1 Introduction

Programming is hard. Programming for the web may be harder. And none of it is made any easier by the tools at our disposal.

FLAS and Ziniki are a systematic attempt to solve many of these problems. There will always remain an *essential* core of programming (what do you want and what will you call it?) that will remain complex, but FLAS and Ziniki are designed to strip away as much of the *inessential* complexity as possible.

Web applications consist of three separate entities:

- visual elements usually presented in HTML and CSS;
- browser-side code written in JavaScript;
- server-side services written in a variety of languages.

Among the problems encountered by web programmers on a daily basis are: the overlap and intermingling (quite often, it seems, intentional) between the visual presentation and browser-side code; the duplication of code on client and server sides because it cannot be effectively shared; a large body of code aimed exclusively at connecting client and server sides; a similar amount of code required to communicate knowledge between client and server; and a significant amount of "boilerplate" code needed to repeat patterns - particularly between applications. All of this complexity pushes exciting, user-friendly features - such as continuous state updates and notifications - onto the "nice to have" list.

FLAS and Ziniki address these problems systematically.

- There is a clear separation of concerns: visual elements are delivered in HTML and CSS; the FLAS compiler scans the HTML to build a model of these and then allows them to be addressed in constructing template models in the program, written in FLAS.
- Code can be easily shared between client and server because it is all written in the same programming language, FLAS; distinct constructs exist to create code which is specific to the client or server environment.

- The Ziniki environment contains a peer-to-peer connector (subsuming the normal client-server relationship) which automatically handles all communication; the notion of message passing is built into the bedrock of the FLAS/Ziniki programming model.
- Changes anywhere in the distributed Ziniki environment are automatically detected and passed to all interested parties. Internally, the FLAS runtime is able to update client displays with little or no code intervention (the little is handling conflict resolution).
- Ziniki inherently knows about concepts such as users, logins, security, access, privacy, sharing, money, deals, analytics and so on. It automatically handles as much as is possible without programmer intervention. When programmer intervention *is* required, we claim that the right access points exist to make the intervention as smooth and easy as possible.

Although complementary, FLAS and Ziniki are separate things: it is possible to write web applications in FLAS without using Ziniki. This guide describes FLAS programming completely independently of the Ziniki network.

In addition, we consider FLAS to be a modern, agile programming language:

- it natively supports unit and system tests;
- it is strongly typed, but uses type inference and supports the `Any` type for simplicity and flexibility;
- it is a lazy functional language from the ground up;
- it produces code for the web, server applications and Android-compatible mobile applications.

1.1 Organization of this Guide

This developer guide is presented as a series of challenges to be overcome. Each chapter presents a challenge, the approach taken to solve it, the resulting HTML, CSS and FLAS code and then adds some commentary on the language and its features.

As is typical with programming languages, Chapter 1 describes a very simple "hello, world" application. As with all such toys, it is mainly about getting you in a position where you can be confident that you have met the prerequisites and understand the toolset before trying to understand the language itself.

Chapter 2 introduces the actor model and how cards can be created to store state and respond to user events, seamlessly updating the display.

Chapter 3 extends this by showing how the state can be used to control the styling and interact with CSS.

Chapter 4 puts a lot of this together to build a gadget that supports a "polymorphic tree" - a tree with multiple different types of nodes and leaves - and handles expanding and contracting the branches.

Chapter 5 attempts to demonstrate that it is possible to build something that most people would agree was a complete web application without getting into the complexities of needing a server: a fully-functional game of patience (or solitaire). Approximately 400 lines of FLAS code combine with 50 lines of HTML and 150 of CSS (including blank lines) to make a complete, standalone, web game.

The samples can all be downloaded from GitHub and can be run directly from the web.

1.2 Prerequisites

The FLAS compiler is written in Java. In order to use FLAS, you must have a Java runtime - at least version 11 - installed. A Java Development Kit (JDK) is *not* required. Obviously, you must also download and install the latest FLAS compiler.

