

This may appear to be a Functional Programming principle, but in fact one can adhere to it or break it either in FP or in OOP:

- Adherence to this principle in OOP means aggregating the code as methods of a static class
- Breaking this principle in FP, means hiding state in the lexical scope of a function

Also, Principle #1 does not relate to the way data is represented: data representation is addressed by Principle #2.

### A.1.2 Illustration of Principle #1

The following illustrates how this principle may be adhered to or broken in simple program that deals with:

1. An author entity with a `firstName`, a `lastName` and the number of `books` they wrote
2. A piece of code that calculates the full name of the author
3. A piece of code that determines if an author is prolific, based on the number of books they wrote

The exploration of Principle #1 begins by illustrating how it can be broken in OOP.

### BREAKING PRINCIPLE #1 IN OOP

Principle #1 in OOP is broken when writing code that combines data and code together in an object, as in [A.1](#).

## Listing A.1 Breaking Principle #1 in OOP

```
class Author {
    constructor(firstName, lastName, books) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.books = books;
    }
    fullName() {
        return this.firstName + " " + this.lastName;
    }
    isProlific() {
        return this.books > 100;
    }
}

var obj = new Author("Isaac", "Asimov", 500); ①
obj.fullName();
// "Isaac Asimov"
```

- ① Isaac Asimov really wrote around 500 books!

## BREAKING PRINCIPLE #1 IN FP

Breaking this principle without classes in FP style means hiding data in the lexical scope of a function, like in [A.2](#).

## Listing A.2 Breaking Principle #1 in FP

```
function createAuthorObject(firstName, lastName, books) {
    return {
        fullName: function() {
            return firstName + " " + lastName;
        },
        isProlific: function () {
            return books > 100;
        }
    };
}

var obj = createAuthorObject("Isaac", "Asimov", 500);
obj.fullName();
// "Isaac Asimov"
```

## ADHERING TO PRINCIPLE #1 IN OOP

Compliance with this principle may be achieved, even with classes, by writing program such that:

1. The code consists of static methods
2. The data is encapsulated in data classes, i.e. classes that are merely containers of data

An example of this approach is shown in [A.3](#).

### Listing A.3 Following Principle #1 in OOP

```

class AuthorData {
    constructor(firstName, lastName, books) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.books = books;
    }
}

class NameCalculation {
    static fullName(data) {
        return data.firstName + " " + data.lastName;
    }
}

class AuthorRating {
    static isProlific (data) {
        return data.books > 100;
    }
}

var data = new AuthorData("Isaac", "Asimov", 500);
NameCalculation.fullName(data);
// "Isaac Asimov"

```

### ADHERING TO PRINCIPLE #1 IN FP

Compliance with this principle in FP style means separating code from data, as in [A.4](#).

### Listing A.4 Following Principle #1 in FP

```

function createAuthorData(firstName, lastName, books) {
    return {
        firstName: firstName,
        lastName: lastName,
        books: books
    };
}

function fullName(data) {
    return data.firstName + " " + data.lastName;
}

function isProlific (data) {
    return data.books > 100;
}

var data = createAuthorData("Isaac", "Asimov", 500);
fullName(data);
// "Isaac Asimov"

```

Having illustrated how to follow or break Principle #1, both in OOP and FP, the benefits that Principle #1 brings to our programs is explored.

#### A.1.3 Benefits of Principle #1

Careful separation of code from data benefits programs in the following ways:

1. Code can be reused in different contexts

2. Code can be tested in isolation
3. Systems tend to be less complex

## BENEFIT #1: CODE CAN BE REUSED IN DIFFERENT CONTEXTS

Imagine that besides the author entity, there is a user entity that has nothing to do with authors but has two of the same data fields as the author entity: `firstName` and `lastName`.

The logic of calculating the full name is the same for authors and users: retrieving the values of two fields with the exact same names. However, in traditional OOP as in the version with `createAuthorObject` in [A.5](#), the code of `fullName` cannot be reused on a user in a *straightforward* way, as it is locked inside the `Author` class.

### **Listing A.5 The code of `fullName` is locked in the `Author` class**

```
class Author {
  constructor(firstName, lastName, books) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.books = books;
  }
  fullName() {
    return this.firstName + " " + this.lastName;
  }
  isProlific() {
    return this.books > 100;
  }
}
```

One way to achieve code re-usability when code and data are mixed is to use OOP mechanisms, like inheritance or composition, to let `User` and `Author` classes use the same `fullName` method. These techniques are adequate for simple use cases, but in real world systems, the abundance of classes (either base classes or composite classes) tends to increase complexity.

Another way is shown in [A.6](#): The code of `fullName` is duplicated inside a `createUserObject` function.

## Listing A.6 Duplicating code in OOP to avoid inheritance

```

function createAuthorObject(firstName, lastName, books) {
    var data = {firstName: firstName, lastName: lastName, books: books};

    return {
        fullName: function fullName() {
            return data.firstName + " " + data.lastName;
        }
    };
}

function createUserObject(firstName, lastName, email) {
    var data = {firstName: firstName, lastName: lastName, email: email};

    return {
        fullName: function fullName() {
            return data.firstName + " " + data.lastName;
        }
    };
}

var obj = createUserObject("John", "Doe", "john@doe.com");
obj.fullName();
// "John Doe"

```

In DOP, code reuse is much simpler for two reasons:

1. The code that deals with full name calculation is separate from the code that deals with the creation of author data.
2. The function that calculates the full name works with any hash map that has a `firstName` and a `lastName` field.

As a consequence, no modification to the code that deals with author entities is necessary in order to make it available to user entities. It is possible to leverage the fact that data relevant to the full name calculation for a user and an author has the same shape. With no modifications, the `fullName` function works properly both on author data and on user data, as shown in [A.7](#).

## Listing A.7 Using the same code on data entities of different types (FP style)

```

function createAuthorData(firstName, lastName, books) {
    return {firstName: firstName, lastName: lastName, books: books};
}

function fullName(data) {
    return data.firstName + " " + data.lastName;
}

function createUserData(firstName, lastName, email) {
    return {firstName: firstName, lastName: lastName, email: email};
}

var authorData = createAuthorData("Isaac", "Asimov", 500);
fullName(authorData);

var userData = createUserData("John", "Doe", "john@doe.com");
fullName(userData);
// "John Doe"

```

When Principle 1 is applied in OOP, code reuse is straightforward even when classes are used. In statically-typed OOP languages (like Java or C), a common interface would have to be created for `AuthorData` and `UserData`, but in a dynamically typed language like JavaScript, it is not required.

The code of `NameCalculation.fullName()` works both with author data and user data, as shown in [A.8](#).

#### **Listing A.8 Using the same code on data entities of different types (OOP style)**

```
class AuthorData {
    constructor(firstName, lastName, books) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.books = books;
    }
}

class NameCalculation {
    static fullName(data) {
        return data.firstName + " " + data.lastName;
    }
}

class UserData {
    constructor(firstName, lastName, email) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
    }
}

var userData = new UserData("John", "Doe", "john@doe.com");
NameCalculation.fullName(userData);

var authorData = new AuthorData("Isaac", "Asimov", 500);
NameCalculation.fullName(authorData);
// "John Doe"
```

**TIP**

When code is separate from data it is straightforward to reuse code in different contexts. This benefit is achievable both in FP and in OOP.

## **BENEFIT #2: CODE CAN BE TESTED IN ISOLATION**

A similar benefit is the ability to test code in an isolated context.

When code is not separate from data, it is necessary to instantiate an object to test its methods. For instance, in order to test the `fullName` code that lives inside the `createAuthorObject` function, an author object must be instantiated, as shown in [A.9](#).

### Listing A.9 Testing code when code and data are mixed requires to instantiate the full object

```
var author = createAuthorObject("Isaac", "Asimov", 500);
author.fullName() === "Isaac Asimov"
// true
```

In this simple scenario, it is not overly burdensome (only loading unnecessarily the code for `isProlific`), but in a real world situation, instantiating an object might involve complex and tedious setup.

In the DOP version, where `createAuthorData` and `fullName` are separate, the data to be passed to `fullName` can be created in isolation, and `fullName` can be tested in isolation. An example is shown in [A.10](#).

### Listing A.10 Separating code from data allows us to test code in isolation (FP style)

```
var author = {
  firstName: "Isaac",
  lastName: "Asimov"
};
fullName(author) === "Isaac Asimov"
// true
```

If classes are used, it is only necessary to instantiate a data object. The code for `isProlific` that lives in a separate class than `fullName` does not have to be loaded in order to test `fullName`, as shown in [A.11](#).

### Listing A.11 Separating code from data allows us to test code in an isolated context (OOP style)

```
var data = new AuthorData("Isaac", "Asimov");

NameCalculation.fullName(data) === "Isaac Asimov"
// true
```

**TIP**

Writing tests is easier when code is separated from data

## BENEFIT #3: SYSTEMS TEND TO BE LESS COMPLEX

The third and last benefit of applying Principle #1 is that systems tend to be less complex.

This benefit is the deepest one but also the one that is most subtle to explain.

The type of complexity I refer to is the one which makes systems hard to understand as defined in the beautiful paper "Out of the Tar Pit" (<https://github.com/papers-we-love/papers-we-love/blob/master/design/out-of-the-tar-pit.pdf>). It has nothing to do with the complexity of the resources consumed by a program.

Similarly, references to simplicity mean "not complex", in other words: easy to understand.

Keep in mind that complexity and simplicity (like hard and easy) are not absolute but relative concepts. The complexity of two systems can be compared, to determine whether system A is more complex (or simpler) than system B.

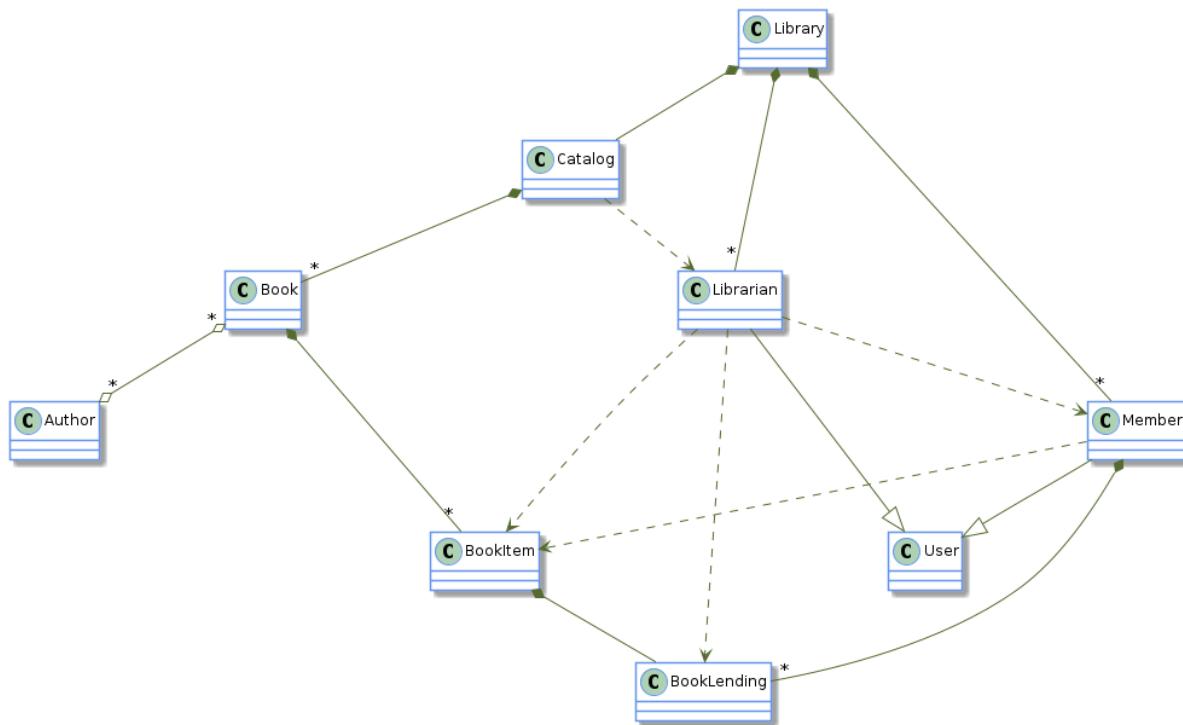
**NOTE**

Complex in the context of this book means: hard to understand

When code and data are kept separate, the system tends to be easier to understand for two reasons:

1. The scope of a data entity or a code entity is smaller than the scope of an entity that combines code and data. Therefore, each entity is easier to understand.
2. Entities of the system are split into disjoint groups: code and data. Therefore entities have fewer relations to other entities.

