Chapter No. 1
"An Overview of Domain-specific Languages with Clojure"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.1 "An Overview of Domain-specific Languages with Clojure"

A synopsis of the book’s content

Information on where to buy this book

About the Author

**Ryan D. Kelker** is a Clojure enthusiast and works as a freelance—he is willing to take on any project that sounds interesting. He started exploring computers and the Internet at a very early age and he eventually ended up building both machines and software. Starting with MS DOS, batch files, and QBasic, he eventually floated towards Arch Linux and the Clojure language.

He has four certifications from both CompTIA and Cisco, and has decided not to pursue any additional certifications. These days, he spends most of his time reading about software development, cyber security, and news surrounding up-and-coming computer languages. While away from the computer, he is usually reading a book or going out to eat with the people he loves the most.

I would like to thank Packt publishing for giving me the opportunity to write something great, and Chatsubolabs.com for giving me my first Clojure job. I would also like to thank Tom Marble, Edward Raison, and Kevin Raison for teaching me countless lessons.

For More Information:

Clojure for Domain-specific Languages

Clojure for Domain-specific Languages is an example-oriented guide to building custom languages. Many of the core components of Clojure are covered to help you better understand your options when making a domain-specific language. By the end of this book, you should be able to make an internal DSL.

What This Book Covers

Chapter 1, An Overview of Domain-specific Languages with Clojure, will help you learn specifically what a domain-specific language (DSL) is and why you may use one. This will include a comparison of many existing DSLs, both in Clojure and other languages.

Chapter 2, Design Concepts with Clojure, will go over some basic concepts that apply to software development in any programming language. Each section will explain what the concept is and why the concept should be applied to your projects. As with all sources of information, choose what works for you.

Chapter 3, Clojure Editing and Project Creation, will help you get started with the Emacs text editor and the Lein project utility. Because entire books can be written on either of the software discussed in this chapter, each section will only go over the basics to help you get started.

Chapter 4, Features, Functions, and Macros, briefly goes over some of Clojure's key components. More specifically, Clojure namespaces, Java classes, immutability, metadata, lazy sequences, collection destructuring, functions, relationships, and macros.

Chapter 5, Collections and Sequencing, specifically focuses on Clojure's collection data structures and ways to construct, use, and manipulate them. Each section will focus on a certain data structure, and the end of the chapter will focus more on operations that can be used on sequences.

Chapter 6, Assignment and Concurrency, starts off by getting more hands-on with variables and how to manipulate them. The variable section explains the most common variable definition methods and functions for handling Clojure variables. It's also okay to skip sections if you feel that you already know the material well enough.

Chapter 7, Flow Control, Error Handling, and Math, starts off by explaining and displaying examples of common flow control methods, then it moves on to object comparison and casting. We will also be covering error handling and, finally, we will move on to the Arithmetic section that contains a lot of surprises for those who aren't familiar with Clojure. After reading all of the sections, remember to try some of the examples in a REPL session while reviewing the chapter.

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
Chapter 8, *Methods for Abstraction*, starts off with an explanation of the classes we need for the multimethod polymorphism tutorial. By the end of this chapter, you should be familiar with making classes, polymorphic functions, and your own data types.

Chapter 9, *An Example Twitter DSL*, is focused on building a Twitter DSL but it starts off by covering some of the key concepts of building layers of Clojure code on top of Java libraries. Some concepts you may already be familiar with, but you're encouraged to read the initial part of this chapter to better understand the concepts presented in this chapter.

Chapter 10, *Unit Testing*, will briefly cover the clojure.test, expectations, midje, and speclj unit testing frameworks. Each framework will be used to make tests for the Twitter mini-DSL created in the last chapter.

Chapter 11, *Clojure DSLs inside Java*, will help you learn about generating Java classes and making them available in your Java project. The first half of this chapter will cover how to build a Clojure class that can be called from both Java and Clojure. The second part will cover the importing and use of Clojure source files from within Java.

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
In this chapter, you'll learn specifically what a domain-specific language (DSL) is and why you should use one. This will include the comparison of many existing DSLs both in Clojure and other languages.

This chapter will cover examples for the following domains:

- Clojure libraries
- Database domain
- HTML domain
- ECMA/JavaScript domain
- Audio domain
- Image domain

This chapter will also go over a few internal and external DSLs, but this book is aimed at the development of an internal DSL. That doesn't mean the information presented in this chapter or book can't be applied to the development of an external DSL. After learning the pros and cons of external and internal DSLs, examples of both will be presented and explained.

After learning the difference between a Clojure library and a Clojure DSL, this chapter will cover several problem domains. Each domain has several examples surrounding different DSLs and compares the solutions of each example. This includes comparing non-Clojure solutions to Clojure-based solutions.

By the end of this chapter, you should have an understanding as to when and where to use a DSL.

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
Domain-specific languages (DSL)

A domain-specific language (DSL) is the opposite of a general-purpose language in the sense that it's designed from beginning to end to solve a very specific set of problems. A DSL can only help solve a single set of problems and is limited by design. The limits of the language are not bound to the expressiveness of the language but to the scope of the problems it can handle. Now we will look at this in more detail.

Limited scope

The limited scope of problems that a DSL can handle is one of the key differences between a general-purpose language and a domain-specific language. It's important for the DSL to remain within its problem domain. For example, a language specifically designed for image manipulation shouldn't contain the expressive ability to manipulate audio, and vice versa.

Syntax

The way you would think about solving a problem in a specific problem domain is the same way you would express the solution in the DSL. Some DSLs can fall short in some way, but a solid DSL usually covers most, if not all, aspects of a problem domain and is developed gradually from the ground up. In general, the concept is to make the complex simpler in terms of literal expression.

