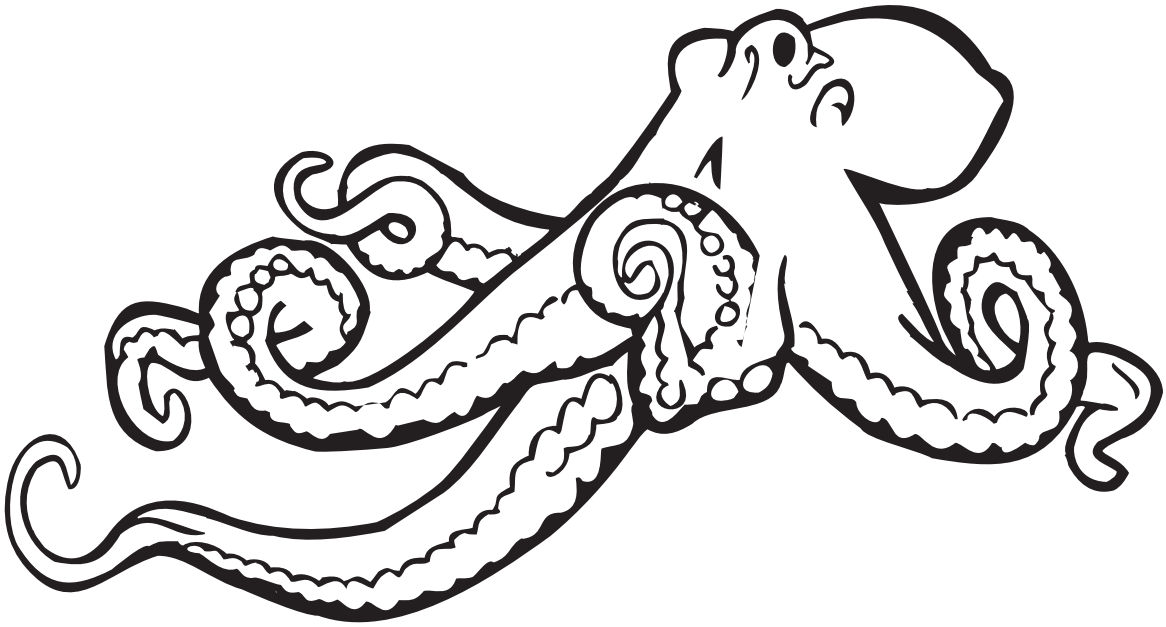


Understanding JSON Schema

Release 1.0

Michael Droettboom, et al
Space Telescope Science Institute

July 28, 2014

1 Conventions used in this book **3**
1.1 Language-specific notes . . . 3
1.2 Examples . . . 3

2 What is a schema? **5**

3 The basics **9**
3.1 Hello, World! . . . 9
3.2 The type keyword . . . 9
3.3 Declaring a JSON Schema . . . 10

4 JSON Schema Reference **11**
4.1 Type-specific keywords . . . 11
4.2 Generic keywords . . . 35
4.3 Combining schemas . . . 37
4.4 The $schema keyword . . . 41
4.5 Regular Expressions . . . 42

5 Structuring a complex schema **45**
5.1 Reuse . . . 45
5.2 Extending . . . 47

Index **49**

JSON Schema is a powerful tool for validating the structure of JSON data. However, learning to use it by reading its specification is like learning to drive a car by looking at its blueprints. You don't need to know how an internal combustion engine fits together if all you want to do is pick up the groceries. This book, therefore, aims to be the friendly driving instructor for JSON Schema. It's for those that want to write it and understand it, but maybe aren't interested in building their own car–er, writing their own JSON Schema validator–just yet.

Note: This book describes JSON Schema draft 4. Earlier versions of JSON Schema are not completely compatible with the format described here.

Where to begin?

- This book uses some novel *conventions* (page 3) for showing schema examples and relating JSON Schema to your programming language of choice.
- If you're not sure what a schema is, check out *What is a schema?* (page 5).
- *The basics* (page 9) chapter should be enough to get you started with understanding the core *JSON Schema Reference* (page 11).
- When you start developing large schemas with many nested and repeated sections, check out *Structuring a complex schema* (page 45).
- json-schema.org has a number of resources, including the official specification and tools for working with JSON Schema from various programming languages.

CHAPTER 1

CONVENTIONS USED IN THIS BOOK

1.1 Language-specific notes

The names of the basic types in JavaScript and JSON can be confusing when coming from another dynamic language. I'm a Python programmer by day, so I've notated here when the names for things are different from what they are in Python, and any other Python-specific advice for using JSON and JSON Schema. I'm by no means trying to create a Python bias to this book, but it is what I know, so I've started there. In the long run, I hope this book will be useful to programmers of all stripes, so if you're interested in translating the Python references into Algol-68 or any other language you may know, pull requests are welcome!

The language-specific sections are shown with tabs for each language. Once you choose a language, that choice will be remembered as you read on from page to page.

For example, here's a language-specific section with advice on using JSON in a few different languages:

Python

In Python, JSON can be read using the json module in the standard library.

Ruby

In Ruby, JSON can be read using the json gem.

C

For C, you may want to consider using Jansson to read and write JSON.

1.2 Examples

There are many examples throughout this book, and they all follow the same format. At the beginning of each example is a short JSON schema, illustrating a particular principle, followed by short JSON snippets that are either valid or invalid against that schema. Valid examples are in green, with a checkmark. Invalid examples are in red, with a cross. Often there are comments in between to explain why something is or isn't valid.

Note: These examples are tested automatically whenever the book is built, so hopefully they are not just helpful, but also correct!

For example, here's a snippet illustrating how to use the `number` type:

{ json schema }

```
{ "type": "number" }
```

✓

```
42
```

✓

```
-1
```

Simple floating point number:

✓

```
5.0
```

Exponential notation also works:

✓

```
2.99792458e8
```

Numbers as strings are rejected:

✗

```
"42"
```

CHAPTER 2

WHAT IS A SCHEMA?

If you've ever used XML Schema, RelaxNG or ASN.1 you probably already know what a schema is and you can happily skip along to the next section. If all that sounds like gobbledygook to you, you've come to the right place. To define what JSON Schema is, we should probably first define what JSON is.

JSON stands for "JavaScript Object Notation", a simple data interchange format. It began as a notation for the world wide web. Since JavaScript exists in most web browsers, and JSON is based on JavaScript, it's very easy to support there. However, it has proven useful enough and simple enough that it is now used in many other contexts that don't involve web surfing.

At its heart, JSON is built on the following data structures:

- object:

  ```
  { "key1": "value1", "key2": "value2" }
  ```

- array:

  ```
  [ "first", "second", "third" ]
  ```

- number:

  ```
  42
  3.1415926
  ```

- string:

  "This is a string"

- boolean:

  ```
  true
  false
  ```

- null:

  ```
  null
  ```

These types have analogs in most programming languages, though they may go by different names.