This insight is illustrated in a class diagram of a Library management system, as in [A.2](#), where code and data are mixed.



**Figure A.2 A class diagram overview for a Library management system**

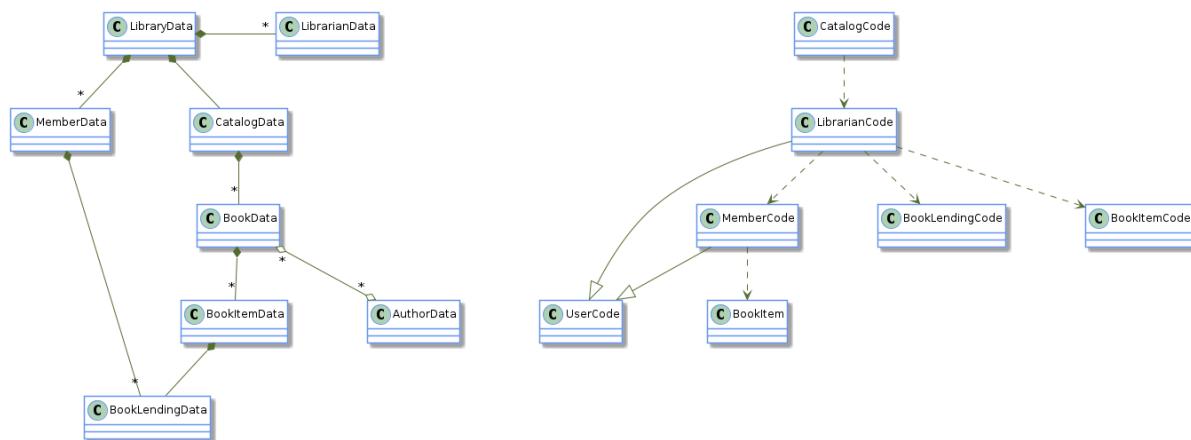
It is not necessary to know the details of the classes of this system to see that the diagram represents a complex system in the sense that it is hard to understand. The system is hard to understand because there are many dependencies between the entities that compose the system.

The most complex entity of the system is the **Librarian** entity which is connected via 6

relations to other entities. Some relations are data relations (association and composition) and some relations are code relations (inheritance and dependency). But in this design, the Librarian entity mixes code and data, therefore it has to be involved in both data and code relations.

Now, if each entity of the system is split into a code entity and a data entity *without making any further modification to the system*, the result is shown in [A.3](#), that is made of two disconnected parts:

- The left part is made only of data entities and data relations: association and composition
- The right part is made only of code entities and code relations: dependency and inheritance



**Figure A.3 A class diagram where every class is split into code and data entities**

The new system where code and data are separate is easier to understand than the original system where code and data are mixed: the data part of the system and the code part of the system can each be understood on its own.

**TIP**

A system made of disconnected parts is less complex than a system made of a single part.

One could argue that the complexity of the original system where code and data are mixed is due to a bad design and that an experienced OOP developer would have designed a simpler system, leveraging smart design patterns. That is true, but in a sense it is irrelevant. The point of Principle #1 is that a system made of entities that do not combine code and data *tends* to be simpler than a system made of entities that combine code and data.

It has been said many times that *simplicity is hard*. According to the first principle of DOP, simplicity is easier to achieve when separating code and data.

**TIP**

**Simplicity is easier to achieve when code is separated from data.**

### A.1.4 Price for Principle #1

The price paid in order to benefit from the separation between code and data is that:

1. There is no control on what code can access what data
2. No packaging
3. Our systems are made from more entities

#### **PRICE #1: THERE IS NO CONTROL ON WHAT CODE CAN ACCESS WHAT DATA**

When code and data are mixed, it is easy to understand what pieces of code can access what kinds of data.

For example, in OOP the data is encapsulated in an object - which guarantees that the data is accessible only by the object's methods.

In DOP, data stands on its own. It is transparent if you like. As a consequence, it can be accessed by any piece of code.

When refactoring the shape of our data, *every* place in our code that accesses it must be known.

Without the application of Principle #3 enforcing data immutability, data being accessible by any piece of code is inherently unsafe. It would be very hard to guarantee the validity of our data.

**TIP**

**Data safety is ensured by another principle (Principle #3) that enforces data immutability.**

#### **PRICE #2: NO PACKAGING**

One of the benefits of mixing code and data is that when you have an object in hand, it is a package that contains both the code (via methods) and the data (via members).

As a consequence, it is easy to discover the how to manipulate the data: you look at the methods of the class.

In DOP, the code that manipulates the data could be anywhere. For example, `createAuthorData` could be in one file and `fullName` in another file. It makes it difficult for developers to discover that the `fullName` function is available. In some situations, it could lead to wasted time and unnecessary code duplication.

### PRICE #3: OUR SYSTEMS ARE MADE FROM MORE ENTITIES

Let's do simple arithmetic. Imagine a system made of  $N$  classes that combine code and data. When you split the system into code entities and data entities, you get a system made of  $2N$  entities.

This calculation is not accurate, because usually when you separate code and data, the class hierarchy tends to get simpler, as less class inheritance and composition are needed. Therefore the number of classes in the resulting system will probably be somewhere between  $N$  and  $2N$ .

On one hand, when adhering to Principle #1, the entities of the system are simpler.

On the other hand, there are more entities.

This price is mitigated by Principle #2 that guides us to represent our data with generic data structures.

**TIP**

**When adhering to Principle #1, systems are made of simpler entities but there are more of them.**

#### A.1.5 Summary

DOP requires the separation of code from data.

In OOP languages, aggregate code in static methods and data in classes with no methods.

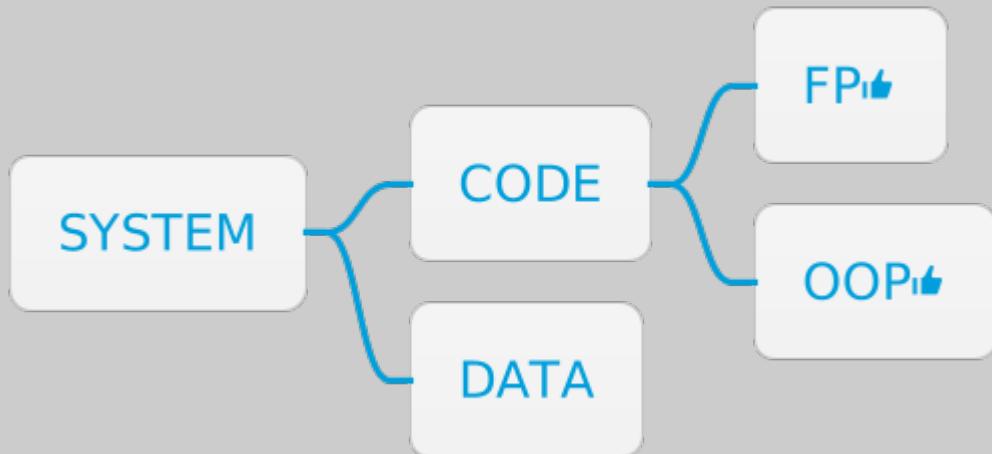
In FP languages, avoid hiding data in the lexical scope of functions.

Separating code from data comes at a price: it reduces control over what pieces of code access our data and could cause our systems to be made of more entities.

But it is worth paying the price because when adhering to this principle our code can be reused in different contexts in a straightforward way and tested in isolation. Moreover, a system made of separate entities for code and data tends to be easier to understand.

**SIDE BAR****DOP Principle #1: Separate code from data****The Principle**

Separate code from data in a way that the code resides in functions whose behavior does not depend on data that encapsulated in the function's context.

**Benefits**

1. Code can be reused in different contexts
2. Code can be tested in isolation
3. Systems tend to be less complex

**Price**

1. No control on what code access what data
2. No packaging
3. Our systems are made from more entities

## A.2 DOP Principle #2: Represent data with generic data structures

### A.2.1 The principle in a nutshell

When adhering to Principle #1, code is separated from data. DOP is not opinionated about the programming constructs to use for organizing the code but it has a lot to say about how the data should be represented. That is the theme of Principle #2.

**NOTE**

**Principle #2: Represent application data with generic data structures.**

The most common generic data structures are maps (a.k.a dictionaries) and arrays (or lists). But other generic data structures (e.g. sets, trees, queues) can be used as well.

Principle #2 does not deal with the mutability or the immutability of the data. That is the theme of Principle #3: Data is immutable.

### A.2.2 Illustration of Principle #2

In DOP, data is represented with generic data structures (like maps and arrays) instead of instantiating data via specific classes.

In fact, most of the data entities that appear in a typical application could be represented with maps and arrays (or lists). But there exist other generic data structures (e.g. sets, lists, queues...) that might be required in some use cases.

Let's look at the same simple example used to illustrate Principle #1: data that represents an author.

An author is a data entity with a `firstName`, a `lastName` and the number of books they have written.

Principle #2 is broken when using a specific class to represent an author, as in [A.12](#).

#### **Listing A.12 Breaking Principle #2 in OOP**

```
class AuthorData {
    constructor(firstName, lastName, books) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.books = books;
    }
}
```

Principle #2 is followed when using a map (a.k.a a dictionary or an associative array)—which is a generic data structure—to represent an author, as in [A.13](#).

#### **Listing A.13 Following Principle #2 in OOP**

```
function createAuthorData(firstName, lastName, books) {
    var data = new Map();
    data.firstName = firstName;
    data.lastName = lastName;
    data.books = books;
    return data;
}
```

In a language like JavaScript, a map could also be instantiated via a data literal, which is a bit more convenient. An example is shown in [A.14](#).

### **Listing A.14 Following Principle #2 with map literals**

```
function createAuthorData(firstName, lastName, books) {
  return {
    firstName: firstName,
    lastName: lastName,
    books: books
  };
}
```

### **A.2.3 Benefits of Principle #2**

Using generic data structures to represent data has the following benefits:

- The ability to leverage generic functions that are not limited to our specific use case
- A Flexible data model

### **LEVERAGE FUNCTIONS THAT ARE NOT LIMITED TO A SPECIFIC USE CASE**

There is a famous quote by Alan Perlis that summarizes this benefit very well:

*It is better to have 100 functions operate on one data structure than 10 functions operate on 10 data structures.*

– Alan Perlis

Using generic data structures to represent data makes it possible to manipulate data with the rich set of functions available on those data structures natively in our programming language in addition to the ones provided by third party libraries.

For instance, JavaScript natively provides some basic functions on maps and arrays and third-party libraries like lodash (<https://lodash.com/>) extend the functionality with even more functions.

As an example, when an author is represented as a map, it can be serialized into JSON using `JSON.stringify()` which is part of JavaScript, as shown in [A.15](#).

### **Listing A.15 Data serialization comes for free when adhering to Principle #2**

```
var data = createAuthorData("Isaac", "Asimov", 500);
JSON.stringify(data);
//  "{\"firstName\":\"Isaac\", \"lastName\":\"Asimov\", \"books\":500}"
```

Serializing author data without the number of books can be accomplished via lodash's `_.pick()` function to create an object with a subset of keys. An example is shown in [A.16](#).

## Listing A.16 Manipulating data with generic functions

```
var data = createAuthorData("Isaac", "Asimov", 500);
var dataWithoutBooks = _.pick(data, ["firstName", "lastName"]);
JSON.stringify(dataWithoutBooks);
// {"firstName": "Isaac", "lastName": "Asimov"}
```

**TIP**

**When adhering to Principle #2, a rich set of functionality is available for data manipulat.**

### FLEXIBLE DATA MODEL

When using generic data structures, the data model is flexible and data is not forced into a specific shape. Data can be created with no predefined shape, and its shape can be modified at will.

In classical OOP—*when not adhering to Principle #2*—each piece of data is instantiated via a class and must follow a rigid shape. When a slightly different data shape is needed, a new class must be defined.

Take for example a class `AuthorData` that represents an author entity made of 3 fields: `firstName`, `lastName` and `books`. Suppose that you want to add a field `fullName` with the full name of the author.

When failing to adhere to Principle #2, a new class `AuthorDataWithFullName` must be defined.

However when using generic data structures, fields may be added to (or removed from) a map *on the fly*, as in [A.17](#).