Making the complex simpler doesn't necessarily mean to simplify in terms of a DSL. The reason to use a DSL is to separate the user from the complexity of the problem domain. Using a DSL can simplify the solution-building process, but the DSL doesn't actually change the problem in the problem domain.

All well-written DSLs should be recognizable by a person familiar with the problem domain. If you were to write a language specifically designed to handle the process of ordering food, assuming the user knows how to make an order, a nontechnical user should be able to understand the syntax of the language without any knowledge of the language's complexities. Nevertheless, depending on the DSL's requirements and the intended user audience, some knowledge of the parent language may be required for the end user to fully understand the expressions of the DSL.

Syntax might be the first thing you see in any language, but that's not the real difference between a good language and a not-so-good language. A good DSL makes expressing a specific solution easier in comparison to expressing the same solution in a general-purpose language.
Using a DSL

For instance, if we want to build a website, we'll probably need a web language for a much simpler development process. If we want to express something in the web browser, we'll probably write a web page in HTML. If we want to add or modify the styles of our web page, we'll use CSS. Then, if we wish to add event-related functionality, we would use ECMAScript (also known as JavaScript) to do so.

Popular DSLs

<table>
<thead>
<tr>
<th>DSL</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured Query Language (SQL)</td>
<td>Databases</td>
</tr>
<tr>
<td>HyperText Markup Language (HTML)</td>
<td>Websites</td>
</tr>
<tr>
<td>Postscript</td>
<td>Publishing</td>
</tr>
</tbody>
</table>

Websites are some of the best examples of DSLs working together in concert. The dynamic results of a page are often rendered by languages and frameworks such as PHP, Node.js, Ruby on Rails, or Django. The content of the page is often retrieved from a database with SQL, SQL object-relational mapping (ORM), or a solution-specific NoSQL DSL. When the dynamic page is requested, the language or framework decides what to display and what not to display. If the page requires content from a database, the language or framework will use a database DSL to get the requested content. Then, the retrieved content is formatted with the DSL HTML. The HTML itself may contain other DSLs such as ECMAScript and/or CSS. Many web languages often have their own templating system that can also be its very own DSL.

A contract between language and domain

We can also think of DSL as an agreement or a protocol. The user agrees to be ignorant of its inner workings and not of the problem, and the DSL agrees to ignore everything not related to the problem. This agreement allows for the DSL to act as a loyal intermediary between the solution and the problem.

A better way to understand this protocol or agreement is to think about your everyday activities. For example, if you were to pick up the phone and order a pizza, you would be speaking a DSL. The problem is that you need a pizza, and the solution can only be expressed in a contextual manner for that very specific problem. This is the agreement of that DSL. Now, if you were to go into a video game store and try to order a pizza, the people there may understand the language, but you wouldn't get a pizza, because that domain is outside the agreement of that type of language.

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
The language of trust
The user must be able to trust that the DSL is doing what it says it's doing in the expressions. The actions of the DSL, either hidden or verbose, must be in direct relation to what's explicitly being expressed to avoid unwanted side effects or mutations. For example, you wouldn't want to use a DSL if a statement such as `roll the ball` actually did more than just rolling the ball.

Internal versus External DSLs
Although many DSLs strive to do the same thing, not all DSLs are built the same. Many try to hide as much of the host language as they possibly can get away with, and some are completely outside the host language. Then there's the in-between, where the host language is used for leverage and becomes a power tool for the user instead of a burden.

The use of a DSL is restricted to the way it's implemented. A DSL that requires itself to be embedded within the host language can be called an Internal DSL. A DSL with its own syntax and that isn't required to be embedded within another language can be called an External DSL.

Regardless of the type of DSL we may end up building, the concept remains the same. We need to separate the user from the complexities of the problem without separating the user from the problem itself. We'll look at the major differences between an Internal and an External DSL, although this book will mostly focus on Internal DSL development with Clojure. That doesn't mean that the information expressed in this book can't be applied to External DSL development or to the development of other languages as well.

External DSLs
An External DSL is like making a completely new language from beginning to end. The language itself is separate from your development language and requires many programming concepts found in the development of a general-purpose high-level language. Some of the main programming concepts of an External DSL encompass other major concepts such as code generation, compilation, symbol parsing, and language interpretation.

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
There are many reasons to build and use an External DSL. You might need a language with a simpler or more expressive syntax than what the host language is able to provide. You might also need a language that is completely separated from the host language.

*Martin Fowler, in a 2005 blog article about language tools, once stated that some XML configuration files can be thought of as an External DSL that uses XML to help simplify the parsing process.*

His point rings true in the sense that an XML configuration file uses a special grammar that's independent from the host language parsing the configuration file.

One of the many positive things about an External DSL is that you can choose the environment in which the code can be executed. Another advantage is that you essentially have an unlimited amount of expressiveness that can be built into the language because you're not restricted by the host language used to build the parser. This type of freedom allows for a custom syntax that the host language wouldn't be able to provide.

Although the custom syntax or grammar of your External DSL can make life easier in some ways, there are a few pitfalls that must be taken into consideration if you wish to develop an External DSL. Developing a new language that is separate from the host language means that the features of your language are restricted to the capabilities of your parser. Another major concern might be the lack of utilities that support the use of your language. For example, existing *Integrated Development Environments (IDEs)* won't be able to give you many common features such as syntax highlighting and support for enhanced error exception handling. You'll also have to figure out a practical way to handle, warn about, and report errors whenever they might occur.