Python

The following table maps from the names of JavaScript types to their analogous types in Python:

JavaScript	Python
string	string
number	int/float
object	dict
array	list
boolean	bool
null	None

Ruby

The following table maps from the names of JavaScript types to their analogous types in Ruby:

JavaScript	Ruby
string	String
number	Integer/Float
object	Hash
array	Array
boolean	TrueClass/FalseClass
null	NilClass

With these simple data types, all kinds of structured data can be represented. With that great flexibility comes great responsibility, however, as the same concept could be represented in myriad ways. For example, you could imagine representing information about a person in JSON in different ways:

```
{
  "name": "George Washington",
  "birthday": "February 22, 1732",
  "address": "Mount Vernon, Virginia, United States"
}
```

```
{
  "first_name": "George",
  "last_name": "Washington",
  "birthday": "1732-02-22",
  "address": {
    "street_address": "3200 Mount Vernon Memorial Highway",
    "city": "Mount Vernon",
    "state": "Virginia",
    "country": "United States"
  }
}
```

Both representations are equally valid, though one is clearly more formal than the other. The design of a record will largely depend on its intended use within the application, so there's no right or wrong answer here. However, when an application says "give me a JSON record for a person", it's important to know exactly how that record should be organized. For example, we need to know what fields are expected, and how the values are represented. That's where JSON Schema comes in. The following JSON Schema fragment describes how the second example above is structured. Don't worry too much about the details for now. They are explained in subsequent chapters.

{ json schema }

```
{
  "type": "object",
  "properties": {
    "first_name": { "type": "string" },
    "last_name": { "type": "string" },
    "birthday": { "type": "string", "format": "date-time" },
    "address": {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" },
        "country": { "type" : "string" }
      }
    }
  }
}
```

By "validating" the first example against this schema, you can see that it fails:

✗

```
{
  "name": "George Washington",
  "birthday": "February 22, 1732",
  "address": "Mount Vernon, Virginia, United States"
}
```

However, the second example passes:

✓

```
{
  "first_name": "George",
  "last_name": "Washington",
  "birthday": "22-02-1732",
  "address": {
    "street_address": "3200 Mount Vernon Memorial Highway",
    "city": "Mount Vernon",
    "state": "Virginia",
    "country": "United States"
  }
}
```

You may have noticed that the JSON Schema itself is written in JSON. It is data itself, not a computer program. It's just a declarative format for "describing the structure of other data". This is both its strength and its weakness (which it shares with other similar schema languages). It is easy to concisely describe the surface structure of data, and automate validating data against it. However, since a JSON Schema can't contain arbitrary code, there are certain constraints on the relationships between data elements that can't be expressed. Any "validation tool" for a sufficiently complex data format, therefore, will likely have two phases of validation: one at the schema (or structural) level, and one at the semantic level. The latter check will likely need to be implemented using a more general-purpose programming language.

CHAPTER 3

THE BASICS

In *What is a schema?* (page 5), we described what a schema is, and hopefully justified the need for schema languages. Here, we proceed to write a simple JSON Schema.

3.1 Hello, World!

When learning any new language, it's often helpful to start with the simplest thing possible. In JSON Schema, an empty object is a completely valid schema that will accept any valid JSON.

{ json schema }

```
{ }
```

This accepts anything, as long as it's valid JSON

✓

```
42
```

✓

```
"I'm a string"
```

✓

```
{ "an": [ "arbitrarily", "nested" ], "data": "structure" }
```

3.2 The type keyword

Of course, we wouldn't be using JSON Schema if we wanted to just accept any JSON document. The most common thing to do in a JSON Schema is to restrict to a specific type. The `type` keyword is used for that.

Note: When this book refers to JSON Schema "keywords", it means the "key" part of the key/value pair in an object. Most of the work of writing a JSON Schema involves mapping a special "keyword" to a value within an object.

For example, in the following, only strings are accepted:

{ json schema }

```
{ "type": "string" }
```

✓

```
"I'm a string"
```

✗

```
42
```

The `type` keyword is described in more detail in *Type-specific keywords* (page 11).

3.3 Declaring a JSON Schema

Since JSON Schema is itself JSON, it's not always easy to tell when something is JSON Schema or just an arbitrary chunk of JSON. The `$schema` keyword is used to declare that something is JSON Schema. It's generally good practice to include it, though it is not required.

Note: For brevity, the `$schema` keyword isn't included in most of the examples in this book, but it should always be used in the real world.

{ json schema }

```
{ "$schema": "http://json-schema.org/schema#" }
```

You can also use this keyword to declare which version of the JSON Schema specification that the schema is written to. See *The $schema keyword* (page 41) for more information.

CHAPTER 4

JSON SCHEMA REFERENCE

4.1 Type-specific keywords

The `type` keyword is fundamental to JSON Schema. It specifies the data type for a schema.

At its core, JSON Schema defines the following basic types:

- *string* (page 13)
- *Numeric types* (page 15)
- *object* (page 19)
- *array* (page 29)
- *boolean* (page 34)
- *null* (page 35)

These types have analogs in most programming languages, though they may go by different names.

Python

The following table maps from the names of JavaScript types to their analogous types in Python:

JavaScript	Python
string	string
number	int/float
object	dict
array	list
boolean	bool
null	None

Ruby

The following table maps from the names of JavaScript types to their analogous types in Ruby:

JavaScript	Ruby
string	String
number	Integer/Float
object	Hash
array	Array
boolean	TrueClass/FalseClass
null	NilClass

The `type` keyword may either be a string or an array:

- If it's a string, it is the name of one of the basic types above.
- If it is an array, it must be an array of strings, where each string is the name of one of the basic types, and each element is unique. In this case, the JSON snippet is valid if it matches *any* of the given types.

Here is a simple example of using the `type` keyword:

{ json schema }

```
{ "type": "number" }
```

✓

```
42
```

✓

```
42.0
```

This is not a number, it is a string containing a number.

✗

```
"42"
```

In the following example, we accept strings and numbers, but not structured data types:

{ json schema }

```
{ "type": ["number", "string"] }
```

✓

```
42
```

✓

```
"Life, the universe, and everything"
```

✗

```
["Life", "the universe", "and everything"]
```

For each of these types, there are keywords that only apply to those types. For example, numeric types have a way of specifying a numeric range, that would not be applicable to other types. In this reference, these validation keywords are described along with each of their corresponding types in the following chapters.

4.1.1 string

The `string` type is used for strings of text. It may contain Unicode characters.

Python

In Python, "string" is analogous to the `unicode` type on Python 2.x, and the `str` type on Python 3.x.

Ruby

In Ruby, "string" is analogous to the `String` type.

{ json schema }

```
{ "type": "string" }
```

✓

```
"This is a string"
```

Unicode characters:

✓

```
"Déjà vu"
```

✓

```
"42"
```

✗

```
42
```

Length

The length of a string can be constrained using the `minLength` and `maxLength` keywords. For both keywords, the value must be a non-negative number.