## Listing A.17 Adding a field on the fly

```
var data = createAuthorData("Isaac", "Asimov", 500);
data.fullName = "Isaac Asimov";
```

**TIP**

**Working with a flexible data model is particularly useful in applications where the shape of the data tends to be dynamic (e.g. web apps and web services).**

Part 1 of the book explored in detail the benefits of a flexible data model in real world applications.

### A.2.4 Price for Principle #2

The price paid for representing data with generic data structures is:

- Slight performance hit

- No data schema
- No compile time check that the data is valid
- In some statically-typed languages, type casting is needed

### PRICE #1: PERFORMANCE HIT

When specific classes are used to instantiate data, retrieving the value of a class member is fast because the compiler knows how the data will look and can do many optimizations.

With generic data structures, it is harder to optimize, so retrieving the value associated to a key in a map is a bit slower than retrieving the value of a class member. Similarly setting the value of an arbitrary key in a map is a bit slower than setting the value of a class member.

**TIP**

**Retrieving and storing the value associated to an arbitrary key from a map is a bit slower than with a class member.**

In most programming languages, this performance hit is not significant, but it is something to keep in mind.

### PRICE #2: NO DATA SCHEMA

When data is instantiated from a class, the information about the data shape is in the class definition. Every piece of data has an associated data shape.

The existence of data schema at a class level is useful for developers and for IDEs:

- Developers can easily discover the expected data shape
- IDEs provide features like field names auto-completion

When data is represented with generic data structures, the data schema is not part of the data representation. As a consequence, some pieces of data might have an associated data schema while other pieces of data do not (See Principle #4).

**TIP**

**When generic data structures are used to store data, the data shape is not part of the data representation.**

### PRICE #3: NO COMPILE TIME CHECK THAT THE DATA IS VALID

Look again at the `fullName` function in [A.18](#), created to explore Principle #1:

#### **Listing A.18 A function that receives the data it manipulates as an argument**

```
function fullName(data) {
    return data.firstName + " " + data.lastName;
}
```

When data is passed to `fullName` that does not conform to the shape `fullName` expects, an error occurs at runtime.

With generic data structures, mistyping the field storing the first name (`fistName` instead of `firstName`), does not result in a compile time error or an exception; rather `firstName` is mysteriously omitted from the result as in [A.19](#).

#### **Listing A.19 Unexpected behavior when data does not conform to the expected shape**

```
fullName({fistName: "Issac", lastName: "Asimov"});
//   undefined Asimov
```

When data is instantiated only via classes with rigid data shape, this type of error is caught at compile-time.

<b>TIP</b>	<b>When data is represented with generic data structures, data shape errors are caught only at runtime.</b>
------------	---

This drawback is mitigated by the the application of Principle #4 that deals with data validation.

### **PRICE #4: IN SOME STATICALLY-TYPED LANGUAGES, EXPLICIT TYPE CASTING IS NEEDED**

#### **EXPLICIT TYPE CASTING IN JAVA**

In a statically-typed language like Java, author data can be represented as a map whose keys are of type `String` and values are of types `Object`. For example, in Java author data is represented by a `Map<String, Object>` as in [A.20](#).

#### **Listing A.20 Representing author data as a string map in Java**

```
var asimov = new HashMap<String, Object>();

asimov.put("firstName", "Isaac");
asimov.put("lastName", "Asimov");
asimov.put("books", 500);
```

The information about the exact type of the field values is not available at compile-time. When accessing a field, an explicit type cast is required. For instance, in order to check whether an author is prolific, value of the `books` field must be type cast to integer, as in [A.21](#).

#### **Listing A.21 Type casting is required when accessing a field in Java**

```
class AuthorRating {
    static boolean isProlific (Map<String, Object> data) {
        return (int)data.get("books") > 100;
    }
}
```

Notice that Java JSON serialization libraries like `gson` (<https://github.com/google/gson>) support

serialization of maps of type `Map<String, Object>`, without requiring from the user to do any type casting. All the magic happens behind the scenes!

#### DYNAMIC FIELDS IN C#

C# supports a dynamic data type called `dynamic`<sup>12</sup> that allows type checking to occur at run-time. Leveraging this feature, author data is represented as a dictionary where the keys are of type `string` and the values are of type `dynamic` as in [A.22](#).

#### Listing A.22 Representing author data as a dynamic string map in C#

```
var asimov = new Dictionary<string, dynamic>();
asimov["name"] = "Isaac Asimov";
asimov["books"] = 500;
```

The information about the exact type of the field values is resolved at run-time. When accessing a field, no type cast is required. For instance, when checking whether an author is prolific, the `books` field can be accessed as though it were declared as an integer as in [A.23](#).

#### Listing A.23 Type casting is not necessary when accessing a dynamic field in C#

```
class AuthorRating {
    public static bool isProlific (Dictionary<String, dynamic> data) {
        return data["books"] > 100;
    }
}
```

### A.2.5 Summary

DOP uses generic data structures to represent data.

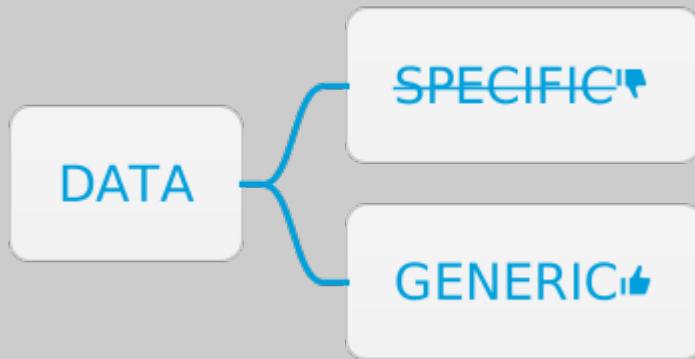
This might cause a (small) performance hit and impose the need to manually document the shape of data - as the compiler cannot validate it statically.

Adherence to this principle enables the manipulation of data with a rich set of generic functions (provided by the language and by third party libraries) and our data model is flexible.

At this point the data could be either mutable or immutable. The next principle illustrates the value of immutability.

**SIDE BAR****DOP Principle #2: Represent data with generic data structures****The Principle**

Represent application data with generic data structures, mostly maps and arrays (or lists).

**Benefits**

- Leverage generic functions that are not limited to our specific use case
- Flexible data model

**Price**

- Performance hit
- No data schema
- No compile time check that the data is valid
- In some statically-typed languages, explicit type casting is needed

## A.3 DOP Principle #3: Data is immutable

### A.3.1 The principle in a nutshell

With data separated from code and represented with generic data structures, how are *changes* to the data managed?

DOP is very strict on this question. Mutation of data is not allowed.

**NOTE****Principle #3: Data is immutable.**

In DOP, changes to data are accomplished by creating new versions of the data.

The *reference* to a variable may be changed so that it refers to a new version of the data, but *value* of the data itself must never change.

### A.3.2 Illustration of Principle #3

Think about the number 42. What happens to 42 when you add 1 to it? Does it become 43?

No! 42 stays 42 forever!!!

Now put 42 inside an object {num: 42}. What happens to the object when you add 1 to 42? Does it become 43?

It depends on the programming language:

- In Clojure, a programming language that embraces data immutability, the value of the num field stays 42 forever, no matter what.
- In many programming languages, the value of the num field becomes 43.

For instance, in JavaScript, mutating the field of a map referred by two variables has an impact on both variables, as shown in [A.24](#).

#### **Listing A.24 Mutating data referred by two variables impact both variables**

```
var myData = {num: 42};
var yourData = myData;

yourData.num = yourData.num + 1;
console.log(myData.num);
// 43
```

Now, myData.num equals 43!

According to DOP, data should never change. Instead of mutating data, a new version of it is created.

A naive (and inefficient) way to create a new version of a data is to clone it before modifying it.

For instance, in [A.25](#) there is a function that changes the value of a field inside an object, by cloning the object via `Object.assign` provided natively by JavaScript. When `changeValue` is called on `myData`, `myData` is not affected: `myData.num` remains 42. That is the essence of data immutability.

### **Listing A.25 Data immutability via cloning**

```
function changeValue(obj, k, v) {
  var res = Object.assign({}, obj);
  res[k] = v;
  return res;
}

var myData = {num: 42};
var yourData = changeValue(myData, "num", myData.num + 1);
console.log(myData.num);
// 43
```

Embracing immutability in an efficient way (both in terms of computation and memory) requires a third party library like Immutable.js (<https://immutable-js.com/>) that provides an efficient implementation of persistent data structures (a.k.a immutable data structures).

In most programming languages, there exist libraries that provide an efficient implementation of persistent data structures.

With `Immutable.js`, JavaScript native maps and arrays are not used, but rather immutable maps and immutable lists instantiated via `Immutable.Map` and `Immutable.List`. An element of a map is accessed using the `get` method. A new version of the map is created when a field is modified, with the `set` method as in [A.26](#).

### **Listing A.26 Creating and manipulating immutable data efficiently with a third-party library**

```
var myData = Immutable.Map({num: 42});
var yourData = myData.set("num", 43);
console.log(yourData.get("num"));
// 43
console.log(myData.get("num"));
// 42
```

`yourData.get("num")` is 43 but `myData.get("num")` remains 42.

**TIP**

**When data is immutable, instead of mutating data, a new version of it is created.**

### **A.3.3 Benefits of Principle #3**

When programs are constrained from mutating data, they benefit from:

- Data access to all with confidence
- Code behavior is predictable
- Equality check is fast
- Concurrency safety for free

## BENEFIT #1: DATA ACCESS TO ALL WITH CONFIDENCE

According to Principle #1: Separate code from data, data access is transparent: Any function is allowed to access any piece of data. Without data immutability, it is necessary to be very careful when passing data as an argument to a function. Either make sure the function does not mutate the data or clone the data before it is passed to the function.

When adhering to data immutability, none of this is required.

**TIP**

**When data is immutable in can be passed to any function with confidence, because data never changes.**

## BENEFIT #2: CODE BEHAVIOR IS PREDICTABLE

As an illustration of what is meant by *predictable*, here is an example of an *unpredictable* piece of code that does not adhere to data immutability.

Please take a look at the piece of asynchronous JavaScript code in [A.27](#).

### **Listing A.27 When data is mutable the behavior of asynchronous code is not predictable**

```
var myData = {num: 42};
setTimeout(function (data){
    console.log(data.num);
}, 1000, myData);
myData.num = 0;
```

The value of `data.num` inside the timeout callback is not predictable. It depends whether or not the data is modified by another piece of code, during the 1000ms of the timeout.

However, with immutable data, it is guaranteed that data never changes and that `data.num` is always 42 inside the callback!

**TIP**

**When data is immutable, the behavior of code that manipulates data is predictable**

## BENEFIT #3: EQUALITY CHECK IS FAST

With UI frameworks like `React.js`, there are frequent checks to see what portion of the "UI data" has been modified since the previous rendering cycle. Portions that did not change are not rendered again.

In fact, in a typical front-end application, most of the UI data is left unchanged between subsequent rendering cycles. In a React application that does not adhere to data immutability, it is necessary to check every (nested) part of the UI data.

However in a React application that follows data immutability, it is possible to optimize the comparison of the data for the case where data is not modified. Indeed, when the object address is the same, then it is certain that the data did not change. Comparing object addresses is much faster than comparing all the fields.

**TIP**

**Immutable data enables fast equality checks by comparing data by reference.**

In Part 1 fast equality checks were used to reconcile between concurrent mutations in a highly scalable production system.

**BENEFIT #4: CONCURRENCY SAFETY FOR FREE**

In a multi threaded environment, concurrency safety mechanisms (e.g. mutexes) are often used to prevent the data in thread A from being modified while it is accessed in thread B.

In addition to the slight performance hit they cause, concurrency safety mechanisms impose a mental burden that makes code writing and reading much more difficult.

**TIP**

**Adherence to data immutability eliminates the need for a concurrency mechanism: the data you have in hand never changes!**

**A.3.4 Price for Principle #3**

Applying Principle #3 comes at a price:

- Performance hit
- Need a library for persistent data structures

**PRICE #1: PERFORMANCE HIT**

As mentioned earlier, there exist implementations of persistent data structures in most programming languages. But even the most efficient implementation is a bit slower than the in-place mutation of the data.

In most applications, the performance hit and the additional memory consumption involved in using immutable data structures is not significant. But it is something to keep in mind.

**PRICE #2: NEED A LIBRARY FOR PERSISTENT DATA STRUCTURES**

In a language like Clojure, the native data structures of the language are immutable. However, in most programming languages, adhering to data immutability requires the inclusion a third party library that provides an implementation of persistent data structures.