Clojure has many language-building friendly features that can transform the structure of the Clojure language itself to fit most of your expressive needs. This brings us back to the concept of an Internal DSL and how we can use Clojure as just another building block.

**Internal DSLs**

An Internal DSL is essentially a language that is either built on top of an existing language or extends another language, and is most commonly packaged as a language library. As with an External DSL, an Internal DSL is written in a very specific grammar, but unlike the External DSL, the grammar is restricted to the legal syntax of the host language. This means that your DSL will instantly adopt all the nice and not-so-nice syntactical features.

For More Information:

www.packtpub.com/clojure-for-domain-specific-languages/book
Adopting the host language's syntactical features means that you'll have to design your language in a way that also makes sense in the host language. This also means that all of the features and problems of the host language will have great influence on how your DSL will be implemented. An Internal DSL instantly gives us a great advantage over an External DSL in the sense that we have tools that can already understand our language, because we are using a language that has to follow the same syntax rules of the host language. Another great advantage is that our language can easily be integrated with existing projects of the host language. Error handling and reporting are also simplified in many ways, because the Internal DSL can just hitch a ride on the host language's error-handling methods. We can also see how an Internal DSL might greatly decrease the learning curve for a group of developers who already know the host language of the DSL.

Using an Internal DSL is a nicer way of interacting with a specific problem domain from within a general-purpose language. This allows for the host language to act as an intermediary between other Internal DSLs that might be required for a completely different set of problems within the project. For example, you can query results from a SQL database with a database DSL and use the host language to format the query results, so that another DSL can handle the query results properly.

If it's not already obvious, you generally wouldn't use an Internal DSL when the problem you're trying to solve can be reasonably expressed without much effort in your general-purpose language. For example, fetching data from a URL probably wouldn't require a DSL, but being able to safely handle and manipulate a PDF file type may very well require a DSL. Every general-purpose language has its strong and weak points, like all languages, so where you might need an Internal DSL will mostly depend on how well the host language can generically handle the problem.

Some Internal DSLs are actually built on top of an existing set of Internal DSLs and libraries to bring forth the overall functionality of the language. Usually, language libraries are often bundles of well-established libraries and DSLs working together in concert to handle the actions of the parent library's methods. Not every DSL or library will depend on another DSL or library, but development time can be greatly decreased by using existing code instead of starting from nothing.

**Clojure libraries**

Many libraries in Clojure could be classified as an Internal DSL. Clojure's LISP-based syntax, and the unique ability to define variables and methods with nonalphabetical characters, makes many Clojure libraries feel like their own language. The difference between a Clojure library and an Internal DSL isn't all that much.

For More Information:

www.packtpub.com/clojure-for-domain-specific-languages/book
The characteristics of a Clojure library

As with most programming languages, Clojure has multiple library solutions for certain classes of problem domains. Some libraries may be a Clojure wrapping of an existing Java library or domain-specific language. It's perfectly acceptable to build a Clojure library or DSL based completely on an existing Java library, but there are some drawbacks that must be kept in mind when using a library of this type. Depending on how the library returns objects, generally speaking, the objects have the characteristics of a Java object and not of a Clojure object.

Pure Clojure library objects are immutable by default, as opposed to Java's default objects being mutable. This means that Java objects welcome mutation or changes without instantiating a new object reflecting the side effects of the applied mutations. This can cause a problem if you're unaware that you're dealing with a mutable object, and you try to treat it as a normal Clojure object. Generally speaking, you'll know when you're dealing with a pure Java object because you'll probably have to use Java interoperation notation to make method calls specifically related to the object type.

Clojure libraries, in comparison to other languages, are significantly smaller in terms of line count. There are many reasons for this, but some of the key reasons are that Clojure's hyper-powerful functions, macros, and sequence handling features can greatly reduce tasks that usually span over several lines in most other languages.

The current state of Clojure libraries

Clojure is still a young language, so Clojure's best development practices are still evolving. There's currently a project that aims to provide a collection of well-documented and feature-complete libraries that are compatible with all versions of Clojure at and beyond Version 1.3.0. The project is called Clojurewerkz and can be found at http://clojurewerkz.org/.

Clojure's standard library is quite large and is growing increasingly with every new version. You can view the latest and other versions of the Clojure standard library documentation at http://clojure.github.com/clojure/. There's also a community-driven website dedicated to example-based documentation for Clojure's standard and contributed libraries at http://clojuredocs.org/. You can also view the source code of the Clojure language at the following GitHub page: https://github.com/clojure/clojure/.

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
Database domains

You could argue that SQL libraries are probably the most popular and widely used form of an Internal DSL. Because of this, we're going to look at the database problem domain with a Clojure Internal DSL library called Korma. If this book doesn't cover Korma as much as you would have liked, you can visit the official website for more details at http://sqlkorma.com/.

To fully appreciate a feature-rich DSL, we have to understand the complexities under the hood. For example, this is what a raw SQL query may look like:

```sql
SELECT name
FROM customer
WHERE id = 10
```

Now let's look at this same SQL query with a Clojure-based Internal SQL DSL:

```clojure
(select customer
    (fields :name)
    (where {:id 10}))
```

Looking at the two DSL code examples side by side in the following table, what conclusions can we draw from this?