Because everything in this guide is about web applications, you must have a browser installed. While everything in FLAS is intended to be 100% HTML5 compliant, we develop with Chrome and recommend that environment. Obviously, where possible, applications should be tested across an array of environments.

While we claim that FLAS will reduce much of the *inessential* complexity of programming, it *cannot* remove the fundamental complexity that *is* programming. We assume that you are familiar with the concepts of programming and at least have some limited exposure to web programming, probably in JavaScript.

Because FLAS builds on HTML and CSS, some familiarity with these technologies is required. However, detailed knowledge - or the ability to write them by hand - most certainly is not. FLAS takes a "bring-your-own-HTML" approach to building web applications and any design tool that spits out HTML and CSS and allows you to customize the `id`s of the HTML elements can be used to construct the web input to FLAS. The simple examples shown in this book were all written by hand - precisely to keep them simple. When building serious web applications, we use - and recommend - WebFlow.

1.3 Other Reading

FLAS Reference GuideZiniki manualsWebFlowThere should be a guide to the Expenses Demo

2 Hello, World

While we are not certain exactly where the tradition began, ever since Kernighan and Ritchie published "The C Programming Language" in 1978, it has been traditional to start every programming language text with a simple "Hello, World" program. Who are we to break this tradition?

Our "Hello, World" program consists of three parts:

- the HTML needed to render a greeting;
- the FLAS Card which amounts to a "program";
- a small "assembly" file which tells the compiler how to build a web application from the parts.

2.1 HTML

There is a limit to how trivial an HTML file is allowed to become. According to the specification it must have an HTML container, a head and a body. Aside from that, this HTML consists of a pair of nested `div`s, one to hold the overall "application" and one to hold the "greeting" itself.

```
<html>
<head>
</head>
<body>
    <div id='flas-card-message'>
        <div id='flas-content-message-area'></div>
    </div>
</body>
</html>
```

The important things to note here are the `id` specifications on the `div` elements. Each `id` must be unique within an HTML document; when present, FLAS uses these in the input document to identify the role that the element will play in the application. The rules by which this works are somewhat complexand are described in the reference manual.

2.2 FLAS

The FLAS code builds on the HTML by defining a `Card` which finds the appropriate `div` and placing the 'hello, world' greeting in it.

```
card Hello
    template message
        message-area <- 'hello, world'
```

2.3 Assembly File

We tried very hard to avoid all kinds of artificial packaging files and steps common in other programming environments where they have generally been added as an afterthought, but at the end of the day there is *some* information which does describe how the application is "assembled" rather than what the application "is". This, very limited, information needs to be placed in an assembly file.

```
application
    title "Hello, World"
    routes
        main <- Hello
```

Basically, an HTML application *needs* to know two things: what *title* should I put on the window or tab displaying this application; and which card (of the potentially many you have given me) should I treat as the "main" one? These are the questions answered in this assembly file.

2.4 Compiling and running the program

Before you can run this program, you need to compile it. Before you can compile it, you need a compiler.

Compiler packages can be downloaded from `www.ziniki.org/downloads`, although currently the compiler package is only available for MacOS and Linux[1] The download is in the form of a zip which you need to unpack to a directory somewhere on your computer and has nested `bin`, `lib` and `flascklib` subdirectories. You should then ensure that the compiler is on your `PATH`. As noted in the introduction, running the compiler also depends on you having Java installed.

All of the samples described in this manual are available on github at `github.com/zinikiGareth/zinikiDemos`.

If you use eclipse, you can install a plugin from the Ziniki P2 repository at `not yet available`.

Assuming you have done the above, you can change into the appropriate sample directory

```
$ cd zinikiDemos/samples/hello
```

and compile and run the `hello` sample

```
flas --web ui org.zinapps.samples.hello --html hello.html --
```

and after a little thinking and whirring, "hello, world" should appear in a new browser window.