The fact that the data structures are not native to the language means that it is difficult (if not impossible) to enforce the usage of immutable data across the board.

Also, when integrating with third party libraries (e.g. a chart library), persistent data structures must be converted into equivalent native data structures.

### A.3.5 Summary

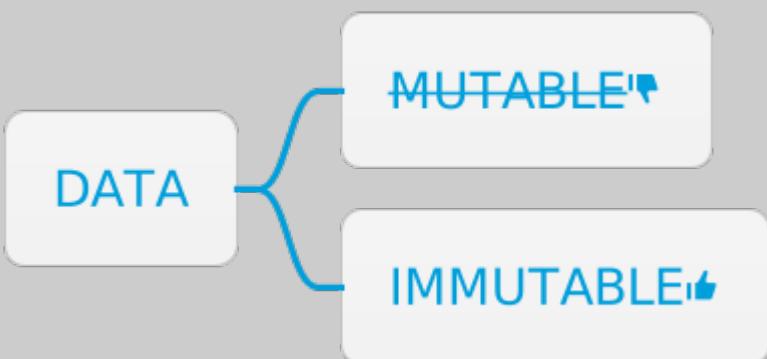
DOP considers data as a value that never changes. Adherence to this principle results in code that is predictable even in a multi threaded environment and equality checks are fast.

However, a non-negligible mind shift is required and in most programming languages a third party library is needed to provide an efficient implementation of persistent data structures.

**SIDE BAR**      **DOP Principle #3: Data is immutable**

**The Principle**

**Data is immutable.**



**Benefits**

- Data access to all with confidence
- Code behavior is predictable
- Equality check is fast
- Concurrency safety for free

**Price**

- Performance hit
- Need a library for persistent data structures

## A.4 DOP Principle #4: Separate data schema from data representation

### A.4.1 The principle in a nutshell

Now, with data separated from code and represented with generic and immutable data structures, comes the question of how to express the shape of the data.

In DOP, the expected shape is expressed as a data schema that is kept separated from the data itself.

**NOTE**

**Principle #4: Separate between data schema and data representation.**

The main benefit of Principle #4 is that it allows developers to decide what pieces of data should have a schema and what pieces of data should not.

### A.4.2 Illustration of Principle #4

Think about handling a request for the addition of an author to the system. To keep things simple, imagine that such a request contains only basic information about the author: their first name and last name and optionally the number of books they have written.

As seen in Principle #2 (Represent data with generic data structures), in DOP request data is represented as a string map where the map is expected to have three fields:

1. `firstName` - a string
2. `lastName` - a string
3. `books` - a number (optional)

In DOP, the *expected* shape of data is represented as data that is kept separated from the request data. For instance, JSON schema (<https://json-schema.org/>) can represent the data schema of the request with the map in [A.28](#).

**Listing A.28 The JSON schema for an `addAuthor` request data**

```
var addAuthorRequestSchema = {
  "type": "object",          ①
  "required": ["firstName", "lastName"], ②
  "properties": {
    "firstName": {"type": "string"}, ③
    "lastName": {"type": "string"},   ④
    "books": {"type": "integer"}     ⑤
  }
};
```

- ① data is expected to be a map (in JSON, a map is called an object)
- ② only `firstName` and `lastName` fields are required

- ③ firstName must be a string
- ④ lastName must be a string
- ⑤ books must be a number (when it is provided)

A data validation library is used to check whether a piece of data conforms to a data schema. For instance, Ajv JSON schema validator (<https://ajv.js.org/>) can be used to validate data using the validate function that returns `true` when data is valid and `false` when data is invalid as in the examples in [A.29](#).

### **Listing A.29 Data validation with Ajv**

```
var validAuthorData = {
  firstName: "Isaac",
  lastName: "Asimov",
  books: 500
};

ajv.validate(addAuthorRequestSchema, validAuthorData); // ❶
// true

var invalidAuthorData = {
  firstName: "Isaac",
  lastNam: "Asimov",
  books: "five hundred"
};

ajv.validate(addAuthorRequestSchema, invalidAuthorData); ❷
// false
```

- ❶ data is valid
- ❷ data has `lastNam` instead of `lastName` and `books` is a string instead of a number

When data is invalid, the details about data validation failures are available in a human readable format as shown in [A.30](#).

### **Listing A.30 Getting details about data validation failure in a human readable format**

```
var invalidAuthorData = {
  firstName: "Isaac",
  lastNam: "Asimov",
  books: "five hundred"
};

var ajv = new Ajv({allErrors: true}); ❶
ajv.validate(addAuthorRequestSchema, invalidAuthorData);
ajv.errorsText(ajv.errors); // <2>
// "data should have required property 'lastName', data.books should be number"
```

- ❶ by default, Ajv stores only the first data validation error. Set `allErrors: true` to store all errors.
- ❷ data validation errors are stored internally as an array. In order to get a human readable string, use the `errorsText` function.

### A.4.3 Benefits of Principle #4

Separation of data schema from data representation provides multiple benefits:

- Freedom to choose what data should be validated
- Optional fields
- Advanced data validation conditions
- Automatic generation of data model visualization

#### **BENEFIT #1: FREEDOM TO CHOOSE WHAT DATA SHOULD BE VALIDATED**

When data schema is separated from data representation data may be instantiated without specifying its expected shape. Such freedom is useful in various situations. For example:

- Rapid prototyping or experimentation
- Data has already been validated

#### **RAPID PROTOTYPING**

In classic OOP, every piece of data must be instantiated through a class. During the exploration phase of coding, when the final shape of our data is not yet known, being forced to update the class definition each time the data model is updated slows us down. DOP enables a faster pace during the exploration phase by delaying the data schema definition to a later phase.

#### **CODE REFACTORING**

One common refactoring pattern is split phase refactoring (<https://refactoring.com/catalog/splitPhase.html>) where a single large function is split into multiple smaller functions with private scope. Those functions are called with data that has already been validated by the large function. In DOP, it is not necessary to specify the shape of the arguments of the inner functions, relying on the data validation that has already occurred.

Consider how to display some information about an author, like their full name and whether they are considered prolific or not. Using the code shown earlier to illustrate Principle #2 to calculate the full name and the prolificity level of the author, one might come up with a `displayAuthorInfo` function as in [A.31](#).

### Listing A.31 Displaying author information

```

class NameCalculation {
    static fullName(data) {
        return data.firstName + " " + data.lastName;
    }
}

class AuthorRating {
    static isProlific (data) {
        return data.books > 100;
    }
}

var authorSchema = {
    "type": "object",
    "required": ["firstName", "lastName"],
    "properties": {
        "firstName": {"type": "string"},
        "lastName": {"type": "string"},
        "books": {"type": "integer"}
    }
};

function displayAuthorInfo(authorData) {
    if(!ajv.validate(authorSchema, authorData)) {
        throw "displayAuthorInfo called with invalid data";
    }
    console.log("Author full name is: ", NameCalculation.fullName(authorData));
    if(authorData.books == null) {
        console.log("Author has not written any book");
    } else {
        if (AuthorRating.isProlific(authorData)) {
            console.log("Author is prolific");
        } else {
            console.log("Author is not prolific");
        }
    }
}

```

Notice that the first thing done inside the body of `displayAuthorInfo` is to validate that the argument passed to the function is valid.

Now, apply the split phase refactoring pattern to this simple example and split the body of `displayAuthorInfo` in two inner functions:

1. `displayFullName`: Display the author full name
2. `displayProlificity`: Display whether the author is prolific or not

The resulting code is in [A.32](#).

### Listing A.32 Application of split phase refactoring pattern

```

function displayFullName(authorData) {
    console.log("Author full name is: ", NameCalculation.fullName(authorData));
}

function displayProlificity(authorData) {
    if(authorData.books == null) {
        console.log("Author has not written any book");
    } else {
        if (AuthorRating.isProlific(authorData)) {
            console.log("Author is prolific");
        } else {
            console.log("Author is not prolific");
        }
    }
}

function displayAuthorInfo(authorData) {
    if(!ajv.validate(authorSchema, authorData)) {
        throw "displayAuthorInfo called with invalid data";
    };
    displayFullName(authorData);
    displayProlificity(authorData);
}

```

Having data schema separated from data representation eliminates the need to specify a data schema for the arguments of the inner functions `displayFullName` and `displayProlificity`. It makes the refactoring process a bit smoother. In some cases, the inner functions are more complicated and it makes sense to specify a data schema for their arguments. DOP gives the freedom to choose!

### BENEFIT #2: OPTIONAL FIELDS

In OOP, allowing a class member to be optional is not easy. For instance, in Java one needs a special construct like the `Optional` class introduced in Java 8 (<https://www.oracle.com/technical-resources/articles/java/java8-optional.html>). In DOP, it is natural to declare a field as optional in a map. In fact in JSON schema, by default every field is optional. In order to make a field non-optional, its name must be included in the `required` array as for instance in the author schema in [A.33](#) where only `firstName` and `lastName` are required while `books` is optional. Notice that when an optional field is defined in a map, its value is validated against the schema.

### Listing A.33 A schema with an optional field

```

var authorSchema = {
    "type": "object",
    "required": ["firstName", "lastName"], ①
    "properties": {
        "firstName": {"type": "string"},
        "lastName": {"type": "string"},
        "books": {"type": "number"} ②
    }
};

```

- ① books is not included in required, as it is an optional field
- ② when present, books must be a number

Let's illustrate how the validation function deals with optional fields: A map without a books field is considered to be valid as shown in [A.34](#), while a map with a books field where the value is not a number is considered to be invalid as shown in [A.35](#).

#### **Listing A.34 A map without an optional field is valid**

```
var authorDataNoBooks = {
  "firstName": "Yehonathan",
  "lastName": "Sharvit"
};

ajv.validate(authorSchema, authorDataNoBooks); ①
// true
```

- ① The validation passes as books is an optional field

#### **Listing A.35 A map with an invalid optional field is invalid**

```
var authorDataInvalidBooks = {
  "firstName": "Albert",
  "lastName": "Einstein",
  "books": "Five"
};

validate(authorSchema, authorDataInvalidBooks); ①
// false
```

- ① The validation fails as books is not a number

### **BENEFIT #3: ADVANCED DATA VALIDATION CONDITIONS**

In DOP, data validation occurs at run time. It allows the definition of data validation conditions that go beyond the type of a field, such as validating that a field is not only a string but a string with a maximal number of characters or a number comprised in a range of numbers. The author schema in [A.36](#) expects firstName and lastName to be strings of less than 100 characters and books to be a number between 0 and 10,000.

### Listing A.36 A schema with advanced data validation conditions

```
var authorComplexSchema = {
    "type": "object",
    "required": ["firstName", "lastName"],
    "properties": {
        "firstName": {
            "type": "string",
            "maxLength": 100
        },
        "lastName": {
            "type": "string",
            "maxLength": 100
        },
        "books": {
            "type": "integer",
            "minimum": 0,
            "maximum": 10000
        }
    }
};
```

JSON schema supports many other advanced data validation conditions, like regular expression validation for string fields or number fields that should be a multiple of a given number.

### BENEFIT #4: AUTOMATIC GENERATION OF DATA MODEL VISUALIZATION

With the data schema defined as data, several tools can be leveraged to generate data model visualizations: with tools like JSON Schema Viewer (<https://navneethg.github.io/jsonschemaviewer/>) and Malli (<https://github.com/metosin/malli>) a UML diagram can be generated out of a JSON schema. For instance, the JSON schema in [A.37](#) defines the shape of a bookList field that is an array of books where each book is a map. It can be visualized as the UML diagram in [A.4](#). These tools generate the UML diagram from the JSON schema.

### Listing A.37 A JSON schema with an array of objects

```
{
    "type": "object",
    "required": ["firstName", "lastName"],
    "properties": {
        "firstName": {"type": "string"},
        "lastName": {"type": "string"},
        "bookList": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "title": {"type": "string"},
                    "publicationYear": {"type": "integer"}
                }
            }
        }
    }
}
```

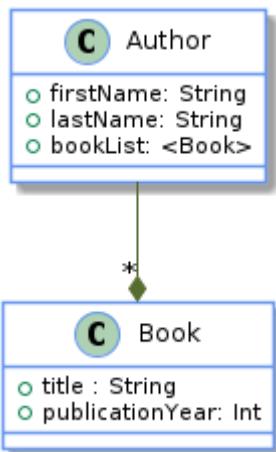


Figure A.4 A UML visualization of the JSON schema in [A.37](#)

#### A.4.4 Price for Principle #4

Applying Principle #4 comes at a price:

- Weak connection between data and its schema
- Small performance hit

##### **PRICE #1: WEAK CONNECTION BETWEEN DATA AND ITS SCHEMA**

By definition, when data schema and data representation are separated, the connection between data and its schema is weaker than when data is represented with classes. Moreover, the schema definition language (e.g. JSON schema) is not part of the programming language. It is up to the developer to decide where data validation is necessary and where it is superfluous. As the idiom says, with great power comes great responsibility.