<table>
<thead>
<tr>
<th>DSL for the database domain</th>
<th>Clojure DSL for the database domain</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT name</code></td>
<td><code>(select customer</code></td>
</tr>
<tr>
<td><code>FROM customer</code></td>
<td><code>(fields :name)</code></td>
</tr>
<tr>
<td><code>WHERE id = 10</code></td>
<td><code>(where {:id 10})</code></td>
</tr>
</tbody>
</table>

Here are four points that I think are very important to keep in mind when structuring a language:

- Some DSLs are based on other DSLs such as SQL
- The syntax of a DSL doesn't always make the process shorter or longer in terms of line count
- The DSL can't (or really shouldn't) remove the nomenclature of the problem domain
- Depending on the type of operations being performed, a DSL may be more verbose in some very specific circumstances

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
Now let's look at another example in the following table where the difference is mostly the line count:

<table>
<thead>
<tr>
<th>DSL for the database domain</th>
<th>Clojure DSL for the database domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT INTO customer (name, age) VALUES ('Hickey', 200)</td>
<td>(insert customer (values {:name &quot;Hickey&quot;, :age 200}))</td>
</tr>
</tbody>
</table>

Comparing SQL's SELECT statement with Korma's SELECT statement, the two weren't all that different. The same could even be said about the INSERT statements of SQL and Korma. So why would we use such a DSL if the language isn't drastically different?

Because it's easier than writing a database connection from scratch, then having to write raw SQL for every database request, and then properly casting the database data types so that the host language can manipulate the results properly. Remember, a DSL doesn't remove the problem domain; it separates much of the complexity involved with the problem domain.

Let's take a look at a different Internal database DSL by the name of clojure.java.jdbc. This DSL is actually used to build the Korma SQL DSL and relies heavily on Java libraries such as java.sql and javax.sql. We'll briefly compare the two Clojure libraries to better understand how DSLs of the same problem domain can have very different syntax.

Korma uses clojure.java.jdbc, but clojure.java.jdbc actually requires you to write SQL in certain circumstances. Korma has the ability to run SQL as well, but it tries very hard to create higher-level abstractions in comparison to the clojure.java.jdbc library. Let us compare raw SQL to the Korma and clojure.java.jdbc libraries, and see what observations we can make about their differences.
An Overview of Domain-Specific Languages with Clojure

Here are the conclusions we come to from the following table:

<table>
<thead>
<tr>
<th>Raw SQL</th>
<th>Korma Internal DSL</th>
<th>clojure.java.jdbc</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT name FROM customer WHERE id = 10</td>
<td>(select customer (fields :name) (where {:id 10}))</td>
<td>(sql/with-connection mysql-db-connection (sql/with-query-results rows [&quot;SELECT name FROM customer WHERE id = ?&quot; 10] rows))</td>
</tr>
</tbody>
</table>

Here are some key observations that you may find interesting:

- The SELECT operations of `clojure.java.jdbc`, act more as a language wrapper than a proper DSL.
- The Solution of `clojure.java.jdbc` in comparison to Korma's solution, requires you to directly write the language of the problem domain.
- Both the raw SQL and Korma solutions are easier to read in this case because their solutions are smaller than the `clojure.java.jdbc` solution. Keep in mind, though, that the smallest solution doesn't always mean the best or easiest solution to work with. Let's take a better look at `clojure.java.jdbc` by comparing its INSERT operations in the following table:

<table>
<thead>
<tr>
<th>Raw SQL</th>
<th>Korma Internal DSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT INTO customer (name,age) VALUES ('Hickey',200)</td>
<td>(insert customer (values [:name &quot;Hickey&quot; :age 200]))</td>
</tr>
</tbody>
</table>

For More Information:  
www.packtpub.com/clojure-for-domain-specific-languages/book
We can already see that the `INSERT` solution of `clojure.java.jdbc` feels more like a DSL than its `SELECT` solution. For one, you don't have to write any SQL, which is one of the main reasons why you would use an Internal SQL DSL. Also, notice how, in comparison to Korma's solution, the line count is the same but the character count is much more.

This portion of the book was a just a small glimpse at what these two Clojure libraries can do. I would encourage you, the reader, to explore them a little bit more for a better understanding of their full capabilities. If you wish to know more about the Korma SQL project, you can visit the official website at http://sqlkorma.com/. If you wish to know about the `clojure.java.jdbc` project, you can visit the official GitHub repository at https://github.com/clojure/java.jdbc.

If you're not interested in using a SQL database with Clojure, you can try any of the following NoSQL Clojure solutions:

- **Redis**: It is available at https://github.com/ptaoussanis/carmine
- **Apache CouchDB**: It is available at https://github.com/clojure-clutch/clutch
- **Apache Cassandra**: It is available at https://github.com/pingles/clj-hector

There are too many NoSQL solutions to list them all, but most Clojure libraries can be found at http://github.com.

You can also find a list of different NoSQL database solutions at http://nosql.findthebest.com.

### The HTML domain

The HTML problem domain has quite a unique collection of Clojure libraries that can make working with HTML feel like a different language altogether. Many Clojure HTML DSLs and libraries often just generate code based on native Clojure data types; in the process, it reminds us all how writing HTML directly can give us a headache. We'll compare some Clojure HTML libraries and see how these tools can help reduce development time.
An Overview of Domain-Specific Languages with Clojure

Formative

The following is a basic HTML form that accepts four inputs. These four inputs are username, e-mail, a checkbox, and a submit button. This form will make a HTTP POST method to the location "/guestbook". The submit button in this form also displays "Submit Form" instead of the default "Submit" value.

```html
<form method="POST" action="/guestbook">
  Username
  <br />
  <input type="input" name="username" value=""></input>
  <br />
  Email
  <br />
  <input type="email" name="email" value=""></input>
  <br />
  <input type="checkbox" value="true" name="agree"></input> Agree
  <br />
  <input type="submit" value="Submit Form" />
</form>
```

As you can see, XML syntax-based languages such as HTML can cause a developer to write a whole lot of brackets for even the simplest of tasks. Also notice how most tags require you to write the name of the tag both at the beginning and at the end of the element definition. Writing HTML without a library or a DSL will often take more time than you might have anticipated because of the tedious nature of writing XML syntax.