Commentary

2.1c Bring Your Own HTML

[1] Although Java is essentially cross-platform, FLAS uses the JavaFX library to run JavaScript unit tests. This library is platform-dependent. The only obstacle to supporting other platforms is a packaging one.

Separation of concerns is a very important part of any well-structured application. The concerns of visual design and application development should be kept very separate and easily maintained independently, with only the minimal loose coupling between the two[2] In embedding this approach within the FLAS compiler, we gave it a name: "Bring your own HTML".

In most web applications I have worked on, the HTML and the code are intertwined. This reaches its apogee in languages and tools such as PHP, ASP and JSP, where the code is literally written in and around each other. Why do we think this is such a problem?

Quite simply, programming languages and HTML have tools that operate on them. If the two are mixed together, the result is programming language fragments that no longer have static meaning - the code has to be executed to produce either a complete set of HTML or a complete set of code - which means that the tools designed to work on either are useless, making the developers' jobs more difficult.

Loose coupling is a well-established technique for connecting two distinct systems that nevertheless need to coordinate their activities. FLAS uses loose coupling to reference the content of the HTML files from within a FLAS program.

It does this using the concept of *templates*. In the popular model-view-controller (MVC) paradigm for building web applications, a separation is made between the data (model) and the presentation (view). In FLAS, the view is constructed hierarchically of a variety of elements, the two of which we have seen so far are the *card* and the *template*.

[2] This is not to say that they should be developed independently or by different teams. In our opinion, the HTML and the application should be delivered alongside each other, by a single team although it is to be expected that there will be different specializations on the team. But the team should develop the two portions independently using the appropriate tools.

Each template in FLAS is introduced with the keyword `template` and an identifying *name*. The loose coupling is provided by the requirement that the HTML has a corresponding element with the `id` `flas-card-`*name*. FLAS automatically extracts and analyzes this element, recording any nested elements with appropriately named `id`s. In this example, there is one nested element, the div `flas-content-message-area`. This "content field" is called `message-area` and is the one referenced by our program in setting the content to be "hello, world".

3 Actors

In order to handle user interfaces, it is necessary to do more than just render templates. It is also necessary to manage state and to handle user events.

In this chapter, we present a simple application which builds upon "hello, world" to demonstrate how *cards* can be used as *actors* to manage state, handle events and update the UI.

3.1 HTML

Interestingly, the HTML for this chapter is completely unchanged from the previous chapter. Quite simply, the template we outlined there is entirely adequate to handle text that changes.

3.2 FLAS

The `Message` card builds on what we learnt in the "hello, world" example by defining a card with a template which presents a more generic "greeting" and allows this to be toggled between `hello` and `goodbye`.

```
card Message
    state
        String greeting <- 'hello'

    template message
        message-area <- greeting

    event switchIt (ClickEvent ce)
        greeting <- newmessage

    newmessage
        | (greeting == "hello") = "goodbye"
        |                       = "hello"
```

In FLAS programs, indentation is everything: the initial indentation of a line must be a number of tabs (not spaces) and those tabs indicate the *indentation level* of the line. Lines are logically "included within" the previous line with one less tab of indentation. Long lines can be broken by repeating the current indent level and then indenting (typically two) further spaces.

Lines which are *not indented at all* are considered comments and ignored, along with blank lines. Additionally, as with C++, Java, JavaScript and a number of other languages, two consecutive slash characters (`//`) cause the rest of the line to be ignored.

The first line here introduces a `card` definition. `card` here is a keyword and may only appear at the top level (i.e. with exactly one tab of indentation). The following token is the name of the card (`Message` in this case). Card names must start with an uppercase letter and have at least three characters in their name.

This card has four component elements: a *state*, a *template*, an *event handler* and a nested[1] *function definition*.

The *state* of a card is defined by a `state` declaration. Each element of the state declaration is indented one further level and consists of a *type*, a *name* and an (optional) initializer. In this case, the state member `greeting` is declared to be of type `String` and is initially set to the value `hello`.