##### **PRICE #2: LIGHT PERFORMANCE HIT**

As mentioned earlier, there exist implementations of JSON schema validation in most programming languages. In DOP, data validation occurs at run time and it takes some time to run the data validation while in OOP, data validation occurs usually at compile time.

This drawback is mitigated by the fact that even in OOP some parts data validation occur at run time. For instance, the conversion of a request JSON payload into an object occurs at run time. Moreover, in DOP it is quite common to have some data validation parts enabled only during development and to disable them when the system runs in production.

As a consequence, the performance hit is not significant.

## A.4.5 Summary

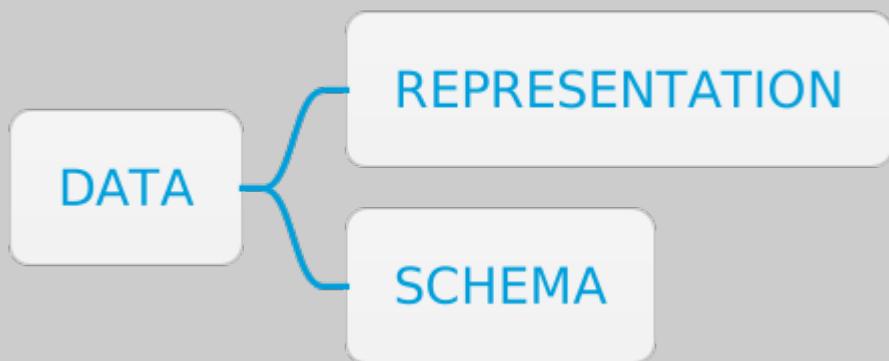
In DOP data is represented with immutable generic data structures. When additional information about the shape of the data is required, a data schema can be defined (e.g. in JSON Schema). Keeping the data schema separate from the data representation gives freedom to the developer to decide where data should be validated. Moreover, data validation occurs at run-time. As a consequence, data validation conditions that go beyond the static data types (e.g. the string length) can be expressed.

However, with great power comes great responsibility and it is up to the developer to remember to validate data.

### SIDE BAR      DOP Principle #4: Separate between data schema and data representation

#### The Principle

Separate between data schema and data representation



#### Benefits

- Freedom to choose what data should be validated
- Optional fields
- Advanced data validation conditions
- Automatic generation of data model visualization

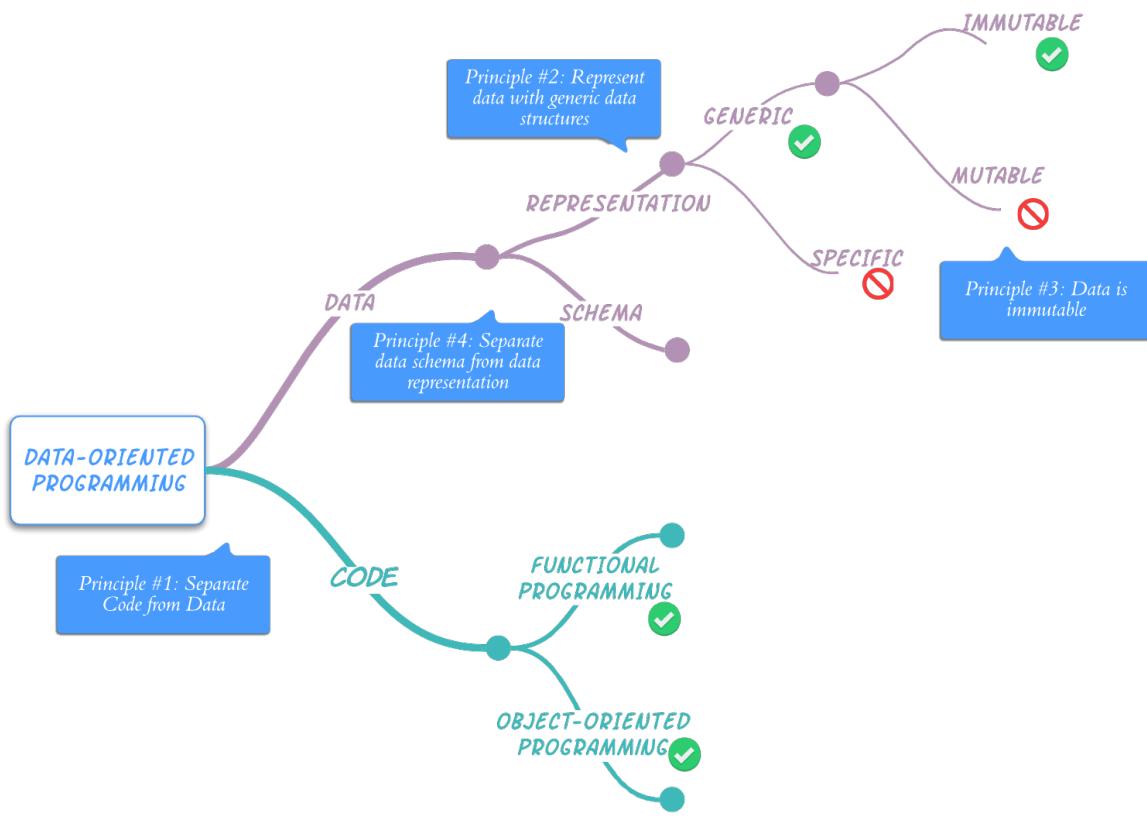
#### Price

- Weak connection between data and its schema
- Small performance hit

## A.5 Conclusion

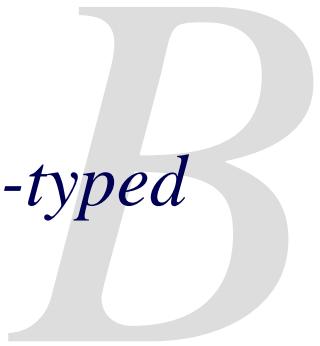
Data-Oriented programming simplifies the design and implementation of information systems by treating data as a first class citizen. This is made possible by adhering to 4 language agnostic core principles:

1. Separate code from data
2. Represent data with generic data structures
3. Data is immutable
4. Separate between data schema and data representation



**Figure A.5 The principles of Data-Oriented programming**

This appendix has illustrated how each principle can be applied both in FP and OOP languages, and describes at a high level the benefits of each principle and the costs of adherence to it.



# *Generic data access in statically-typed languages*

Representing data with generic data structures fits naturally in dynamically-typed programming languages like JavaScript, Ruby or Python. However, in statically-typed programming languages like Java or C#, representing data as string maps with values of unspecified type is not natural for several reasons:

1. Accessing map fields requires a type cast
2. Map field names are not validated at compile-time
3. Auto-completion and other convenient IDE features are not available

This appendix explores various ways to improve access to generic data in statically-typed languages:

1. Value getters for maps to avoid type casting when accessing map fields
2. Typed getters for maps to benefit from compile-time check for map field names
3. Generic access for classes using reflection to benefit from auto-completion and other convenient IDE features

## B.1 Dynamic getters for string maps

Let's start with a refresher about the approach we presented in Part 1, namely the representation of records as string maps and accessing map fields with dynamic getters and type casting.

**WARNING** Most of the code snippets in this appendix use Java but the approaches illustrated can be applied to other Object-Oriented statically-typed languages, like C# or Go.

### B.1.1 Accessing non-nested map fields with dynamic getters

Throughout this appendix, we will illustrate various ways to provide generic data access using a book record, made of:

- title: a string
- isbn: a string
- publicationYear: an integer

[B.1](#) shows the representation of two books "Watchmen" and "Seven Habits of highly effective people" in Java as string maps, whose values are of type Object.

### Listing B.1 Two books represented as maps

```
Map watchmenMap = Map.of(
    "isbn", "978-1779501127",
    "title", "Watchmen",
    "publicationYear", 1987
);

Map sevenHabitsMap = Map.of(
    "isbn", "978-1982137274",
    "title", "7 Habits of Highly Effective People",
    "publicationYear", 2020
);
```

The map fields can be accessed generically using a dynamic getter, whose implementation is shown in [B.2](#).

### Listing B.2 The implementation of dynamic getter for map fields

```
class DynamicAccess {
    static Object get(Map m, String k) {
        return (m).get(k);
    }
}
```

The drawback of dynamic getters is that a type cast is required to manipulate the value of a map field. For instance, as shown in [B.3](#), a cast to String is needed to call the `toUpperCase` string method on the `title` field value.

### Listing B.3 Accessing map fields with a dynamic getter and type casting

```
((String)DynamicAccess.get(watchmenMap, "title")).toUpperCase();
// "WATCHMEN"
```

Dynamic getters provide generic data access in the sense that they do not require specific knowledge of the type of data the string map represents. As a consequence, the name of the field can be received dynamically (e.g. from the user) as shown in [B.4](#). This works because in order to access a Book data field in a string map, it is not necessary to import the class that defines the Book.

## Listing B.4 Mapping over a list of maps with a dynamic getter and type casting

```
var books = List.of(watchmenMap, sevenHabitsMap);
var fieldName = "title";

books.stream()
.map(x -> DynamicAccess.get(x, fieldName))
.map(x -> ((String)x).toUpperCase())
.collect(Collectors.toList())

// [ "WATCHMEN", "7 HABITS OF HIGHLY EFFECTIVE PEOPLE" ]
```

Another aspect of the genericity of dynamic getters is that they work on any type of data. For instance, the dynamic getter for `title` works not only on books but on any piece of data that has a `title` field.

### B.1.2 Accessing nested map fields with dynamic getters

Suppose that search results represent as a string map where:

1. Keys are book ISBNs
2. Values are book data represented as string maps as in the previous section.

[B.5](#) presents an example of search results.

## Listing B.5 Search results represented as a map

```
Map searchResultsMap = Map.of(
    "978-1779501127", Map.of(
        "isbn", "978-1779501127",
        "title", "Watchmen",
        "publicationYear", 1987
    ),
    "978-1982137274", Map.of(
        "isbn", "978-1982137274",
        "title", "7 Habits of Highly Effective People",
        "publicationYear", 2020
    )
);
```

Book fields are nested in the search result map. In order to access nested map fields, a `get` method is added to the `DynamicAccess` class that receives a list of strings that represents the information path of the nested map field as in [B.6](#).

## Listing B.6 The implementation of dynamic getter for nested map fields

```
class DynamicAccess {
    static Object get(Map m, String k) {
        return (m).get(k);
    }

    static Object get(Map m, List<String> path) {
        Object v = m;
        for (String k : path) {
            v = get((Map)v, k);
            if (v == null) {
                return null;
            }
        }
        return v;
    }
}
```

As with non-nested map fields in the previous section, type casting is required to manipulate a nested map field, as in [B.7](#).

## Listing B.7 Accessing nested map fields with dynamic getter and type casting

```
((String)DynamicAccess.get(searchResultsMap,
    List.of("978-1779501127", "title"))).toUpperCase();
// "WATCHMEN"
```

The next section shows how to avoid type casting when manipulating values in string maps.

## B.2 Value getters for maps

The simplest way to avoid type casting when manipulating the value of a string map field is to use a dynamic data type as shown in Appendix A. Dynamic data types are supported in languages like C# but not in languages like Java.

Next is illustration of how value getters make it possible to avoid type casting.

### B.2.1 Accessing non-nested map fields with value getters

In this section, books are still represented as string maps with values of type `Object`, as in [B.8](#).

## Listing B.8 Two books represented as maps

```
Map watchmenMap = Map.of(
    "isbn", "978-1779501127",
    "title", "Watchmen",
    "publicationYear", 1987
);

Map sevenHabitsMap = Map.of(
    "isbn", "978-1982137274",
    "title", "7 Habits of Highly Effective People",
    "publicationYear", 2020
);
```

The idea of value getters is quite simple: instead of doing the type casting outside the getter, it is done inside the getter. A value getter is required for every type: `getAsString` for strings, `getAsInt` for integers, `getAsFloat` for float numbers, `getAsBoolean` for boolean values etc...