{ json schema }

```
{
  "type": "string",
  "minLength": 2,
  "maxLength": 3
}
```

✗

```
"A"
```

✓

```
"AB"
```

✓

```
"ABC"
```

✗

```
"ABCD"
```

Regular Expressions

The `pattern` keyword is used to restrict a string to a particular regular expression. The regular expression syntax is the one defined in JavaScript (ECMA 262 specifically). See *Regular Expressions* (page 42) for more information.

Note: When defining the regular expressions, it's important to note that the string is considered valid if the expression matches anywhere within the string. For example, the regular expression `"p"` will match any string with a `p` in it, such as `"apple"` not just a string that is simply `"p"`. Therefore, it is usually less confusing, as a matter of course, to surround the regular expression in `^...$`, for example, `"^p$"`, unless there is a good reason not to do so.

The following example matches a simple North American telephone number with an optional area code:

{ json schema }

```
{
   "type": "string",
   "pattern": "^(\\([0-9]{3}\\))?[0-9]{3}-[0-9]{4}$"
}
```

✓

```
"555-1212"
```

✓

```
"(888)555-1212"
```

✗

```
"(888)555-1212 ext. 532"
```

✗

```
"(800)FLOWERS"
```

Format

The `format` keyword allows for basic semantic validation on certain kinds of string values that are commonly used. This allows values to be constrained beyond what the other tools in JSON Schema, including *Regular Expressions* (page 42) can do.

Note: JSON Schema implementations are not required to implement this part of the specification, and many of them do not.

There is a bias toward networking-related formats in the JSON Schema specification, most likely due to its heritage in web technologies. However, custom formats may also be used, as long as the parties exchanging the JSON documents also exchange information about the custom format types. A JSON Schema validator will ignore any format type that it does not understand.

Built-in formats

The following is the list of formats specified in the JSON Schema specification.

- `"date-time"`: Date representation, as defined by RFC 3339, section 5.6.
- `"email"`: Internet email address, see RFC 5322, section 3.4.1.
- `"hostname"`: Internet host name, see RFC 1034, section 3.1.
- `"ipv4"`: IPv4 address, according to dotted-quad ABNF syntax as defined in RFC 2673, section 3.2.
- `"ipv6"`: IPv6 address, as defined in RFC 2373, section 2.2.
- `"uri"`: A universal resource identifier (URI), according to RFC3986.

4.1.2 Numeric types

There are two numeric types in JSON Schema: *integer* (page 15) and *number* (page 17). They share the same validation keywords.

Note: JSON has no standard way to represent complex numbers, so there is no way to test for them in JSON Schema.

---

integer

The `integer` type is used for integral numbers.

Python

In Python, "integer" is analogous to the `int` type.

Ruby

In Ruby, "integer" is analogous to the `Integer` type.

{ json schema }

```
{ "type": "integer" }
```

✓

```
42
```

✓

```
-1
```

Floating point numbers are rejected:

✗

```
3.1415926
```

Numbers as strings are rejected:

✗

```
"42"
```

Warning: The precise treatment of the "integer" type may depend on the implementation of your JSON Schema validator. JavaScript (and thus also JSON) does not have distinct types for integers and floating-point values. Therefore, JSON Schema can not use type alone to distinguish between integers and non-integers. The JSON Schema specification recommends, but does not require, that validators use the mathematical value to determine whether a number is an integer, and not the type alone. Therefore, there is some disagreement between validators on this point. For example, a JavaScript-based may accept `1.0` as an integer, whereas the Python-based jsonschema does not.

Clever use of the `multipleOf` keyword (see *Multiples* (page 18)) can be used to get around this discrepancy. For example, the following likely has the same behavior on all JSON Schema implementations:

{ json schema }

```
{ "type": "number", "multipleOf": 1.0 }
```

✓

```
42
```

✓

```
42.0
```

✗

```
3.14156926
```

number

The `number` type is used for any numeric type, either integers or floating point numbers.

Python

In Python, "number" is analogous to the `float` type.

Ruby

In Ruby, "number" is analogous to the `Float` type.

{ json schema }

```
{ "type": "number" }
```

✓

```
42
```

✓

```
-1
```

Simple floating point number:

✓

```
5.0
```

Exponential notation also works:

✓

```
2.99792458e8
```

Numbers as strings are rejected:

✗

```
"42"
```

Multiples

Numbers can be restricted to a multiple of a given number, using the `multipleOf` keyword. It may be set to any positive number.

{ json schema }

```
{
    "type"       : "number",
    "multipleOf" : 10
}
```

✓

```
0
```

✓

```
10
```

✓

```
20
```

Not a multiple of 10:

✗

```
23
```

Range

Ranges of numbers are specified using a combination of the `minimum`, `maximum`, `exclusiveMinimum` and `exclusiveMaximum` keywords.

- `minimum` specifies a minimum numeric value.
- `exclusiveMinimum` is a boolean. When `true`, it indicates that the range excludes the minimum value, i.e., $x > \min$. When `false` (or not included), it indicates that the range includes the minimum value, i.e., $x \geq \min$.
- `maximum` specifies a maximum numeric value.
- `exclusiveMaximum` is a boolean. When `true`, it indicates that the range excludes the maximum value, i.e., $x < \max$. When `false` (or not included), it indicates that the range includes the maximum value, i.e., $x \leq \max$.

{ json schema }

```
{
  "type": "number",
  "minimum": 0,
  "maximum": 100,
  "exclusiveMaximum": true
}
```

Less than `minimum`:

✗

```
-1
```

`exclusiveMinimum` was not specified, so 0 is included:

✓

```
0
```

✓

```
10
```

✓

```
99
```

`exclusiveMaximum` is `true`, so 100 is not included:

✗

```
100
```

Greater than `maximum`:

✗

```
101
```

4.1.3 object

Objects are the mapping type in JSON. They map "keys" to "values". In JSON, the "keys" must always be strings. Each of these pairs is conventionally referred to as a "property".

Python

In Python, "objects" are analogous to the `dict` type. An important difference, however, is that while Python dictionaries may use anything hashable instance as a key, in JSON, all the keys must be strings.

Try not to be confused by the two uses of the word "object" here: Python uses the word `object` to mean the generic base class for everything, whereas in JSON it is used only to mean a mapping from string keys to values.

Ruby

In Ruby, "objects" are analogous to the `Hash` type. An important difference, however, is that all keys in JSON must be strings, and therefore any non-string keys are converted over to their string representation.

Try not to be confused by the two uses of the word "object" here: Ruby uses the word `Object` to mean the generic base class for everything, whereas in JSON it is used only to mean a mapping from string keys to values.

{ json schema }

```
{ "type": "object" }
```

✓

```
{
   "key"         : "value",
   "another_key" : "another_value"
}
```

✓

```
{
    "Sun"     : 1.9891e30,
    "Jupiter" : 1.8986e27,
    "Saturn"  : 5.6846e26,
    "Neptune" : 10.243e25,
    "Uranus"  : 8.6810e25,
    "Earth"   : 5.9736e24,
    "Venus"   : 4.8685e24,
    "Mars"    : 6.4185e23,
    "Mercury" : 3.3022e23,
    "Moon"    : 7.349e22,
    "Pluto"   : 1.25e22
}
```

Using non-strings as keys is invalid JSON:

✗

```
{
    0.01 : "cm"
    1    : "m",
    1000 : "km"
}
```

✗

```
"Not an object"
```

✗

```
["An", "array", "not", "an", "object"]
```

Properties

The properties (key-value pairs) on an object are defined using the `properties` keyword. The value of `properties` is an object, where each key is the name of a property and each value is a JSON schema used to validate that property.