The following is the same form written in a Clojure Internal DSL called Formative. Formative is a Clojure library that tries to solve the problem of making HTML forms in Clojure. If you wish to know more about Formative, you can view the official GitHub project page at https://github.com/jkk/formative.

```clojure
(f/render-form
 {:method "POST"
  :action "/guestbook"
  :fields [{:name :username}
    {:name :email :type :email}
    {:name :agree :type :checkbox}]
  :submit-label "Submit Form")
```

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
Formative is the clear winner in comparison to raw HTML. We can see that the three of the four inputs named username, email, and checkbox are defined within the form fields with the form key named :fields. Also notice how a submit button doesn't have to be defined in this form. That's because the submit button code is automatically generated with our form when rendered to HTML. You might have also noticed that the form POST location and method are set with the form keys :method and :action.

### Hiccup

Hiccup makes HTML form building somewhat exciting, but let's see how we can develop this same form in another DSL and Clojure library called Hiccup. Hiccup is like Formative in the sense that it generates HTML after converting Clojure data, but Hiccup isn't restricted to only forms. Hiccup provides two methods for us to make HTML forms. The following code is the same form as the previous HTML and Formative examples, but this time it's written in the Hiccup DSL:

```
(form-to [:post "/guestbook"]
  (label "username-label" "Username")[:br]
  (text-field "username")[:br]
  (label "email-label" "Email")[:br]
  (email-field "email")[:br]
  (check-box "agree")
  (label "agree-label" "Agree")[:br]
  (submit-button "Submit Form")
)
```

This may look a little crowded in comparison to Formative's solution, but this is still clearly better than writing HTML directly. We don't have to write the tag name of elements more than once when defining an element. We also don't have to write anything that feels as if it's outside the Clojure language, so there's almost no learning curve when developing with Hiccup.

Now that we've seen a Hiccup form using Hiccup helper methods, let's take a look at a Hiccup form without any helper methods. The following is the same form as Formative's and the last Hiccup example:

```
[:form {:method "POST"
      :action "/guestbook"]
  [:label {:name "username-label"} "Username"]
  [:br]
  [:input {:type "text"
            :name "username"
            :value ""}]
  [:br]
```

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
An Overview of Domain-Specific Languages with Clojure

```clojure
[:label {:name "email-label"} "Email"]
[:br]
[:input {:type "email"
      :name "email"
      :value ""}]
[:br]
[:input {:type "checkbox"
      :name "agree"
      :value "true"}] " Agree"
[:br]
[:input {:type "submit"
      :value "Submit Form"}]
```

This is obviously the more verbose way to build a HTML form without writing HTML directly. Also, building forms this way is almost as time-consuming as writing HTML directly and somewhat defeats the purpose of using a DSL. If you wish to know more about Hiccup and its capabilities, you can visit the GitHub project homepage at https://github.com/weavejester/hiccup.

**Mustache**

Let's take a look at one more Clojure library and DSL before moving on. The library that we're about to look at is a parser for an External DSL language by the name of Mustache. Mustache is an External logicless templating DSL that is mainly used for templating websites, but its uses aren't restricted to strictly webpage templating. If you wish to know more about the Mustache templating language, you can visit the project's GitHub homepage at http://mustache.github.com/.

Mustache's templating syntax ignores everything that's not a Mustache tag. The tags are then replaced with data supplied to the parser and then the parser returns the data within the template. What's unique about Mustache's templating language is that it doesn't have common flow control tags such as `else` and `if`.

**Clostache**

Because Mustache is an External DSL, we'll have to use a Clojure library to parse Mustache templates. There are a few good Clojure libraries that can parse Mustache, but we're only going to focus on one in this example. Clostache is a Mustache parsing library that supports multiple versions of Clojure and is compliant with the Mustache specification.

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
We're going to compare the Hiccup library and the Mustache library Clostache when programmatically generating the following sample HTML code:

```html
div
  a {href="#"} Link Red </a>
br/
  a {href="#"} Link Black </a>
br/
  a {href="#"} Link Yellow </a>
br/
  a {href="#"} Link White </a>
br/
</div>
```

First, let's look at the Hiccup solution, and then compare it to the Mustache solution. The following is one way we could generate the sample HTML with the Hiccup library:

```clojure
[:div
  (interleave
    (for [c ['Red 'Black 'Yellow 'White]
      :let [link [:a {:href "#"} (str "Link " c)]]
      link)
    (repeat 4 [:br]))]
```

This example of the preceding code probably looks insane to you if you're a Clojure beginner, but this code will produce the same code as the previous HTML example. This isn't the only way to do it, but it's one of the nicer ways of generating the sample HTML. Let's try this again with Mustache and the Clostache Clojure library, as follows:

```clojure
{def template "
<div>
  {{#colors}}
    <a {href="#"}>Link {{color}}</a>
  {{/colors}}
</div>"
(clostache.parser/render
template
{:colors
  [{:color 'Red}
   {:color 'Black}
   {:color 'Yellow}
   {:color 'White}]})
```

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
Looking at this example may make more sense than the previous Hiccup solution that we just looked at, but there are some things that we need to understand about Mustache to fully understand this solution. The {{#colors}} tag tells Mustache to loop through our Clojure collection of colors. Inside this loop, there's the {{color}} tag. The {{color}} tag belongs to the {{#colors}} collection and Mustache automatically assumes that this is the color inside the {{#colors}} collection loop.