The value getter approach is used by Java libraries like Apache Wicket (<https://ci.apache.org/projects/wicket/apidocs/9.x/org/apache/wicket/util/value/ValueMap.html>) and Gson (<https://github.com/google/gson>).

[B.9](#) shows an implementation for `getAsString` that retrieves a map field value as a string.

### Listing B.9 The implementation of value getter for map fields

```
class DynamicAccess {
    static Object get(Map m, String k) {
        return (m).get(k);
    }

    static String getAsString(Map m, String k) {
        return (String) get(m, k);
    }
}
```

Now a map field can be accessed without type casting, for instance using `getAsString` to manipulate a book title as in [B.10](#).

### Listing B.10 Accessing non-nested fields with value getter

```
DynamicAccess.getAsString(watchmenMap, "title").toUpperCase();
// "WATCHMEN"
```

Mapping over books with a value getter is a bit more convenient without type casting, as in [B.11](#).

### Listing B.11 Mapping over a list of maps with a value getter

```
var books = List.of(watchmenMap, sevenHabitsMap);

books.stream()
.map(x -> DynamicAccess.getAsString(x, "title"))
.map(x -> x.toUpperCase())
.collect(Collectors.toList())
// [ "WATCHMEN", "7 HABITS OF HIGHLY EFFECTIVE PEOPLE" ]
```

## B.2.2 Accessing nested map fields with value getters

The value getter approach applies naturally to nested map fields.

As in the dynamic getter section, suppose that search results are represented as a string map where:

1. Keys are book ISBNs
2. Values are book data represented as string maps as in the previous section.

[B.12](#) presents an example of search results.

### Listing B.12 Search results represented as a map

```
Map searchResultsMap = Map.of(
    "978-1779501127", Map.of(
        "isbn", "978-1779501127",
        "title", "Watchmen",
        "publicationYear", 1987
    ),
    "978-1982137274", Map.of(
        "isbn", "978-1982137274",
        "title", "7 Habits of Highly Effective People",
        "publicationYear", 2020
    )
);
```

Book fields are nested in the search results map. In order to access nested map fields without type casting, a `getAsString` method is added to the `DynamicAccess` class that receives a list of strings that represents the information path of the nested map field as in [B.13](#).

### Listing B.13 The implementation of value getter for nested map fields

```
class DynamicAccess {
    static Object get(Map m, String k) {
        return (m).get(k);
    }

    static Object get(Map m, List<String> p) {
        Object v = m;
        for (String k : p) {
            v = get((Map)v, k);
            if (v == null) {
                return null;
            }
        }
        return v;
    }

    static String getAsString(Map m, String k) {
        return (String)get(m, k);
    }

    static String getAsString(Map m, List<String> p) {
        return (String)get(m, p);
    }
}
```

With nested value getter, book titles can be manipulated inside search results without type casting, as in [B.14](#).

### Listing B.14 Accessing nested map fields with value getter

```
var informationPath = List.of("978-1779501127", "title");

DynamicAccess.getAsString(searchResultsMap, informationPath)
    .toUpperCase();
// "WATCHMEN"
```

Value getters make data access a bit more convenient when avoiding type casting. The next

section shows how typed getters make it possible to benefit from compile-time checks even when data is represented as string maps.

## B.3 Typed getters for maps

**WARNING** The typed getter approach is applicable in statically-typed languages that support generic types (like Java and C#). In this section, the typed getter approach in Java is illustrated.

### B.3.1 Accessing non-nested map fields with typed getters

As in the two previous sections, in [B.8](#), is the representation of two books "Watchmen" and "Seven Habits of highly effective people" in Java as string maps, whose values are of type Object.

#### **Listing B.15 Two books represented as maps**

```
Map watchmenMap = Map.of(
    "isbn", "978-1779501127",
    "title", "Watchmen",
    "publicationYear", 1987
);

Map sevenHabitsMap = Map.of(
    "isbn", "978-1982137274",
    "title", "7 Habits of Highly Effective People",
    "publicationYear", 2020
);
```

The idea of typed getters is to create a generic object that contains information about:

1. The field name
2. The type of the field value

Then, this object can be used on a string map to retrieve the typed value of the field in the map.

For example, in [B.16](#) there is a typed getter named TITLE that retrieves the value of a field named title as a string.

#### **Listing B.16 Accessing map fields with a typed getter**

```
Getter<String> TITLE = new Getter("title");
TITLE.get(watchmenMap).toUpperCase();
// "WATCHMEN"
```

The implementation of typed getter is in [B.17](#).

### **Listing B.17 The implementation of a typed getter**

```
class Getter <T> {
    private String key;

    public <T> Getter (String k) {
        this.key = k;
    }

    public T get (Map m) {
        return (T)(DynamicAccess.get(m, key));
    }
}
```

**TIP**      **Typed getters are generic objects. Unlike value getters from the previous section, it is not necessary to provide an implementation for every type.**

In a sense, typed getters support compile-time validation and auto-completion. If the name of the typed getter `TITLE` is misspelled, the compiler will throw an error. Typing the first few letters of `TITLE` into an IDE will provide auto-completion of the symbol of the typed getter.

However, when you instantiate a typed getter, the field name must be passed as a string and neither compile-time checks nor auto-completion are available.

Mapping over a list of maps with a typed getter is quite simple as seen in [B.18](#).

### **Listing B.18 Mapping over a list of maps with a typed getter**

```
var books = List.of(watchmenMap, sevenHabitsMap);

books.stream()
.map(x -> TITLE.get(x))
.map(x -> x.toUpperCase())
.collect(Collectors.toList())
// [ "WATCHMEN", "7 HABITS OF HIGHLY EFFECTIVE PEOPLE" ]
```

### **B.3.2 Accessing nested map fields with typed getters**

The typed getter approach extends well to nested map fields.

As in the value getter section, suppose that search results are represented as a string map where:

1. Keys are book ISBNs
2. Values are book data represented as string maps as in the previous section.

[B.19](#) presents an example of search results.

## Listing B.19 Search results represented as a map

```
Map searchResultsMap = Map.of(
    "978-1779501127", Map.of(
        "isbn", "978-1779501127",
        "title", "Watchmen",
        "publicationYear", 1987
    ),
    "978-1982137274", Map.of(
        "isbn", "978-1982137274",
        "title", "7 Habits of Highly Effective People",
        "publicationYear", 2020
    )
);
```

In order to support nested map fields, a constructor is added to the `Getter` class that receives a list of strings (representing the information path) as in [B.20](#).

## Listing B.20 The implementation of a nested typed getter

```
class Getter <T> {
    private List<String> path;
    private String key;
    private boolean nested;

    public <T> Getter (List<String> path) {
        this.path = path;
        nested = true;
    }

    public <T> Getter (String k) {
        this.key = k;
        nested = false;
    }

    public T get (Map m) {
        if(nested) {
            return (T)(DynamicAccess.get(m, path));
        }
        return (T)(DynamicAccess.get(m, key));
    }
}
```

Nested map fields are manipulated with typed getters without any type casting, as in [B.21](#).

## Listing B.21 Accessing nested map fields with typed getter

```
var informationPath = List.of("978-1779501127",
    "title");

Getter<String> NESTED_TITLE = new Getter(informationPath);
NESTED_TITLE.get(searchResultsMap).toUpperCase();
// "WATCHMEN"
```

Typed getters provide several benefits:

1. No type casting is required
2. No need for implementing a getter for each and every type
3. Compile-time validation at usage time

#### 4. Auto-completion at usage time

However, at creation time, map fields are accessed as strings. The next section illustrates how to provide generic access when data is represented not as string maps but as classes.

## B.4 Generic access to class members

A totally different approach to providing generic access to data is to represent data with classes as in traditional OOP and to leverage reflection in order to provide generic data access.

**WARNING** The generic access to class members approach is applicable in statically-typed languages that support reflection (like Java and C#). This section illustrates the approach in Java.

### B.4.1 Generic access to non-nested class members

Instead of representing data as string maps, data can be represented as classes with data members only and provide generic access to the class members via reflection. This approach is interesting only read data access is needed. However, when creating new versions of data or adding new data fields, it is better to represent data with maps as in Part 1.

**WARNING** The approach presented in this section is applicable only for read data access.

In order to represent a book as a class, make sure that:

1. The class has only data members (no methods)
2. The members are public
3. The members are immutable
4. A proper implementation of `hashCode()`, `equals()` and `toString()` methods

For instance, in Java, mark the members with `public` and `final` as in [B.22](#) (the implementation of `hashCode()`, `equals()` and `toString()` methods are omitted for the sake of simplicity).

## Listing B.22 Representing books with a class

```
public class BookData {
    public final String isbn;
    public final String title;
    public final Integer publicationYear;
    public BookData (
        String isbn,
        String title,
        Integer publicationYear) {
        this.isbn = isbn;
        this.title = title;
        this.publicationYear = publicationYear;
    }

    public boolean equals(Object o) {
        // Omitted for sake of simplicity
    }

    public int hashCode() {
        // Omitted for sake of simplicity
    }

    public String toString() {
        // Omitted for sake of simplicity
    }
}
```

Since Java 14, there is a simpler way to represent data using data records (<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Record.html>). Data records provide:

1. Private `final` fields
2. Public read accessors
3. Public constructor whose signature is derived from the record component list
4. Implementations of `equals()` and `hashCode()` methods which specify that two records are equal if they are of the same type and their record components are equal
5. Implementation of `toString()` method that includes the string representation of the record components with their names

## Listing B.23 Representing books with a record

```
public record BookData (String isbn,
                      String title,
                      Integer publicationYear
                    ) {}
```

Let's create two objects (or records) for "Watchmen" and "Seven habits of highly effective people", as in [B.24](#).

### **Listing B.24 Two book records**

```
BookData watchmenRecord = new BookData(
    "978-1779501127",
    "Watchmen",
    1987
);

BookData sevenHabitsRecord = new BookData(
    "978-1982137274",
    "7 Habits of Highly Effective People",
    2020
);
```

The traditional way to access a data member is via its accessor, e.g. `watchmen.title()` to retrieve the title of "Watchmen". In order to access a data member whose name comes from a dynamic source like a variable (or as part of a request payload), we need to use reflection ([https://en.wikipedia.org/wiki/Reflective\\_programming](https://en.wikipedia.org/wiki/Reflective_programming)). In java, accessing the `title` field in a book looks like the code snippet in [B.25](#).

### **Listing B.25 Accessing a data member via reflection**

```
watchmenRecord
    .getClass()
    .getDeclaredField("title")
    .get(watchmenRecord)
// "watchmen"
```

Reflection can be used to provide access to any data member using the code in [B.26](#).

### **Listing B.26 The implementation of dynamic access to non-nested class members**

```
class DynamicAccess {
    static Object get(Object o, String k) {
        if(o instanceof Map) {
            return ((Map)o).get(k);
        }
        try {
            return (o.getClass()).getDeclaredField(k).get(o);
        } catch (IllegalAccessException | NoSuchFieldException e) {
            return null;
        }
    }

    static String getAsString(Object o, String k) {
        return (String)get(o, k);
    }
}
```

And now, data members are accessible with the same genericity and dynamism as fields in a string map, as in [B.27](#).

### **Listing B.27 Accessing dynamically a class member**

```
((String)DynamicAccess.get(watchmenRecord, "title")).toUpperCase();
// "WATCHMEN"
```

Without any code modification, value getters presented earlier in this appendix in the context of string maps work with classes and records as in [B.28](#).

### **Listing B.28 Accessing a class member with a value getter**

```
DynamicAccess.getAsString(watchmenRecord, "title").toUpperCase();
// "WATCHMEN"
```

It is possible to map over a list of objects without having to import the class definition of the objects we map over, as in [B.29](#).

### **Listing B.29 Mapping over a list of objects with a value getter**

```
var books = List.of(watchmenRecord, sevenHabitsRecord);

books.stream()
.map(x -> DynamicAccess.getAsString(x, "title"))
.map(x -> x.toUpperCase())
.collect(Collectors.toList())
// ["WATCHMEN", "7 HABITS OF HIGHLY EFFECTIVE PEOPLE"]
```

The typed getters we introduced earlier in the appendix can be used on objects, as in [B.30](#).