The *template* identifies that the `message` template in the HTML file should be selected and that the content area `message-area` should be populated with the "current" value of `greeting`.

`switchIt` is defined to be an *event handler* for the `click` event - i.e. a boring mouse click. The event object is passed to the handler as `ev`, although this is not actually used.

An event handler is an example of a FLAS *method*. In reality, a FLAS method is a function that maps input arguments to a list of *messages*. However, methods may be given a special syntax so that it is clearer what is going on. In this case, we use a simple *state assignment* to create a message that says "update the state member `greeting` to be the value of the function `newmessage`".

And finally, `newmessage` is a function that uses conditional logic to determine the new value of the message. If the value of `greeting` is currently `hello`, the value is `goodbye`; otherwise, it is `hello`. The upshot of this, of course, is to toggle the message between `hello` and `goodbye` every time the card is clicked.

[1] Function definitions may appear at the top level along with cards; however, a function definition within a card is able to refer to the current state of the card by simply referring to the state members by name, whereas a global function definition would need to be passed the state members.

3.3 Assembly File

As with the previous example, the assembly file gives the minimum amount of information necessary to turn what we already have into a web application.

```
application
    title "Toggle Message"
    routes
        main <- Message
```

Commentary

3.1c Cards and the Actor Model

Returning to the MVC paradigm, we offer the *card* as the basic unit of interaction and thus as our candidate for the *controller*.

It's more complex than that, though. A card is one of three *actors* defined by the FLAS language. Actors have a long history in Computer Science, but their history is often obscure. They are derived from Simula 67 and formalised in the 1970s. This work ended up laying the foundation for what we know today as Object-Oriented Programming. However, as Alan Kay observed in the 1990s, it would have been better to call it "message-oriented programming" because the key concept is the message and how messages are distributed and processed; in Object-Oriented Programming, too much of the focus is on the objects and how they stand in relation to each other.

While the obvious modern-day derivative is the common-or-garden *object*, an *actor* should more realistically be thought of as akin to a microservice: it is something with an independent lifecycle - its own memory and processing cycle - that exclusively communicates with the outside world through message passing in a very strict form. For those familiar with the concepts of programming languages, an actor exists essentially in its own *virtual machine* with its only contact with the outside world being through message passing.

In FLAS, actors follow this paradigm very strictly: in some instances, actors may even be literally isolated within a locked-down container (such as an `iframe` or `WebWorker` in JavaScript, or a standalone JVM in Java). In every case, however, an actor *only* does work to respond to an incoming message and responds by issuing a set of messages which have different side-effects, including updating the actor's internal state and any visual components. But during the *processing* of a message, no state changes happen, no messages are sent, no more messages may be received: the evaluation context is *purely functional*.

There are many possible sources for messages within FLAS; over the course of this guide we will come across a number of them. In this chapter, we looked at the *UI Event* message: all UI events in a FLAS application are directed to the card that owns the screen real estate where they happened. Some complex events (such as drag-and-drop) may involve multiple cards, but do so through message passing.

At the start of the message processing loop, all the state members have a particular value: the first time through the loop it is the value they were initialized with (or else they are simply uninitialized); for subsequent messages, the state members have the value they were left with after the previous message was processed. These values *remain unchanged* during the processing of the message but may be updated *after* the message has been processed.

In this example, `switchIt` requests that `greeting` be updated with a new value. But there is no need to consider *which* value of `greeting` is to be considered at any point during processing: whenever it is referenced (as it is in `newmessage`) it will always have the value that it had when the message arrived.

Contrariwise, when considering the template, the value used for `greeting` will be the value *after* all the assignments have taken place. It is reasonable to look on message processing as a five-step process as follows:

- a message arrives;
- the appropriate handler is dispatched, which returns a list of "response" messages;
- the messages are processed, including updating local state and sending messages to remote actors;

- the templates are re-evaluated to determine the new contents of the display;
- the UI is updated accordingly, including changing any event handlers.