As we can see, there are many ways to use different libraries to achieve the same result in this problem domain. Some methods and libraries are obviously better in their own respects, but choosing a library and an Internal DSL is a combination of preference and project requirements. Clojure's ability to use either Java or Clojure libraries will give any developer plenty of solutions to choose from.

**The ECMA/JavaScript domain**

There are many DSLs that ultimately compile into the JavaScript language. There's CoffeeScript for Node.js, Script# for C#.Net, and Haxe that has native compile to JS features. Clojure, on the other hand, has an alternative compiler just to handle JavaScript.

**ClojureScript**

The alternative compiler doesn't come with the **Java Virtual Machine (JVM)** implementation of the Clojure language, so you'll have to download the compiler from a GitHub project called **ClojureScript**. The project page can be found at [https://github.com/clojure/clojurescript](https://github.com/clojure/clojurescript). Because it might be a while before you want to go through all the trouble to set up the compiler, you can run ClojureScript directly in your web browser using the ClojureScript website [https://himera.herokuapp.com/index.html](https://himera.herokuapp.com/index.html).

ClojureScript's target language is JavaScript, so using any Java interoperation functions on objects will no longer work. We'll have to use JavaScript interoperation instead, so the knowledge of JavaScript language is required to some extent to get the best out of ClojureScript's capabilities. We'll cover some of the differences between ClojureScript and the JavaScript language, but you can view a full comparison chart at [https://himera.herokuapp.com/synonym.html](https://himera.herokuapp.com/synonym.html).
Comparing ClojureScript and JavaScript

The JavaScript variable definitions are similar to the C language's style of defining variables. The following is a JavaScript variable named `car` that holds the value `Red`:

```javascript
var car = 'Red';
```

The following is the same JavaScript variable written in ClojureScript. Notice that this syntax is also how you would define an object named `car` in the Clojure JVM implementation:

```clojure
(def car "Red")
```

Let's look at a more interesting example where the JavaScript actually does something, and let's see what the same code would look like in ClojureScript. The following is a JavaScript example of destructuring parts of an object. The parts of the object are then placed within a string and alerted in a browser window as follows:

```javascript
var example = {type: "car", color: "red"};
var vehicle = example.type,
    color = example.color;
alert('The '+vehicle+' is '+color);
```

Here's how we would produce the same results with ClojureScript. The following example displays local variables being assigned values using `let`. The first local variable `example` is assigned to what looks like a JSON JavaScript object, but what is actually a key value set. The next two local variables are `type` and `color`, and are assigned by matching the key values in the local variable `example`. What you'll see after the local variable definitions is JavaScript interoperation with the call to `(js/alert)`. This will make an alert message window within the web browser and display the local variables within a string, as follows:

```clojure
(let [example {:type "car":color "red"}
    {:keys [type color]} example]
(js/alert
 (str "The " type " is " color)))
```

Let us take a look at using dynamic bindings in JavaScript and then rewrite the same code in ClojureScript:

```javascript
var x = 1, y = 1, doubleIt = { x: x*2, y: y*2};

with(doubleIt) {
    alert(x*y);
}
```

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
An Overview of Domain-Specific Languages with Clojure

The preceding JavaScript example will define two variables with the same value of 1. Those values are x and y, and they will have their values changed when with uses the doubleIt object to locally bind new values to x and y within the scope of the with curly braces. If you were to run this JavaScript in your browser, you would get an alert message window stating that x multiplied by y is 4.

Now let's see how the same thing can be done in ClojureScript, as follows:

```clojure
(def ^:dynamic x 1)
(def ^:dynamic y 1)
(binding [x 2 y 2]
  (js/alert (* x y)))
```

The preceding example is the same as the JavaScript concept. Both variables have a local value of 2 within the braces of binding. Then the multiples of both values are alerted in a browser message window. One thing that's very different in this example as opposed to the JavaScript example is that the variables have metadata tags in front of the variable names. This metadata setting is ^:dynamic and is needed to rebind the values, because by default, Clojure doesn't allow the rebinding of defined objects. The metadata allows Clojure to know that this object should be able to change and allows us to bind other values to the object definition. Unlike JavaScript, the mutation of a Clojure object can't be done unless the object type is explicitly mutable and mutated explicitly (think of the swap! function). Keep this in mind when working with Java objects because you might treat it like a Clojure object and be very surprised when you call a function that changes the value of your object.

For loops in JavaScript remind us of our days programming C, but Clojure and ClojureScript take a different route and we're about to see just how. The following example is a JavaScript for loop that counts down from 3 and does nothing when reaching 0. Before the loop reaches zero, the web browser will alert three messages that display the current number that the countdown is currently at.

```javascript
for(i = 3; i >= 1; i--) { alert(i); }
```

The preceding example was a simple one liner, but let's take a look at the ClojureScript way of doing it:

```clojure
(loop [i 3]
  (if-not (zero? i)
    (do (js/alert i)
      (recur (dec i))))))
```
Loop defines a local variable \( i \) with the value of 3. Within the loop body, you'll see that action is taken if the variable \( i \) isn't equal to zero. If the variable isn't equal to zero, the web browser will send an alert message displaying the current number of countdown. After the message is displayed, \texttt{recur} changes the local variable bindings by decrementing \( i \) with the \texttt{dec} function before the loop restarts.