### **Listing B.30 Mapping over a list of objects with a typed getter**

```
var books = List.of(watchmenRecord, sevenHabitsRecord);

books.stream()
.map(x -> TITLE.get(x))
.map(x -> x.toUpperCase())
.collect(Collectors.toList())
// ["WATCHMEN", "7 HABITS OF HIGHLY EFFECTIVE PEOPLE"]
```

## **B.4.2 Generic access to nested class members**

The previous section showed how to provide the same data access to classes as we have for string maps. It becomes powerful when we combine classes and maps.

Suppose search results are represented as a map, where:

1. Keys are book ISBNs (strings)
2. Values are book data represented as data classes (or records) as in the previous section.

An example of the search results is shown in [B.31](#).

### Listing B.31 Search results represented as a map of records

```
Map searchResultsRecords = Map.of(
    "978-1779501127", new BookData(
        "978-1779501127",
        "Watchmen",
        1987
    ),
    "978-1982137274", new BookData(
        "978-1982137274",
        "7 Habits of Highly Effective People",
        2020
    )
);
```

It is necessary to add `get` and `getAsString()` methods that receive a list of strings, as in [B.32](#).

### Listing B.32 The implementation of value getter for nested class members

```
class DynamicAccess {
    static Object get(Object o, String k) {
        if(o instanceof Map) {
            return ((Map)o).get(k);
        }
        try {
            return (o.getClass().getDeclaredField(k).get(o));
        } catch (IllegalAccessException | NoSuchFieldException e) {
            return null;
        }
    }

    static Object get(Object o, List<String> p) {
        Object v = o;
        for (String k : p) {
            v = get(v, k);
        }
        return v;
    }

    static String getAsString(Object o, String k) {
        return (String)get(o, k);
    }

    static String getAsString(Object o, List<String> p) {
        return (String)get(o, p);
    }
}
```

Now, a data member that is nested inside a string map can be accessed through its information path, as for instance in [B.6](#).

### Listing B.33 Accessing a member of a class nested in a map with a value getter

```
var informationPath = List.of("978-1779501127", "title");
DynamicClassAccess
    .getAsString(searchResultsRecords, informationPath)
    .toUpperCase();
// "WATCHMEN"
```

There is a second kind of nested data members when a data member is itself an object. For

instance, a `bookAttributes` field can be made from a `BookAttributes` class, as in [B.34](#).

### **Listing B.34 Representing book attributes with a nested class**

```
public class BookAttributes {
    public Integer numberOfPages;
    public String language;
    public BookAttributes(Integer numberOfPages, String language) {
        this.numberOfPages = numberOfPages;
        this.language = language;
    }
}

public class BookWithAttributes {
    public String isbn;
    public String title;
    public Integer publicationYear;
    public BookAttributes attributes;
    public Book (
        String isbn,
        String title,
        Integer publicationYear,
        Integer numberOfPages,
        String language) {
        this.isbn = isbn;
        this.title = title;
        this.publicationYear = publicationYear;
        this.attributes = new BookAttributes(numberOfPages, language);
    }
}
```

[B.35](#) shows an example of a nested class.

### **Listing B.35 An instance of a nested class**

```
BookData sevenHabitsNestedRecord = new BookWithAttributes(
    "978-1982137274",
    "7 Habits of Highly Effective People",
    2020,
    432,
    "en"
);
```

Value getters work without any modification on nested data members, as in [B.36](#).

### **Listing B.36 Accessing a nested class member with a value getter**

```
var informationPath = List.of("attributes",
                             "language");
DynamicClassAccess.getAsString(sevenHabitsNestedRecord, informationPath)
.toUpperCase();
// "EN"
```

## **B.4.3 Automatic JSON serialization of objects**

An approach similar to the one illustrated in this section is leveraged by JSON serialization libraries like Gson (<https://github.com/google/gson>) in order to serialize objects to JSON automatically. Gson leverages reflection to go over the class members and generate a JSON representation of each member value.

An example of Gson in action is shown in [B.37](#).

### **Listing B.37 JSON serialization of an object with Gson**

```
import com.google.gson.*;
var gson = new Gson();

BookData sevenHabitsRecord = new BookData(
    "978-1982137274",
    "7 Habits of Highly Effective People",
    2020
);

System.out.println(gson.toJson(sevenHabitsRecord));
// {"isbn":"978-1982137274","title":"7 Habits of Highly Effective People","publicationYear":2020}
```

It works also with objects nested in maps, as in [B.37](#), and with objects nested in objects, as in [B.39](#).

### **Listing B.38 JSON serialization of objects nested in a map with Gson**

```
Map searchResultsRecords = Map.of(
    "978-1779501127", new BookData(
        "978-1779501127",
        "Watchmen",
        1987
    ),
    "978-1982137274", new BookData(
        "978-1982137274",
        "7 Habits of Highly Effective People",
        2020
    )
);

System.out.println(gson.toJson(searchResultsRecords));
// {"978-1779501127":{"isbn":"978-1779501127","title":"Watchmen",\n// "publicationYear":1987}, "978-1982137274":{"isbn":"978-1982137274"\n// , "title":"7 Habits of Highly Effective People", "publicationYear":2020}}
```

### **Listing B.39 JSON serialization of an object nested in an object with Gson**

```
BookData sevenHabitsNestedRecord = new BookWithAttributes(
    "978-1982137274",
    "7 Habits of Highly Effective People",
    2020,
    432,
    "en"
);

System.out.println(gson.toJson(sevenHabitsNestedRecord));
// {"isbn":"978-1982137274","title":"7 Habits of Highly Effective \n//People", "publicationYear":2020, "attributes":{"numberOfPages":432,\n// "language": "en"}}
```

## **B.5 Summary**

This appendix has presented various ways to provide generic data access in statically-typed programming languages. Data can be represented either as string maps or as classes (or records) and benefit from generic data access via:

1. Dynamic getters
2. Value getters
3. Typed getters
4. Reflection

A summary of the benefits and drawbacks of each approach is presented in [B.1](#).

**Table B.1 Various ways to provide generic data access in statically-typed programming languages**

Approach	Representation	Benefits	Drawbacks
Dynamic getters	Map	Generic access	Requires type casting
Value getters	Map	No type casting	Implementation per type
Typed getters	Map	Compile-time validation on usage	No compile-time validation on creation
Reflection	Class	Full compile-time validation	Not modifiable



## *Data-Oriented Programming: A link in the chain of programming paradigms*

Data-Oriented Programming has its origins in the 1950s and the invention of LISP and is based on a set of best practices that can be found in both Functional Programming and Object-Oriented Programming. However, this paradigm has only been applicable in production systems at scale since the 2010s and the implementation of efficient persistent data structures.

This appendix traces the major ideas and discoveries which, over the years, have led to the emergence of DOP.

## C.1 Time-line

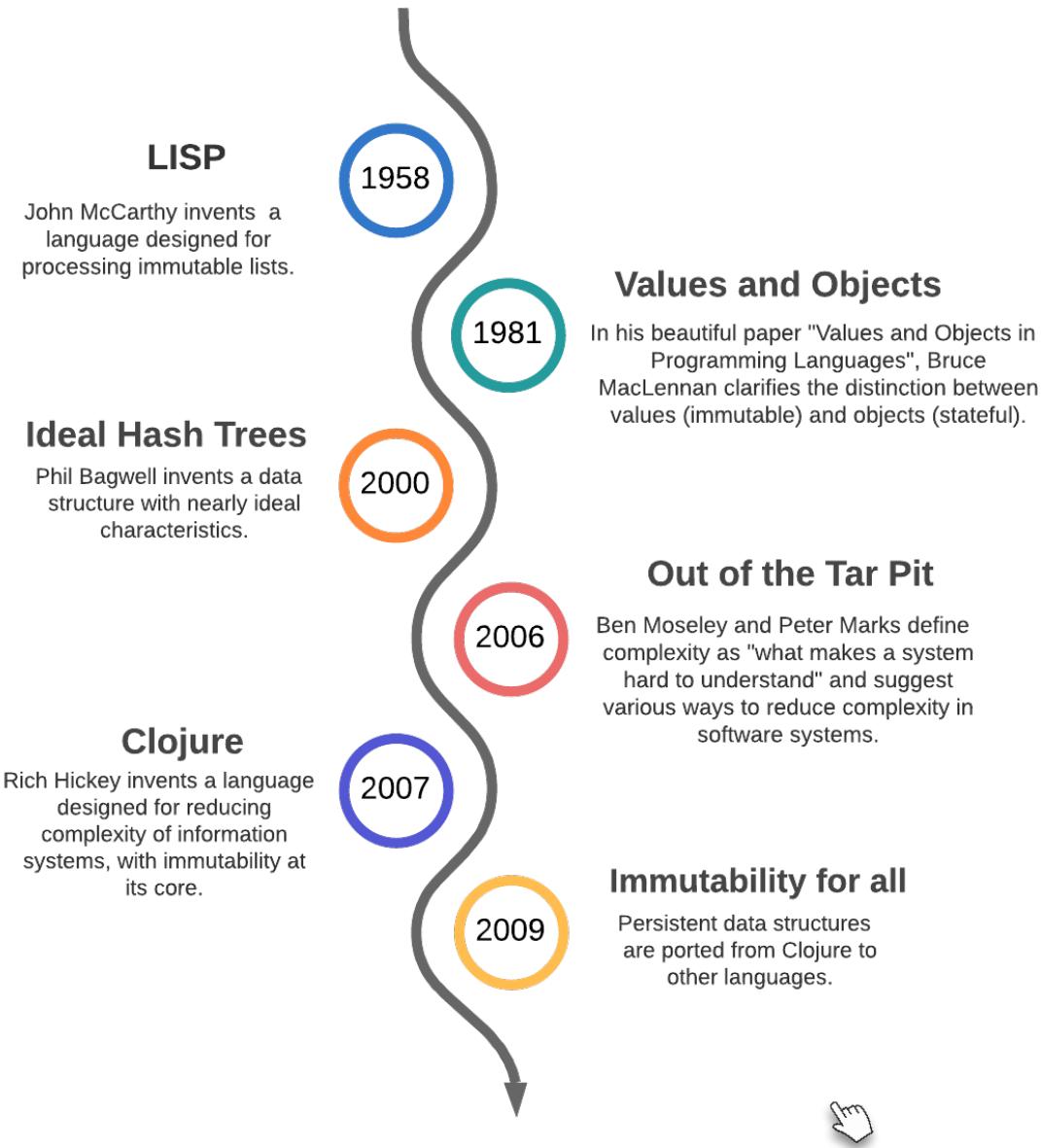


Figure C.1 Data-Oriented Programming Time-line

### C.1.1 1958: LISP

In LISP, John McCarthy has the ingenious idea to represent data as generic immutable lists and to invent a language that makes it very natural to create lists and to access any part of a list. That's the reason why LISP stands for LISt Processing.

In a way, LISP lists are the ancestors of JavaScript object literals. The idea that it makes sense to represent data with generic data structures (DOP Principle #2) definitely comes from LISP.

The main limitation of LISP lists is that when we update a list, we need to create a new version by cloning it, which has a negative impact on performance both in terms of CPU and memory.

### C.1.2 1981: Values and Objects

In a beautiful, short and easy-to-read paper named "Values and Objects in Programming Languages", <sup>13</sup> Bruce MacLennan clarifies the distinction between values and objects. In a nutshell:

- Values are timeless abstractions for which the concepts of updating, sharing and instantiation have no meaning. For instance, numbers are values.
- Objects exist in time and hence can be created, destroyed, copied, shared and updated. For instance, an employee in a human resource software system is an object.

**TIP**

The meaning of the term object in this paper is not exactly the same as in the context of Object-Oriented Programming.

The author explains why it's much simpler to write code that deals with values than code that deals with objects.

This paper has been a source of inspiration for Data-Oriented Programming as it encourages us to implement our systems in such a way that most of our code deals with values.

### C.1.3 2000: Ideal Hash Trees

Phil Bagwell invented a data structure called Hash Array Mapped Trie (HAMT). In his paper "Ideal Hash trees" (<https://lampwww.epfl.ch/papers/idealhashtrees.pdf>), he used HAMT to implement hash maps with nearly ideal characteristics both in terms of computation and memory usage.

As we have illustrated in Chapter 9, HAMT and Ideal hash trees are the foundation of efficient persistent data structures.

### C.1.4 2006: Out of the Tar Pit

In Out of the Tar Pit, <sup>14</sup> Ben Moseley and Peter Marks claim that complexity is the single major difficulty in the development of large-scale software systems. In the context of their paper, complexity means "that which makes large systems hard to understand".