3.2c Event Handlers

Event handlers are methods which react to user interface events.

As stated above, a method is essentially a function that transforms input into output.

Event handlers are a little bit special in the way in which they are invoked, however. Event handling is innate to UI processing. When a FLAS application sees a UI event it *has* to handle it in some way. One option, of course, is to ignore the event: this is what happens whenever no handler is defined for the event.

But unlike most messages, very little information is conveyed along with an event: maybe a mouse position or button setting, maybe the key pressed on the keyboard. In light of this, every event handler looks alike: they all accept exactly one argument which is some kind of event.

It is possible, as we shall see in Chapter 4, to bind events to specific portions of the UI and have them respond appropriately; however, by default an event handler responds to the event anywhere on the card.

3.3c Functions

Functions in FLAS are essentially the same as those in standard *functional* programming languages such as Haskell. They are pure, lazily-evaluated mappings from input parameters to a single output value.

Functions may be specified in multiple parts using *pattern matching* and then each part may be broken down into conditional cases as we have done here. In every case, only one of the expressions on the right hand side (after the =) will be evaluated.

Lazy evaluation means that once an expression has been selected, the *expression itself* is returned as the value of the function, *unevaluated*. Expressions are only evaluated when the actual *value* is required. This makes it easy and pleasant to handle infinite and cyclic data structures where only part of the value will be traversed. For more information, see any introductory text on functional programming languages.

In FLAS, two things drive evaluation:

- pattern matching in function declarations;
- storing values in state or sending them as part of messages.

When passing an expression to another function, it is generally *head evaluated*: that is, enough of the structure is determined to see which pattern it matches (if any). In some cases - particularly with builtin functions, but also with patterns such as `Number`, `String` or constants, it is necessary to completely evaluate an expression in order to determine if it matches the pattern.

When storing values (or passing them off as messages, which is much the same thing), it is necessary to fully evaluate an expression (although cycles may sometimes still be acceptable). As much as anything, this is a design choice which reflects the desire to make message processing *atomic* and *transactional*: if evaluation could be deferred, "your" errors could show up later while processing a different message.

3.4c State Member Initialization

When initializers are provided for state members, these are evaluated before the card is truly created. Consequently, they cannot reference the state of other members, nor can they obtain access to the outside world. However, there are other mechanisms (such as the `Lifecycle` contract we will use in Chapter 5) by which it is possible to defer initialization until later in the cycle when more information is available. This is particularly relevant when using Ziniki microservices.

It is, however, perfectly acceptable to simply *not initialize* fields. If a field is referenced when uninitialized, an error will result but, errors being values like anything else, this will not cause the actor to fail, *per se*, although it is improbable that desirable results will follow. As with any programming language, you need to be aware of the possible states your actors can be in and what actions are valid in the current state.

4 Styling

For the past twenty years, a key to developing effective websites and web applications has been the recognition of the separation of content and styling, colloquially referred to as the "html/css divide". HTML provides the content; CSS provides the styling. In a number of ways, this separation reflects and informs the separation of concerns in FLAS between web design and application code.

While providing an abstraction over it, FLAS assumes a styling model consistent with HTML and CSS; that is, it assumes that each card is made up of a set of elements, or *areas*, each of which can be individually *styled* using an abstract mechanism entirely consistent with CSS classes. In other words, FLAS knows absolutely nothing about the real-world concerns of fonts, colors or layout, but it supports a "code book" of terms which it assumes *can be converted* into those concerns. On the FLAS side, these are simple strings; on the web design[1] side, these are CSS class names.

The CSS is written in the usual way: as a set of styles applied when certain elements are associated with certain classes. While the two sides have their separate concerns, they are loosely coupled through the *names* of the elements and the *names* of the classes.