For example, let's say we want to define a simple schema for an address made up of a number, street name and street type:

{ json schema }

```
{
  "type": "object",
  "properties": {
    "number":      { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": { "type": "string",
                     "enum": ["Street", "Avenue", "Boulevard"]
                   }
  }
}
```

✓

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue" }
```

If we provide the number in the wrong type, it is invalid:

✗

```
{ "number": "1600", "street_name": "Pennsylvania", "street_type": "Avenue" }
```

By default, leaving out properties is valid. See *Required Properties* (page 23).

✓

```
{ "number": 1600, "street_name": "Pennsylvania" }
```

By extension, even an empty object is valid:

✓

```
{ }
```

By default, providing additional properties is valid:

✓

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue",
  "direction": "NW" }
```

The `additionalProperties` keyword is used to control the handling of extra stuff, that is, properties whose names are not listed in the `properties` keyword. By default any additional properties are allowed.

The `additionalProperties` keyword may be either a boolean or an object. If `additionalProperties` is a boolean and set to `false`, no additional properties will be allowed.

Reusing the example above, but this time setting `additionalProperties` to `false`.

{ json schema }

```
{
  "type": "object",
  "properties": {
    "number":      { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": { "type": "string",
                     "enum": ["Street", "Avenue", "Boulevard"]
                   }
  },
  "additionalProperties": false
}
```

✓

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue" }
```

Since `additionalProperties` is `false`, this extra property "direction" makes the object invalid:

✗

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue",
  "direction": "NW" }
```

If `additionalProperties` is an object, that object is a schema that will be used to validate any additional properties not listed in `properties`.

For example, one can allow additional properties, but only if they are each a string:

{ json schema }

```
{
  "type": "object",
  "properties": {
    "number":      { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": { "type": "string",
                     "enum": ["Street", "Avenue", "Boulevard"]
                   }
  },
  "additionalProperties": { "type": "string" }
}
```

✓

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue" }
```

This is valid, since the additional property's value is a string:

✓

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue",
  "direction": "NW" }
```

This is invalid, since the additional property's value is not a string:

✗

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue",
  "office_number": 201  }
```

Required Properties

By default, the properties defined by the `properties` keyword are not required. However, one can provide a list of required properties using the `required` keyword.

The `required` keyword takes an array of one or more strings. Each of these strings must be unique.

In the following example schema defining a user record, we require that each user has a name and e-mail address, but we don't mind if they don't provide their address or telephone number:

{ json schema }

```
{
  "type": "object",
  "properties": {
    "name":      { "type": "string" },
    "email":     { "type": "string" },
    "address":   { "type": "string" },
    "telephone": { "type": "string" }
  },
  "required": ["name", "email"]
}
```

✓

```
{
  "name": "William Shakespeare",
  "email": "bill@stratford-upon-avon.co.uk"
}
```

Providing extra properties is fine, even properties not defined in the schema:

✓

```
{
  "name": "William Shakespeare",
  "email": "bill@stratford-upon-avon.co.uk",
  "address": "Henley Street, Stratford-upon-Avon, Warwickshire, England",
  "authorship": "in question"
}
```

Missing the required "email" property makes the JSON document invalid:

✗

```
{
  "name": "William Shakespeare",
  "address": "Henley Street, Stratford-upon-Avon, Warwickshire, England",
}
```

Size

The number of properties on an object can be restricted using the `minProperties` and `maxProperties` keywords. Each of these must be a non-negative integer.

{ json schema }

```
{
  "type": "object",
  "minProperties": 2,
  "maxProperties": 3
}
```

✗

```
{}
```

✗

```
{ "a": 0 }
```

✓

```
{ "a": 0, "b": 1 }
```

✓

```
{ "a": 0, "b": 1, "c": 2 }
```

✗

```
{ "a": 0, "b": 1, "c": 2, "d": 3 }
```

Dependencies

Note: This is an advanced feature of JSON Schema. Windy road ahead.

The `dependencies` keyword allows the schema of the object to change based on the presence of certain special properties.

There are two forms of dependencies in JSON Schema:

- **Property dependencies** declare that certain other properties must be present if a given property is present.
- **Schema dependencies** declare that the schema changes when a given property is present.

Property dependencies

Let's start with the simpler case of property dependencies. For example, suppose we have a schema representing a customer. If you have their credit card number, you also want to ensure you have a billing address. If you don't have their credit card number, a billing address would not be required. We represent this dependency of one property on another using the `dependencies` keyword. The value of the `dependencies` keyword is an object. Each entry in

the object maps from the name of a property, *p*, to an array of strings listing properties that are required whenever *p* is present.

In the following example, whenever a `credit_card` property is provided, a `billing_address` property must also be present:

{ json schema }

```
{
  "type": "object",

  "properties": {
    "name": { "type": "string" },
    "credit_card": { "type": "number" },
    "billing_address": { "type": "string" }
  },

  "required": ["name"],

  "dependencies": {
    "credit_card": ["billing_address"]
  }
}
```

✓

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555,
  "billing_address": "555 Debtor's Lane"
}
```

This instance has a `credit_card`, but it's missing a `billing_address`.

✗

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555
}
```

This is okay, since we have neither a `credit_card`, or a `billing_address`.

✓

```
{
  "name": "John Doe"
}
```

Note that dependencies are not bidirectional. It's okay to have a billing address without a credit card number.

✓

```
{
  "name": "John Doe",
  "billing_address": "555 Debtor's Lane"
}
```

To fix the last issue above (that dependencies are not bidirectional), you can, of course, define the bidirectional dependencies explicitly:

{ json schema }

```
{
  "type": "object",

  "properties": {
    "name": { "type": "string" },
    "credit_card": { "type": "number" },
    "billing_address": { "type": "string" }
  },

  "required": ["name"],

  "dependencies": {
    "credit_card": ["billing_address"],
    "billing_address": ["credit_card"]
  }
}
```

This instance has a `credit_card`, but it's missing a `billing_address`.

✗

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555
}
```

This has a `billing_address`, but is missing a `credit_card`.

✗

```
{
  "name": "John Doe",
  "billing_address": "555 Debtor's Lane"
}
```

Schema dependencies

Schema dependencies work like property dependencies, but instead of just specifying other required properties, they can extend the schema to have other constraints.