Let's look at one last ClojureScript example before moving on to other domains. The following example is defining a JavaScript object type that has a function named \texttt{fullTitle}. When this function is called, the web browser makes an alert message window displaying a formatted version of the author and title of the book object, as follows:

```javascript
function Book(author,title) {
  this.author = author;
  this.title = title;
}

Book.prototype.fullTitle = function() {
  msg = "Author: " + this.author + "\r\nTitle: " + this.title;
  alert(msg);
}

var book = new Book("B. Writer","Clojure All Day");
book.fullTitle();
```

The following is the ClojureScript solution for the same result. We define the type, \texttt{Book}, as a child to the JavaScript object, \texttt{Object}, and implement a function named \texttt{full-title}. One difference you might have already noticed is that the function name is in front of the defined variable and is prefixed with a decimal. This is because we're calling a method function that belongs from within the defined JavaScript object as opposed to calling an externally defined function.

```clojure
(deftype Book [author title]
  Object
  (full-title [_]
    (js/alert
      (str "Author: " author "\r\nTitle: " title))))

(def book (Book. "B. Writer" "Clojure All Day"))

(.full-title book)
```

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
**The Audio domain**

Although you can use Java audio libraries and DSLs within Clojure, there are only about two really solid DSLs for synthetic audio manipulation from within the Clojure language. This is not only because Clojure is a very young language, but also because these libraries are very good at what they do. Because some of the many DSLs in this section can get very complex, we’ll look at only some of the more basic uses of them.

**Music-as-data**

The first Clojure library and Internal DSL we’ll look at is named **Music-as-data**. This is a project aimed at easy note, pattern, and sample playback. The notes are synthetics and the samples are bound to the WAV files on the project's classpath.

Although the documentation seems to have fallen behind on the project, there are some examples and snippets that still work and are worth covering. If you wish to know more about the project, you can view the project's GitHub homepage at https://github.com/jonromero/music-as-data.

You might wonder why we would use examples from a project with out-of-date documentation? The documentation might not be up-to-date, but the principles of a DSL are very strong in this particular library. For example, you can create instruments, samples, and chords without any knowledge outside the Clojure language with the exception of knowing the names of the musical notes you wish to play.

It's okay if you're not a music theory major. No knowledge of music theory is required to exploit this library to the fullest extent. Luckily for us, there's a function that helps us play musical notes. We can use the `(p)` function to playback a sequence of musical notes. We can also use `(pattern)` and `(chord)` to modify how the note sequences are played.

If we wish to play each note at the same speed, we will simply use the `(p)` function and a sequence of musical note symbols. If we want the speed of the playback to progress for each musical note, we will wrap the sequence of notes in the `(pattern)` function. The `(pattern)` function not only accepts a sequence of musical notes, but also accepts a duration integer, which will play each note for the stated duration. The following are two examples of playing the musical pattern of the notes A4 and B4. The second of the two examples has a duration of 5 seconds of play for each note within the musical sequence, as follows:

```clojure
;; Example 1
(p (pattern [A4 B4]))
;; Example 2
(p (pattern [A4 B4] 5))
```

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
Let's say we want to make a little melody but we don't really know anything about audio programming. First, we should define the initial part of our melody and call it `pattern-1`. The following is an example of the `pattern-1` definition to start our series of musical notes:

```
(def pattern-1 (pattern [A4 B4 C5 D5] 3))
```

Let's also define the second pattern of our melody, and since we know nothing about music theory, let's reverse `pattern-1` in a definition of `pattern-2` and call it a day:

```
(def pattern-2 (pattern [D5 C5 B4 A4] 3))
```

Now that we've defined two parts of our melody, let's combine the patterns and play the melody to hear what we've created. Let's not forget that the `(p)` function allows us to play a collection of musical notes and patterns, as follows:

```
(def melody (flatten (merge pattern-1 pattern-2)))
```

```
(p melody)
```

If you don't already know, we have to merge the two pattern sequences with the `(merge)` function, and then flatten the collection of note patterns into a single collection of note patterns with the `(flatten)` function.

### Overtone

If you know the mathematics of music theory, the Music-as-data library and Internal DSL probably don't meet your musical needs. Thankfully, there's a library that allows a musical theorist to express even more complex sounds; this library is known as Overtone.

Overtone is by far the most complete and well-documented Clojure sound library and the project's GitHub homepage can be located at https://github.com/overtone/overtone. Overtone is the obvious leader in Clojure synthetic sound-generation and sound-pattern composition. Unlike the Music-as-data library, the understanding of music at its core is needed to take full advantage of this library and Internal DSL.

We'll try to play the same melody as we did in the Music-as-data example, but we'll use the most basic features of the Overtone library to jazz it up a bit. Any serious musically-inclined software developer should study the Overtone documentation to better understand how to exploit this DSL. Before we start working on our melody, we'll have to define some playback helpers and which instruments they'll use.
An Overview of Domain-Specific Languages with Clojure

Overtone has a collection of instruments already built-in, so we can get started right away as we did with the Music-as-data library. In the following example code, we’re telling Overtone to use the built-in piano:

```clojure
(ns test.core
  (:use [overtone.live]
        [overtone.inst.piano]))
```

Next, we'll need to define a function to play our melodies. In the following example code, we're going to specify a definition named \(p\). It will accept a collection of musical notes named \(chord\) and a real number (not negative) named \(delay\). \(doseq\) will then take each note out of the chord sequence and play the piano note before forcing the program's current thread to wait for the amount of delay time specified multiplied by 100 milliseconds, as follows:

```clojure
(defn p
  [chord delay]
  (doseq [f chord]
    (piano f)
    (Thread/sleep (* delay 100)))))
```