The main insight of the authors is that most of the complexity of software systems is not essential but accidental: the complexity doesn't come from the problem we have to solve but from the software constructs we use to solve the problem. They suggest various ways to reduce complexity of software systems.

In a sense, Data-Oriented Programming is a way to get us out of the tar pit.

### C.1.5 2007: Clojure

Rich Hickey, an Object-Oriented Programming expert, invented Clojure to make it easier to develop information systems at scale. Rich Hickey likes to summarize Clojure's core value with the phrase: "Just use maps!". By maps, he means immutable maps to be manipulated efficiently by generic functions. Those maps were implemented using the data structures presented by Phil Bagwell in "Ideal Hash Trees".

Clojure has been the main source of inspiration for Data-Oriented Programming. In a sense, this book is a formalization of the underlying principles of Clojure and how to apply them in other programming languages.

### C.1.6 2009: Immutability for all

Clojure's efficient implementation of persistent data structures has been attractive for developers from other programming languages. In 2009, there were ported to Scala. Over the years, they have been ported to other programming languages either by organizations (like Facebook for Immutable.js) or by individual contributors (like Glen Peterson for Paguro in Java).

Nowadays, DOP is applicable in virtually any programming language!

## C.2 DOP principles as best practices

The principles of Data-Oriented programming as we have presented them through the book and formalized in Appendix A are not new. They come from best practices that are well-known among software developers from various programming languages. The *innovation* of Data-Oriented programming is the combination of those principles into a cohesive whole.

In this section, we put each of the 4 DOP principles into its broader scope.

### C.2.1 Principle #1: Separate code from data

Separating code from data used to be the main point of tension between Object-Oriented Programming (OOP) and Functional Programming (FP). Traditionally, in OOP we encapsulate data together with code in stateful objects, while in FP we write stateless functions that receive data they manipulate as an explicit argument.

This tension has been reduced over the years as it is possible in FP to write stateful functions with data encapsulated in their lexical scope ([https://en.wikipedia.org/wiki/Scope\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Scope_(computer_science))). Moreover, OOP languages like Java and C# have added support for anonymous functions (lambdas).

## C.2.2 Principle #2: Represent data with generic data structures

One of the main innovations of JavaScript when it was released in December 1995 was the ease of creating and manipulating hash maps via object literals. The increasing popularity of JavaScript over the years as a language used everywhere (front-end, back-end, desktop) has influenced the developer community to represent data with hash maps when possible. It feels more natural in dynamically-typed programming languages, but as we have seen in Appendix B, it is applicable also in statically-typed programming languages.

## C.2.3 Principle #3: Data is immutable

Data immutability is considered a best practice as it makes the behavior of our program more predictable. For instance, in "Effective Java" (<https://www.oreilly.com/library/view/effective-java/9780134686097>), Joshua Block mentions "Minimize mutability" as one of Java best practices.

There is a famous quote from Alan Kay - who is considered by many as the inventor of Object-Oriented Programming - about the value of immutability:

**TIP**

The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. Instead, the objects should be presented as site of higher level behaviors more appropriate for use as dynamic components. (...) It is unfortunate that much of what is called "object-Oriented programming" today is simply old style programming with fancier constructs. Many programs are loaded with "assignment-style" operations now done by more expensive attached procedures.

Unfortunately, until 2007 and the implementation of efficient persistent data structures in Clojure, immutability was not applicable for production applications at scale.

As we have already mentioned in Chapter 9, nowadays, efficient persistent data structures are available in most programming languages (see [C.1](#)).

**Table C.1 Persistent data structure libraries**

Language	Library
Java	Paguro ( <a href="https://github.com/GlenKPeterson/Paguro">https://github.com/GlenKPeterson/Paguro</a> )
C#	Provided by the language ( <a href="https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/march/net-framework-immutable-collections">https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/march/net-framework-immutable-collections</a> )
JavaScript	Immutable.js ( <a href="https://immutable-js.com/">https://immutable-js.com/</a> )
Python	Pyrsistent ( <a href="https://github.com/tobgu/pyrsistent">https://github.com/tobgu/pyrsistent</a> )
Ruby	Hamster ( <a href="https://github.com/hamstergem/hamster">https://github.com/hamstergem/hamster</a> )

In addition, many languages provide support for read-only objects natively. Java added record

classes (<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Record.html>) in Java 14. C# introduced a `record` type in C# 9. There is a ECMAScript proposal for supporting immutable records and tuples in JavaScript (<https://github.com/tc39/proposal-record-tuple>). Python 3.7 introduced Immutable data classes (<https://docs.python.org/3/library/dataclasses.html>).

#### C.2.4 Principle #4: Separate data schema from data representation

One of the more virulent critiques against dynamically-typed programming languages used to be related to the lack of data validation. The answer that dynamically-typed languages used to give to this critique was that you trade data safety for data flexibility.

Since the development of data schema languages like JSON schema (<https://json-schema.org/>), it is natural to validate data even when data is represented as hash maps. As we have seen in Chapters 7 and 12, data validation is not only possible but in some sense it is more powerful than when data is represented with classes.

### C.3 DOP and other data-related paradigms

In this section, we clarify the distinction between Data-Oriented Programming and two other programming paradigms whose name contain the term *data*: Data-Oriented Design and Data-Driven Programming.

QUOTE: "There are only two hard things in Computer Science: cache invalidation and naming things." - Phil Karlton

Each paradigm has its own objective and pursues it by focusing on a different aspect of data, as it is summarized on [C.2](#).

**Table C.2 Data-related paradigms: objectives and main data aspect focus**

Paradigm	Objective	Main data aspect focus
Data-Oriented design	Increase performance	Data layout
Data-Driven programming	Increase clarity	Behavior described by data
Data-Oriented programming	Reduce complexity	Data representation

#### C.3.1 Data-Oriented Design

Data-Oriented Design is a program optimization approach motivated by efficient usage of the CPU cache, used mostly in video game development.

The approach is to focus on the data layout, separating and sorting fields according to when they are needed, and to think about transformations of data.

In this context, what's important is how the data resides in memory.

The objective of this paradigm is to improve the performance of the system.

### C.3.2 Data-Driven Programming

Data-Driven Programming is the idea that you create domain specific languages (DSLs) which are made out of descriptive data. It is a branch of declarative programming.

In this context, what's important is to describe the behavior of a program in terms of data.

The objective of this paradigm is to increase code clarity and to reduce the risk of bugs related to mistakes in the implementation of the expected behavior of the program.

### C.3.3 Data-Oriented Programming

As we have illustrated in this book, Data-Oriented Programming is a paradigm that treats data of the system as a first-class citizen. Data is represented by generic immutable data structures (like maps and vectors) that are manipulated by general purpose functions (like map, filter, select, group, sort ...).

In this context, what's important is the representation of data by the program.

The objective of this paradigm is to reduce the complexity of the system.

## C.4 Summary

In this appendix, we have explored the ideas that inspired Data-Oriented Programming and the discoveries that made it applicable in production systems at scale in most programming languages.

# Lodash reference

Throughout the book, we have used Lodash (<https://lodash.com/>) to illustrate how to manipulate data with generic functions. But there is nothing unique about Lodash. The exact same approach could be implemented via other data manipulation libraries or custom code.

We have used Lodash FP (<https://github.com/lodash/lodash/wiki/FP-Guide>) to manipulate data without mutating it. By default, the order of the arguments in immutable functions is shuffled. The code in [D.1](#) is needed in order to make sure the signature of the immutable functions is exactly the same as the mutable functions.

## **Listing D.1 Configuring Lodash so that the immutable functions have the same signature as the mutable functions**

```
_ = fp.convert({
  "cap": false,
  "curry": false,
  "fixed": false,
  "immutable": true,
  "rearg": false
});
```

This short appendix lists the 28 Lodash functions used in the book in order to help you in case you are looking at a code snippet in the book that uses a Lodash function that you want to understand.

The functions are split in 3 categories:

1. Functions on maps in [D.1](#)
2. Functions on arrays in [D.2](#)
3. Function on collections (both arrays and maps) in [D.3](#)

Each table has three columns:

1. **Function:** The function with its signature

2. **Description:** A brief description of the function
3. **§:** The chapter number where the function appears for the first time

**Table D.1 Lodash functions on maps**

Function	Description	§
at(map, [paths])	Creates an array of values corresponding to paths of map	10
get(map, path)	Gets the value at path of map	3
has(map, path)	Checks if map has a field at path	3
merge(mapA, mapB)	Creates a map resulting from the recursive merges between mapA and mapB	3
omit(map, [paths])	Creates a map composed of the fields of map not in paths	10
set(map, path, value)	Creates a map with the same fields as map with the addition of a field <path, value>	4
values(map)	Creates an array of values of map	3

**Table D.2 Lodash functions on arrays**

Function	Description	§
concat(arrA, arrB)	Creates a new array concatenating arrA and arrB	5
flatten(arr)	Flattens arr a single level deep	14
intersection(arrA, arrB)	Creates an array of unique values both in arrA and arrB	5
nth(arr, n)	Get the element at index n in arr	10
sum(arr)	Computes the sum of the values in arr	14
union(arrA, arrB)	Creates an array of unique values from arrA and arrB	5
uniq(arr)	Creates an array of unique values from arr	14

**Table D.3 Lodash functions on collections (both arrays and maps)**

Function	Description	§
<code>every(coll, pred)</code>	Checks if <code>pred</code> returns <code>true</code> for all elements of <code>coll</code>	14
<code>filter(coll, pred)</code>	Iterates over elements of <code>coll</code> , returning an array of all elements for which <code>pred</code> returns <code>true</code>	3
<code>find(coll, pred)</code>	Iterates over elements of <code>coll</code> , returning the first element for which <code>pred</code> returns <code>true</code>	15
<code>forEach(coll, f)</code>	Iterates over elements of <code>coll</code> and invokes <code>f</code> for each element	14
<code>groupBy(coll, f)</code>	Creates a map composed of keys generated from the results of running each element of <code>coll</code> through <code>f</code> . The corresponding value for each key is an array of elements responsible for generating the key	10
<code>isEmpty(coll)</code>	Checks if <code>coll</code> is empty	5
<code>keyBy(coll, f)</code>	Creates a map composed of keys generated from the results of running each element of <code>coll</code> through <code>f</code> ; the corresponding value for each key is the last element responsible for generating the key	11
<code>map(coll, f)</code>	Creates an array of values by running each element in <code>coll</code> through <code>f</code>	3
<code>reduce(coll, f, initVal)</code>	Reduces <code>coll</code> to a value which is the accumulated result of running each element in <code>coll</code> through <code>f</code> , where each successive invocation is supplied the return value of the previous	5
<code>size(coll)</code>	Gets the size of <code>coll</code>	13
<code>sortBy(coll, f)</code>	Creates an array of elements, sorted in ascending order by the results of running each element in <code>coll</code> through <code>f</code>	14
<code>isEqual(collA, collB)</code>	Performs a deep comparison between <code>collA</code> and <code>collB</code>	6
<code>isArray(coll)</code>	Checks if <code>coll</code> is an array	5
<code>isObject(coll)</code>	Checks if <code>coll</code> is a collection	5

## Notes

The International Standard Book Number (ISBN) is a numeric commercial book identifier which is intended

1. to be unique
2. <https://docs.oracle.com/javase/9/core/creating-immutable-lists-sets-and-maps.htm>
3. A lapalissade is an obvious truth — i.e. a truism or tautology — which produces a comical effect
4. <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/reference-types#the-dynamic-type>
5. See Appendix A for details
6. <https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/march/net-framework-immutable-collections>
7. Bagwell, P. (2001). Ideal hash trees (No. REP\_WORK)
- JavaScript's strict mode is a way to opt in to a restricted variant of JavaScript that changes some silent errors
8. to throw errors
9. [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular\\_Expressions/Cheatsheet](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions/Cheatsheet)
- Lodash provides an implementation of `update`, but for the sake of teaching we are writing our own
10. implementation
- Lodash provides an implementation of `flatMap`, but for the sake of teaching we are writing our own
11. implementation
12. <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/reference-types#the-dynamic-type>
13. [https://www.researchgate.net/publication/220177801\\_Values\\_and\\_Objects\\_in\\_Programming\\_Languages](https://www.researchgate.net/publication/220177801_Values_and_Objects_in_Programming_Languages)
14. <https://www.semanticscholar.org/paper/Out-of-the-Tar-Pit-Moseley-Marks/41dc590506528e9f9d7650c235b71801483>