For example, here is another way to write the above:

{ json schema }

```
{
  "type": "object",

  "properties": {
    "name": { "type": "string" },
    "credit_card": { "type": "number" }
  },

  "required": ["name"],

  "dependencies": {
    "credit_card": {
      "properties": {
        "billing_address": { "type": "string" }
      },
      "required": ["billing_address"]
    }
  }
}
```

✓

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555,
  "billing_address": "555 Debtor's Lane"
}
```

This instance has a `credit_card`, but it's missing a `billing_address`:

✗

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555
}
```

This has a `billing_address`, but is missing a `credit_card`. This passes, because here `billing_address` just looks like an additional property:

✓

```
{
  "name": "John Doe",
  "billing_address": "555 Debtor's Lane"
}
```

Pattern Properties

As we saw above, `additionalProperties` can restrict the object so that it either has no additional properties that weren't explicitly listed, or it can specify a schema for any additional properties on the object. Sometimes that isn't enough, and you may want to restrict the names of the extra properties, or you may want to say that, given a particular kind of name, the value should match a particular schema. That's where `patternProperties` comes in: it

is a new keyword that maps from regular expressions to schemas. If an additional property matches a given regular expression, it must also validate against the corresponding schema.

Note: When defining the regular expressions, it's important to note that the expression may match anywhere within the property name. For example, the regular expression `"p"` will match any property name with a `p` in it, such as `"apple"`, not just a property whose name is simply `"p"`. It's therefore usually less confusing to surround the regular expression in `^...$`, for example, `"^p$"`.

In this example, any additional properties whose names start with the prefix `S_` must be strings, and any with the prefix `I_` must be integers. Any properties explicitly defined in the `properties` keyword are also accepted, and any additional properties that do not match either regular expression are forbidden.

{ json schema }

```
{
  "type": "object",
  "patternProperties": {
    "^S_": { "type": "string" },
    "^I_": { "type": "integer" }
  }
}
```

✓

```
{ "S_25": "This is a string" }
```

✓

```
{ "I_0": 42 }
```

If the name starts with `S_`, it must be a string

✗

```
{ "S_0": 42 }
```

✗

```
{ "I_42": "This is a string" }
```

This is a key that doesn't match any of the regular expressions:

✓

```
{ "keyword": "value" }
```

`patternProperties` can be used in conjunction with `additionalProperties`. In that case, `additionalProperties` will refer to any properties that are not explicitly listed in `properties` and don't match any of the `patternProperties`. In the following example, based on above, we add a `"builtin"` property, which must be a number, and declare that all additional properties (that are neither built-in or matched by `patternProperties`) must be strings:

{ json schema }

```
{
  "type": "object",
  "properties": {
    "builtin": { "type": "number" }
  },
  "patternProperties": {
    "^S_": { "type": "string" },
    "^I_": { "type": "integer" }
  },
  "additionalProperties": { "type": "string" }
}
```

✓

```
{ "builtin": 42 }
```

This is a key that doesn't match any of the regular expressions:

✓

```
{ "keyword": "value" }
```

It must be a string:

✗

```
{ "keyword": 42 }
```

4.1.4 array

Arrays are used for ordered elements. In JSON, each element in an array may be of a different type.

Python

In Python, "array" is analogous to a `list` or `tuple` type, depending on usage. However, the `json` module in the Python standard library will always use Python lists to represent JSON arrays.

Ruby

In Ruby, "array" is analogous to a `Array` type.

{ json schema }

```
{ "type": "array" }
```

✓

```
[1, 2, 3, 4, 5]
```

✓

```
[3, "different", { "types" : "of values" }]
```

✗

```
{"Not": "an array"}
```

Items

By default, the elements of the array may be anything at all. However, it's often useful to validate the items of the array against some schema as well. This is done using the `items` and `additionalItems` keywords.

There are two ways in which arrays are generally used in JSON:

- **List validation:** a sequence of arbitrary length where each item matches the same schema.
- **Tuple validation:** a sequence of fixed length where each item may have a different schema. In this usage, the index (or location) of each item is meaningful as to how the value is interpreted. (This usage is often given a whole separate type in some programming languages, such as Python's `tuple`).

List validation

List validation is useful for arrays of arbitrary length where each item matches the same schema. For this kind of array, set the `items` keyword to a single schema that will be used to validate all of the items in the array.

Note: When `items` is a single schema, the `additionalItems` keyword is meaningless, and it should not be used.

In the following example, we define that each item in an array is a number:

{ json schema }

```
{
  "type": "array",
  "items": {
    "type": "number"
  }
}
```

✓

```
[1, 2, 3, 4, 5]
```

A single "non-number" causes the whole array to be invalid:

✗

```
[1, 2, "3", 4, 5]
```

The empty array is always valid:

✓

```
[]
```

Tuple validation

Tuple validation is useful when the array is a collection of items where each has a different schema and the ordinal index of each item is meaningful.

For example, you may represent a street address such as:

```
1600 Pennsylvania Avenue NW
```

as a 4-tuple of the form:

> [number, street_name, street_type, direction]

Each of these fields will have a different schema:

- `number`: The address number. Must be a number.
- `street_name`: The name of the street. Must be a string.
- `street_type`: The type of street. Should be a string from a fixed set of values.
- `direction`: The city quadrant of the address. Should be a string from a different set of values.

To do this, we set the `items` keyword to an array, where each item is a schema that corresponds to each index of the document's array. That is, an array where the first element validates the first element of the input array, the second element validates the second element of the input array, etc.

Here's the example schema:

{ json schema }

```
{
  "type": "array",
  "items": [
    {
      "type": "number"
    },
    {
      "type": "string"
    },
    {
      "type": "string",
      "enum": ["Street", "Avenue", "Boulevard"]
    },
    {
      "type": "string",
      "enum": ["NW", "NE", "SW", "SE"]
    }
  ]
}
```

✓

```
[1600, "Pennsylvania", "Avenue", "NW"]
```

"Drive" is not one of the acceptable street types:

✗

```
[24, "Sussex", "Drive"]
```

This address is missing a street number

✗

```
["Palais de l'Élysée"]
```

It's okay to not provide all of the items:

✓

```
[10, "Downing", "Street"]
```

And, by default, it's also okay add additional items to end:

✓

```
[1600, "Pennsylvania", "Avenue", "NW", "Washington"]
```

The `additionalItems` keyword controls whether it's valid to have additional items in the array beyond what is defined in the schema. Here, we'll reuse the example schema above, but set `additionalItems` to `false`, which has the effect of disallowing extra items in the array.

{ json schema }

```
{
  "type": "array",
  "items": [
    {
      "type": "number"
    },
    {
      "type": "string"
    },
    {
      "type": "string",
      "enum": ["Street", "Avenue", "Boulevard"]
    },
    {
      "type": "string",
      "enum": ["NW", "NE", "SW", "SE"]
    }
  ],
  "additionalItems": false
}
```

✓

```
[1600, "Pennsylvania", "Avenue", "NW"]
```

It's ok to not provide all of the items:

✓

```
[1600, "Pennsylvania", "Avenue"]
```

But, since `additionalItems` is `false`, we can't provide extra items:

✗

```
[1600, "Pennsylvania", "Avenue", "NW", "Washington"]
```

Length

The length of the array can be specified using the `minItems` and `maxItems` keywords. The value of each keyword must be a non-negative number. These keywords work whether doing *List validation* (page 30) or *Tuple validation* (page 31).