This example builds on the previous example by taking a simple message and providing three buttons which allow it to be styled:

- By default, the message is "normal size" and black;
- `Bold` always makes the message "large size" and red;
- `Red` toggles whether the message is black or red;
- `Big` toggles the size of the message.

---

[1] This chapter describes styling from the perspective of web applications; in part, the abstraction exists precisely because there are alternatives with natively-rendered Mobile Application Frameworks. However, these are beyond the scope of this guide.

4.1 HTML

This HTML is, understandably, considerably more complex than the previous HTML. In the header it links to a CSS file (described in the next section) and the card contains a message and three buttons.

```
<html>
<head>
    <link rel='stylesheet' type='text/css' href='styling.css'
                            >
</head>
<body>
    <div id='flas-card-page'>
        <input type='button' id='flas-style-bold' value
                             ='Bold'></input>
        <input type='button' id='flas-style-red' value
                             ='Red'></input>
        <input type='button' id='flas-style-big' value
                             ='Big'></input>
        <div id='flas-style-display'>hello, world</div
                            >
    </div>
</body>
</html>
```

In the `head` portion, the link is a perfectly normal link to a perfectly normal spreadsheet. Note that this directive is not relevant to the FLAS system: it pulls the CSS files directly from the `ui` directory and includes them. It is your responsibility to make sure that they are consistent when all included at the same time.

As with the previous cards, the `body` defines one card which has the name `page`. Within this are three buttons, each of which has a unique identifier starting with `flas-style-`. This prefix tells the compiler that this element may be styled and have events added to it, but its content is not to be touched. Finally, there is a `display` element, which contains the usual "hello, world" message, waiting to be styled.

4.2 CSS

Hopefully, this CSS is sufficiently simple that no real explanation is required.

```
.big {
font-size: x-large;
}
.red {
color: red;
}
```

Essentially, elements that have the class "big" will have their text made larger; elements that have the class "red" will have their foreground color made red.

Just in case there might be any confusion, let's be clear that the names "big" and "red" have no significance whatsoever; they are merely coupling between the code, the CSS file and the HTML.

4.3 FLAS

Although the only truly new concept here is styling, this example feels like a step change from the previous two, so we are going to work through the example code step by step.

At the top level, the card is defined and called `Styling`. It has two state variables - both `Boolean` values - which are initially set to `False`. The idea is that they control (independently) whether the message is big, and whether it is red.

```
card Styling
    state
        Boolean wantBig <- False
        Boolean wantRed <- False


    ...
```

We can then control the display of the message by attaching the styles (or *classes* in CSS) to it depending on these values. We do this by first identifying a *coupling* to an HTML element by name: in this case `display` identifies the HTML element with the `id` `flas-style-display` - in other words, the message.

We then apply two *conditional styling rules* to it. In FLAS, *all* styling rules are, in fact, conditional, but the conditional may be specified as True - or simply omitted - in order to make the condition apply in all cases.

In this case, each of our two rules uses one of the boolean variables defined in the state to select the corresponding style - simply a string - and apply it to the element. FLAS takes care of combining all the various selections into a single class specification for the element.

```
card Styling
    ...
    template page
        display
            | wantBig => 'big'
            | wantRed => 'red'
        ...
```

The rest of the template consists of attaching event handlers to the button elements. Again, loose coupling is used to identify the elements in the HTML and then, for each one, an event handler binding is specified. This limits the *target area* of the event handler to this specific element in the card.

```
card Styling
    ...
    template page
        ...
        bold
            => makeBold
        big
            => makeBig
        red
            => makeRed

    ...
```

Once the template is defined, we can finish up by defining the event handlers themselves, much as we did in the last chapter.

`makeBold` always makes both `wantBig` and `wantRed` to be `True`, thus ensuring that every time this event handler is called, the text will be big and red.

`makeBig` inverts the value of `wantBig`, so that if the text was big before, it will become normal size; if it was normal before, it will become big.