Now that we have a function that can play a musical sequence with a piano, we'll have to define a pattern that will be the foundation for the rest of the patterns. We'll also need a function that can translate our pattern into piano-friendly notes. We can call this function \(builder\). The \(builder\) function takes a note sequence named \(chord-pattern\) and a setting of either major or minor with the required variable named \(m-or-m\). This function will take every note out of our chord pattern and apply the \(chord\) function to each note. After each note has been turned into another pattern of notes, the \(builder\) function uses the \((flatten)\) function on the newly-generated patterns to make sure all of the note sequences become one sequence for our instrument to play. Once all of this is done, the \(builder\) function will return an instrument-playable format. The following is an example of this function:

```clojure
(def pattern
  [:A4 :B4 :C5 :D5])
(defn builder
  [chord-pattern m-or-m]
  (->> chord-pattern
       (map #(chord % m-or-m))
       flatten))
```

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
Since we have our components for making a basic melody, let's define our first two patterns. The first pattern is pretty simple and uses the `builder` function on our already defined pattern named `pattern`. This pattern is called `pattern-1` and it will play our pattern in minor chords. Our second pattern named `pattern-2` is simply reversing the first pattern we made. The following code is an example of this:

```clojure
(def pattern-1
  (builder pattern :minor))

(def pattern-2
  (reverse pattern-1))
```

Now that we have two musical patterns, we can define a melody named `melody`. This definition will repeat our patterns twice with the `repeat` function and then flatten the musical sequences into one musical sequence. The following is an example of this very definition:

```clojure
(def melody
  (flatten
   (repeat 2 [pattern-1 pattern-2])))
```

Now that we have a melody and a function that can play the piano for us, let's play each note in the melody with the play delay of 3. The following code is an example of this:

```clojure
(p melody 3)
```

**Image domains**

Because Clojure is such a young language, only a few image libraries and DSLs are written in Clojure. **Quil** is a library and DSL that can handle both the creation and animation of images within the Clojure language. Quil is based on a programming project that specializes in image and animation creation; this project is called **Processing**. Because image generation is as difficult as, or even more difficult than, audio generation, we'll only cover some basic image generation examples. You can find out more information on both these projects by visiting the documentation section of Quil's GitHub project homepage at https://github.com/quil/quil.

Quil is very easy to use but a knowledge of graphing coordinates and geometry is required to exploit this library to the fullest extent. Don't feel afraid if you're not a mathematician. This section of the chapter only covers basic line drawing and animations.

For More Information:
www.packtpub.com/clojure-for-domain-specific-languages/book
Quil provides a Clojure macro named defsketch and allows us to send our drawing information to the main application. This macro accepts an application name and a few named options for rendering the final results. We'll define a still-frame image within defsketch by using what's called an anonymous function (a function without a name).

This is an explanation of the following example of how to draw vertical lines with Quil. The anonymous function for the setup option declares that the image should use an anti-aliasing filter with the function named smooth. The anonymous function for the draw option declares the width of the line with a function named stroke-weight and then states that a line should be drawn. The line function accepts four arguments in the form of \(x_1, y_1, x_2, \text{ and } y_2\). The starting point of the line is obviously \(x_1\) and \(y_1\) with the ending point of the line being obviously \(x_2\) and \(y_2\).

```
(defsketch drawing
  :title "Book Example"
  :setup (fn [] (smooth))
  :draw (fn []
    (stroke-weight 9)
    (line 100 50 100 100))
  :size [200 200])
```

The difference in the y axis coordinates is what draws the line this way, if you didn't know. To make this line horizontal and grow longer for every passing second, we will have to make a few interesting changes to our anonymous function for the setup and draw options. First the frame-rate function will have to be called for the setup option. This allows Quil to know that this image is indeed an animation. Secondly, we will define an atom named right to hold the \(x_2\) value for our function as the line animates closer to the right of the window. Last but not least, the anonymous function for draw will increment and return the value of the right atom when the function is called. The following code is an example of these changes:

```
(def right (atom 55))

(defsketch drawing
  :title "Book Example"
  :setup (fn []
    (smooth)
    (frame-rate 3))
  :draw (fn []
    (let [x2 (do (swap! right inc) @right)]
      (stroke-weight 9)
      (line 50 100 x2 100))
  :size [200 200])
```
Summary

A DSL is a language dedicated to solving only one type of problem. The language can't make the problem domain less complex, but it can separate the complexities of the problem for the person using the language. Generally speaking, a DSL only simplifies the nomenclature of the problem domain but never completely removes it. The same terms used to solve the problem need to persist within the language, so that the expressions can be written and understood by those familiar with that particular domain.

An Internal DSL is built on top of or within another language and uses that language as its core foundation. Clojure's power and flexibility allows for highly expressive Internal DSLs to be built quickly and cleanly. Although this book doesn't cover External DSLs, Clojure has many External DSL-friendly features such as code generation, the ability to have nonalphabetical definitions, and metadata, to name a few.

As we can see, there are usually competing languages in each problem domain. Some of these languages can actually increase the complexity or the time needed to solve our problems if certain aspects of the languages we use aren't taken into consideration. We should also ask ourselves if we even need a DSL when trying to solve our problems, because sometimes they add more layers of complexity than the original problem.

In the next chapter, you'll learn about editing Clojure programs with the Emacs and Leiningen applications.

For additional information and documentation, please refer to the chapter information sources in the next section.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.