{ json schema }

```
{
  "type": "array",
  "minItems": 2,
  "maxItems": 3
}
```

✗

```
[]
```

✗

```
[1]
```

✓

```
[1, 2]
```

✓

```
[1, 2, 3]
```

✗

```
[1, 2, 3, 4]
```

Uniqueness

A schema can ensure that each of the items in an array are unique. Simply set the `uniqueItems` keyword to `true`.

{ json schema }

```
{
  "type": "array",
  "uniqueItems": true
}
```

✓

```
[1, 2, 3, 4, 5]
```

✗

```
[1, 2, 3, 3, 4]
```

The empty array always passes:

✓

```
[]
```

4.1.5 boolean

The boolean type matches only two special values: `true` and `false`. Note that values that *evaluate* to `true` or `false`, such as 1 and 0, are not accepted by the schema.

Python

In Python, "boolean" is analogous to `bool`. Note that in JSON, `true` and `false` are lower case, whereas in Python they are capitalized (`True` and `False`).

Ruby

In Ruby, "boolean" is analogous to `TrueClass` and `FalseClass`. Note that in Ruby there is no `Boolean` class.

{ json schema }

```
{ "type": "boolean" }
```

✓

```
true
```

✓

```
false
```

✗

```
"true"
```

Values that evaluate to `true` or `false` are still not accepted by the schema:

✗

```
0
```

4.1.6 null

The null type is generally used to represent a missing value. When a schema specifies a `type` of `null`, it has only one acceptable value: `null`.

Python

In Python, `null` is analogous to `None`.

Ruby

In Ruby, `null` is analogous to `nil`.

{ json schema }

```
{ "type": "null" }
```

✓

```
null
```

✗

```
false
```

✗

```
0
```

✗

```
""
```

4.2 Generic keywords

This chapter lists some miscellaneous properties that are available for all JSON types.

4.2.1 Metadata

JSON Schema includes a few keywords, `title`, `description` and `default`, that aren't strictly used for validation, but are used to describe parts of a schema.

The `title` and `description` keywords must be strings. A "title" will preferably be short, whereas a "description" will provide a more lengthy explanation about the purpose of the data described by the schema. Neither are required, but they are encouraged for good practice.

The `default` keyword specifies a default value for an item. JSON processing tools may use this information to provide a default value for a missing key/value pair, though many JSON schema validators simply ignore the `default` keyword. It should validate against the schema in which it resides, but that isn't required.

{ json schema }

```
{
  "title" : "Match anything",
  "description" : "This is a schema that matches anything.",
  "default" : "Default value"
}
```

4.2.2 Enumerated values

The `enum` keyword is used to restrict a value to a fixed set of values. It must be an array with at least one element, where each element is unique.

The following is an example for validating street light colors:

{ json schema }

```
{
  "type": "string",
  "enum": ["red", "amber", "green"]
}
```

✓

```
"red"
```

✗

```
"blue"
```

You can use `enum` even without a type, to accept values of different types. Let's extend the example to use `null` to indicate "off", and also add 42, just for fun.

{ json schema }

```
{
  "enum": ["red", "amber", "green", null, 42]
}
```

✓

```
"red"
```

✓

```
null
```

✓

```
42
```

✗

```
0
```

However, in most cases, the elements in the `enum` array should also be valid against the enclosing schema:

{ json schema }

```
{
  "type": "string",
  "enum": ["red", "amber", "green", null]
}
```

✓

```
"red"
```

This is in the `enum`, but it's invalid against `{ "type": "string" }`, so it's ultimately invalid:

✗

```
null
```

4.3 Combining schemas

JSON Schema includes a few keywords for combining schemas together. Note that this doesn't necessarily mean combining schemas from multiple files or JSON trees, though these facilities help to enable that and are described in *Structuring a complex schema* (page 45). Combining schemas may be as simple as allowing a value to be validated against multiple criteria at the same time.

For example, in the following schema, the `anyOf` keyword is used to say that the given value may be valid against any of the given subschemas. The first subschema requires a string with maximum length 5. The second subschema requires a number with a minimum value of 0. As long as a value validates against *either* of these schemas, it is considered valid against the entire combined schema.

{ json schema }

```
{
  "anyOf": [
    { "type": "string", "maxLength": 5 },
    { "type": "number", "minimum": 0 }
  ]
}
```

✓

```
"short"
```

✗

```
"too long"
```

✓

```
12
```

✗

```
-5
```

The keywords used to combine schemas are:

- *allOf* (page 38): Must be valid against *all* of the subschemas
- *anyOf* (page 39): Must be valid against *any* of the subschemas
- *oneOf* (page 39): Must be valid against *exactly one* of the subschemas

All of these keywords must be set to an array, where each item is a schema.

In addition, there is:

- *not* (page 40): Must *not* be valid against the given schema

4.3.1 allOf

To validate against `allOf`, the given data must be valid against all of the given subschemas.

{ json schema }

```
{
  "allOf": [
    { "type": "string" },
    { "maxLength": 5 }
  ]
}
```

✓

```
"short"
```

✗

```
"too long"
```

Note that it's quite easy to create schemas that are logical impossibilities with `allOf`. The following example creates a schema that won't validate against anything (since something may not be both a string and a number at the same time):

{ json schema }

```
{
  "allOf": [
    { "type": "string" },
    { "type": "number" }
  ]
}
```

✗

```
"No way"
```

✗

```
-1
```

4.3.2 anyOf

To validate against `anyOf`, the given data must be valid against any (one or more) of the given subschemas.

{ json schema }

```
{
  "anyOf": [
    { "type": "string" },
    { "type": "number" }
  ]
}
```

✓

```
"Yes"
```

✓

```
42
```

✗

```
{ "Not a": "string or number" }
```

4.3.3 oneOf

To validate against `oneOf`, the given data must be valid against exactly one of the given subschemas.

{ json schema }

```
{
  "oneOf": [
    { "type": "number", "multipleOf": 5 },
    { "type": "number", "multipleOf": 3 }
  ]
}
```

✓

```
10
```

✓

```
9
```

Not a multiple of either 5 or 3.

✗

```
2
```

Multiple of *both* 5 and 3 is rejected.