`makeRed` works in the same way as `wantBig`, inverting the value of `wantRed` and toggling between red and black.

```
card Styling
    ...
    event makeBold (ClickEvent e)
        wantBig <- True
        wantRed <- True

    event makeBig (ClickEvent e)
        wantBig <- !wantBig

    event makeRed (ClickEvent e)
        wantRed <- !wantRed
```

Commentary

4.1c Styling

It is hard to overstate the importance of styling in web design. Even the simplest HTML template can be transformed by the appropriate styling. While this example is overly simplistic, we will build towards examples where, even with handwritten HTML and CSS, it is possible to see that bare HTML can be transformed through styling.

It is vitally important, then, that FLAS supports the full power of CSS, but without the overhead of managing the complexity. It does this by repeatedly and consistently applying the techniques of *loose coupling*, *separation of concerns*, *abstraction* and *avoidance of repetition*,

The main abstraction in styling is the notion of the *class*. Many things which are possible in CSS are not supported by FLAS - styling on absolute elements defined by `id`, for example. The reasons for this are largely technical - in a card-based environment, it is not reasonable to assume that *any* `id` that you would specify would be unique - but there is a more important, philosophical, point here too: you do not want to be that tightly coupled to your styling and your HTML.

It might be argued that FLAS itself requires `id`s to be specified in order to obtain its coupling; but those `id`s are used during the translation process and *not* at runtime. The information is preserved as a `data` attribute, however, and can be used if necessary in styling; however, alternative techniques would generally be preferred.

During the ingesting and translation phase, the input HTML is scraped and repackaged (the CSS files are passed through and included untouched). While this has dramatic effects on some of the content, by and large, apart from removing `id`s from *all* elements, any elements which do not have a `id` beginning `flas-` will be passed through untouched. This means that you can wrap any FLAS `content` or `style` elements in an HTML element which you can label and style as you wish.

Essentially, styling *in* FLAS comes down to associating a list of styles with each element under its control - including the card itself. This list of styles corresponds to a list of class names in CSS. From there, all of the styling is handled in the usual way.

Each styling definition is introduced by the condition symbol (`|`), followed by an optional condition (which may be any expression), and then, following the sendto operator (`=>`), it may have multiple styles, which may be strings, lists of strings, or any expression that evaluates to one of these.

Styles may be applied directly to the card (by indenting them one tab stop under `template`), to any `content` or `style` binding (as with `display` in this example) or to a parent style; that is, styling definitions in FLAS may be nested. A nested styling definition will only apply if both the parent and nested conditions apply.

4.2c Event Targets

When we used the click event to toggle the message in the last chapter, we allowed a click anywhere on the card. Given that the message was the only thing in the card, this made no difference to specifically requesting the event be on the message. But to make this example - with three separate buttons *and* a message - work, it is necessary to more specifically identify the targets when the events apply.

As with many other aspects of FLAS, we apply the *loose coupling* and *avoidance of repetition* techniques. Event handlers are defined in one place on the card, are identified by the `event` keyword, and are linked to a specific event by the (exactly one) parameter that they take. If that event handler is not mentioned on *any* template, it is assumed to apply to the whole card. Otherwise, it applies to exactly the elements which define it.

Event target definitions are much simpler than styling definitions, although they may appear in exactly the same set of places. An event definition consists of the sendto operator (`=>`) followed by the name of the event handler.

Because event handlers may be nested within style definitions, it follows that they may also be conditional; that is, the event handler will only be bound to the element if the style condition is true.

5 Data

Data is at the heart of complex programs. To better support the *inherent* complexity of programs, FLAS attempts to simplify data handling by offering varied and precise data constructs (and builtin data types) that avoid developers having to reinvent the wheel and continually write boilerplate code. Furthermore, FLAS attempts to simplify development by using *type inference* where possible to give all the advantages of both strong and dynamic typing.