✗

```
15
```

Note that it's possible to "factor" out the common parts of the subschemas. The following schema is equivalent to the one above:

{ json schema }

```
{
  "type": "number",
  "oneOf": [
    { "multipleOf": 5 },
    { "multipleOf": 3 }
  ]
}
```

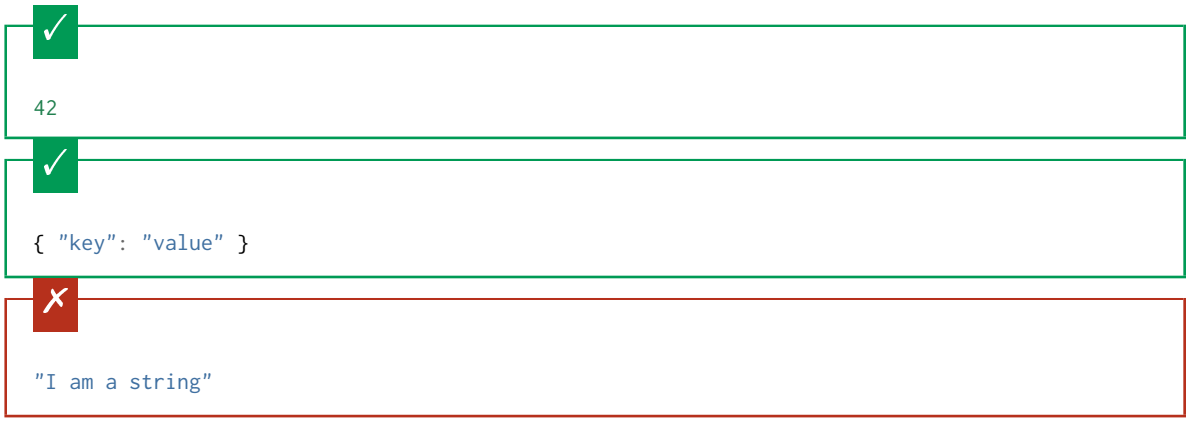
4.3.4 not

This doesn't strictly combine schemas, but it belongs in this chapter along with other things that help to modify the effect of schemas in some way. The `not` keyword declares that a instance validates if it doesn't validate against the given subschema.

For example, the following schema validates against anything that is not a string:

{ json schema }

```
{ "not": { "type": "string" } }
```



4.4 The $schema keyword

The `$schema` keyword is used to declare that a JSON fragment is actually a piece of JSON Schema. It also declares which version of the JSON Schema standard that the schema was written against.

It is recommended that all JSON Schemas have a `$schema` entry, which must be at the root. Therefore most of the time, you'll want this at the root of your schema:

```
"$schema": "http://json-schema.org/schema#"
```

4.4.1 Advanced

If you need to declare that your schema was written against a specific version of the JSON Schema standard, and not just the latest version, you can use one of these predefined values:

- `http://json-schema.org/schema#`

  JSON Schema written against the current version of the specification.

- `http://json-schema.org/hyper-schema#`

  JSON Schema hyperschema written against the current version of the specification.

- `http://json-schema.org/draft-04/schema#`

  JSON Schema written against this version.

- `http://json-schema.org/draft-04/hyper-schema#`

  JSON Schema hyperschema written against this version.

- `http://json-schema.org/draft-03/schema#`

  JSON Schema written against JSON Schema, draft v3

- `http://json-schema.org/draft-03/hyper-schema#`

  JSON Schema hyperschema written against JSON Schema, draft v3

Additionally, if you have extended the JSON Schema language to include your own custom keywords for validating values, you can use a custom URI for `$schema`. It must not be one of the predefined values above.

4.5 Regular Expressions

The *pattern* (page 14) and *Pattern Properties* (page 27) keywords use regular expressions to express constraints. The regular expression syntax used is from JavaScript (ECMA 262, specifically). However, that complete syntax is not widely supported, therefore it is recommended that you stick to the subset of that syntax described below.

- A single unicode character (other than the special characters below) matches itself.
- `^`: Matches only at the beginning of the string.
- `$`: Matches only at the end of the string.
- `(...)`: Group a series of regular expressions into a single regular expression.
- `|`: Matches either the regular expression preceding or following the `|` symbol.
- `[abc]`: Matches any of the characters inside the square brackets.
- `[a-z]`: Matches the range of characters.
- `[^abc]`: Matches any character *not* listed.
- `[^a-z]`: Matches any character outside of the range.
- `+`: Matches one or more repetitions of the preceding regular expression.
- `*`: Matches zero or more repetitions of the preceding regular expression.
- `?`: Matches zero or one repetitions of the preceding regular expression.
- `+?`, `*?`, `??`: The `*`, `+`, and `?` qualifiers are all greedy; they match as much text as possible. Sometimes this behavior isn't desired and you want to match as few characters as possible.
- `{x}`: Match exactly `x` occurrences of the preceding regular expression.
- `{x,y}`: Match at least `x` and at most `y` occurrences of the preceding regular expression.
- `{x,}`: Match `x` occurrences or more of the preceding regular expression.
- `{x}?`, `{x,y}?`, `{x,}?`: Lazy versions of the above expressions.

Python

This subset of JavaScript regular expressions is compatible with Python regular expressions. Pay close attention to what is missing, however. Notably, it is not recommended to use `.` to match any character.

4.5.1 Example

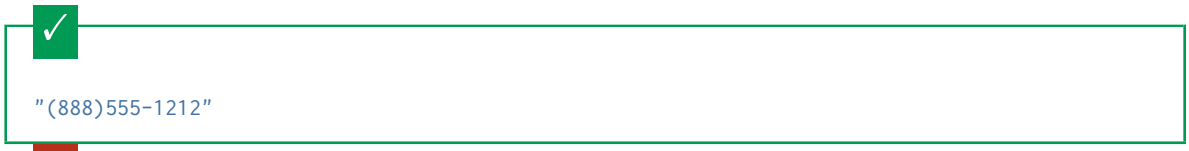
The following example matches a simple North American telephone number with an optional area code:

{ json schema }

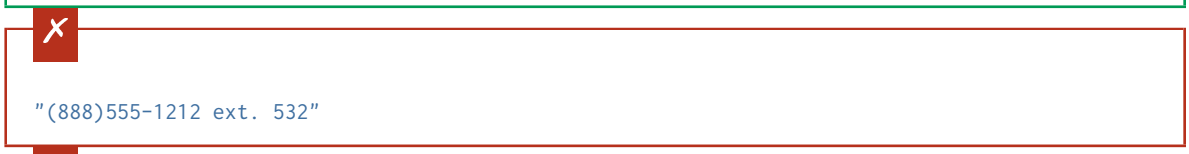
```
{
    "type": "string",
    "pattern": "^(\\([0-9]{3}\\))?[0-9]{3}-[0-9]{4}$"
}
```

✓

```
"555-1212"
```



```
"(888)555-1212"
```



```
"(888)555-1212 ext. 532"
```

```
"(800)FLOWERS"
```

CHAPTER 5

STRUCTURING A COMPLEX SCHEMA

When writing computer programs of even moderate complexity, it's commonly accepted that "structuring" the program into reusable functions is better than copying-and-pasting duplicate bits of code everywhere they are used. Likewise in JSON Schema, for anything but the most trivial schema, it's really useful to structure the schema into parts that can be reused in a number of places. This chapter will present some practical examples that use the tools available for reusing and structuring schemas.

5.1 Reuse

For this example, let's say we want to define a customer record, where each customer may have both a shipping and a billing address. Addresses are always the same–they have a street address, city and state–so we don't want to duplicate that part of the schema everywhere we want to store an address. Not only does it make the schema more verbose, but it makes updating it in the future more difficult. If our imaginary company were to start international business in the future and we wanted to add a country field to all the addresses, it would be better to do this in a single place rather than everywhere that addresses are used.

So let's start with the schema that defines an address:

```
{
  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "city":           { "type": "string" },
    "state":          { "type": "string" }
  },
  "required": ["street_address", "city", "state"]
}
```

Since we are going to reuse this schema, it is customary (but not required) to put it in the parent schema under a key called `definitions`:

```
{
  "definitions": {
    "address": {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city":           { "type": "string" },
        "state":          { "type": "string" }
      },
      "required": ["street_address", "city", "state"]
    }
  }
}
```

We can then refer to this schema snippet from elsewhere using the `$ref` keyword. The easiest way to describe `$ref` is that it gets logically replaced with the thing that it points to. So, to refer to the above, we would include:

```
{ "$ref": "#/definitions/address" }
```

The value of `$ref` is a string in a format called JSON Pointer.

Note: JSON Pointer aims to serve the same purpose as XPath from the XML world, but it is much simpler.

The pound symbol (#) refers to the current document, and then the slash (/) separated keys thereafter just traverse the keys in the objects in the document. Therefore, in our example `"#/definitions/address"` means:

1. go to the root of the document
2. find the value of the key `"definitions"`
3. within that object, find the value of the key `"address"`

`$ref` can also be a relative or absolute URI, so if you prefer to include your definitions in separate files, you can also do that. For example:

```
{ "$ref": "definitions.json#/address" }
```

would load the address schema from another file residing alongside this one.

Now let's put this together and use our address schema to create a schema for a customer:

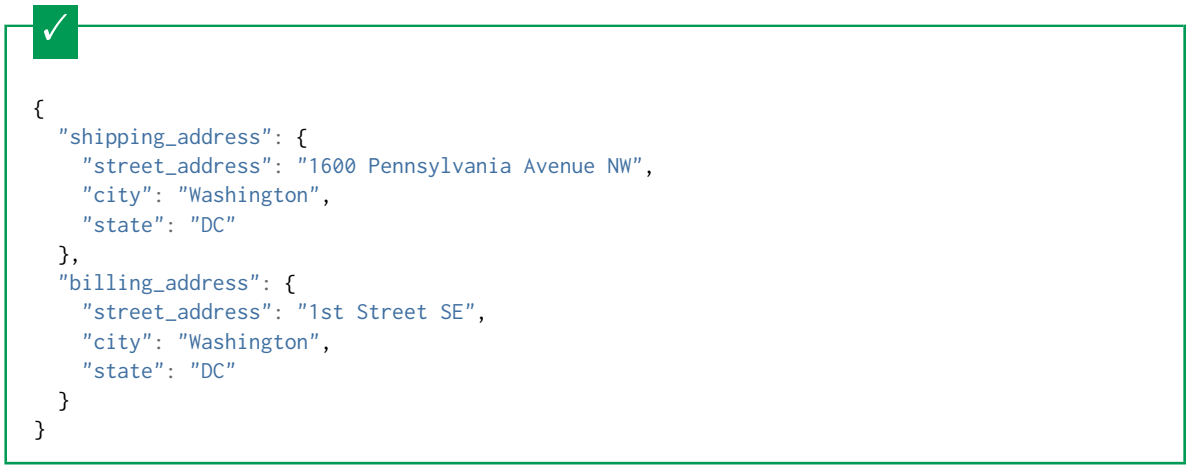
{ json schema }

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",

  "definitions": {
    "address": {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city":           { "type": "string" },
        "state":          { "type": "string" }
      },
      "required": ["street_address", "city", "state"]
    }
  },

  "type": "object",

  "properties": {
    "billing_address": { "$ref": "#/definitions/address" },
    "shipping_address": { "$ref": "#/definitions/address" }
  }
}
```



```
{
  "shipping_address": {
    "street_address": "1600 Pennsylvania Avenue NW",
    "city": "Washington",
    "state": "DC"
  },
  "billing_address": {
    "street_address": "1st Street SE",
    "city": "Washington",
    "state": "DC"
  }
}
```

5.2 Extending

The power of `$ref` really shines when it is combined with the combining keywords `allOf`, `anyOf` and `oneOf` (see *Combining schemas* (page 37)).

Let's say that for shipping address, we want to know whether the address is a residential or business address, because the shipping method used may depend on that. For the billing address, we don't want to store that information, because it's not applicable.

To handle this, we'll update our definition of shipping address:

```
"shipping_address": { "$ref": "#/definitions/address" }
```

to instead use an `allOf` keyword entry combining both the core address schema definition and an extra schema snippet for the address type:

```
"shipping_address": {
  "allOf": [
    // Here, we include our "core" address schema...
    { "$ref": "#/definitions/address" },

    // ...and then extend it with stuff specific to a shipping
    // address
    { "properties": {
        "type": { "enum": [ "residential", "business" ] }
      },
      "required": ["type"]
    }
  ]
}
```

Tying this all together,

{ json schema }

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",

  "definitions": {
    "address": {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city":           { "type": "string" },
        "state":          { "type": "string" }
      },
      "required": ["street_address", "city", "state"]
    }
  },

  "type": "object",

  "properties": {
    "billing_address": { "$ref": "#/definitions/address" },
    "shipping_address": {
      "allOf": [
        { "$ref": "#/definitions/address" },
        { "properties":
          { "type": { "enum": [ "residential", "business" ] } },
          "required": ["type"]
        }
      ]
    }
  }
}
```

This fails, because it's missing an address type:

✗

```
{
  "shipping_address": {
    "street_address": "1600 Pennsylvania Avenue NW",
    "city": "Washington",
    "state": "DC"
  }
}
```

✓

```
{
  "shipping_address": {
    "street_address": "1600 Pennsylvania Avenue NW",
    "city": "Washington",
    "state": "DC",
    "type": "business"
  }
}
```

From these basic pieces, it's possible to build very powerful constructions without a lot of duplication.

Symbols

$ref, 46
$schema, 41

A

additionalItems, 30
additionalProperties, 20
allOf, 38
anyOf, 39
array, 29
    items, 30
    length, 33
    list validation, 30
    tuple validation, 31
    uniqueness, 33

B

boolean, 34

C

combining schemas, 37
    allOf, 38
    anyOf, 39
    not, 40
    oneOf, 39

D

dependencies, 24
description, 35

E

enum, 36
enumerated values, 36
exclusiveMaximum, 18
exclusiveMinimum, 18

F

format, 15

I

integer, 15
items, 30

M

maximum, 18
maxItems, 33
maxLength, 13
maxProperties, 23
metadata, 35
minimum, 18
minItems, 33
minLength, 13
minProperties, 23
multipleOf, 17

N

not, 40
null, 35
number, 15, 17
    multiple of, 17
    range, 18

O

object, 19
    dependencies, 24
    properties, 20
    regular expression, 27
    required properties, 23
    size, 23
oneOf, 39

P

pattern, 14
patternProperties, 27
properties, 20

R

regular expressions, 41

required, 23

S

schema
    keyword, 41
string, 13
    format, 15
    length, 13
    regular expression, 14
structure, 43

T

title, 35
type, 11
types
    basic, 11
    numeric, 15

U

uniqueItems, 33