FLAS draws on a number of different underlying technologies, but core to its essence is *functional* programming. Because functional programs do not concern themselves with mutable state, but rather with *values*, functional languages typically have highly expressive mechanisms for describing data.

FLAS offers a number of such structures, but gives them a (possibly) more familiar terminology and syntax by describing the two most fundamental data structures as `struct` and `union`. A `struct` describes a set of values constituting the cross-product of a set of named, typed fields. A `union` represents the set of all values contained in a named set of `struct` or `union` types. The builtin type `Any` represents the set of all values.

In common with most functional languages, FLAS then allows these types to be broken apart by functions through a mechanism known as *pattern matching*[1] This allows a function to be specified in parts, each part describing how the function intends to handle *some part* of the complete set of input values. The FLAS compiler analyzes these declarations and identifies the overlaps and commonalities and comes up with a set of logic which traverses all the data structures to execute the correct portion of the function.

Furthermore, the FLAS compiler supports a similar technique for handling templates and allows values to be "placed" in content slots in a template, and then goes on to find the appropriate sub-template to use to render the value.

---

[1] This is very different from the sort of pattern matching which uses regular expressions and should not be confused with it. If you are likely to be confused, pretend you have never heard the term *pattern matching* before and read on.

This example is going to show something which can often be hard in programming languages and UI toolkits but is nevertheless quite a common requirement - to show a clickable tree with different content elements, each of which displays appropriately.

Let's consider a company that wants to keep track of its worldwide assets. It has some notion of "Organizational Unit" which can nest other organization units recursively. We're going to call this a "Group" within the company and have a `struct` to represent that. Along with sub-groups, each `Group` can have a number of locations, so the contents of a `Group` are a list of a `union` we will call `Item` which can either be a `Group` or a `Location`. Finally, each `Location` has a list of associated assets.

5.1 HTML

So, what we need to do is describe how to render not just the overall card - which contains the single top-level `Group` but also each of `Group`, `Location` and `Asset`; four templates in all. But in line with the "bring your own HTML" model (in which the user delivers HTML files each of which looks like it *could* be a page from the web application), we don't expect users to provide four separate HTML files but the FLAS compiler is expected to tease apart the relevant definitions from the single file provided by the user.

The top level of the HTML looks like this:

```
<html>
<head>
    <link rel='stylesheet' type='text/css' href='polytree.css'
                                >
</head>
<body>
    <div id='flas-card-top'>
        <div id='flas-container-entries' class='top-table'
                                >
            ...
        </div>
    </div>
</body>
</html>
```


Aside from the boilerplate that we have seen before, this says that the top level of the card is a `container`. In FLAS, `containers` have a number of different uses, but essentially this says that *something* is going to go here. In ingesting and tranforming this, the compiler "throws away" the contents of a container and just leaves this outer `div` in the card.

Of course, it doesn't completely throw away the contents: it scans them first for any nested template definitions.

```
    <div id='flas-container-entries' class='top-table'
                                >
        <div id='flas-item-group-node' class='group'>
            <span id='flas-content-expander'
                  class='expansion-control'>▲</span>
            <div id='flas-content-name'
                 class='group-name'>Global</div>
            <div id='flas-container-items'>
                    ...
            </div>
        </div>
    </div>
```

In this case, there is only one element within the container, although it is perfectly reasonable to contain many. And the one that is here is going to be "preserved": the `id` `flas-item-group-node` says that this element is an *item template* and can be used to generate the HTML for *item*s that are going to be placed in a container.

The item has three subelements: an expansion control (`flas-content-expander`), a display name (`flas-content-name`) and a `container` for any nested trees (`flas-container-items`). All three of these are tagged with FLAS-specific `id`s, so they will correspond to fields in the template. As with the `container` above, during processing `flas-container-items` will be cleared and its nested contents processed and ingested as applicable.

In this case, the container nests a location.

```
    <div id='flas-container-items'>
        <div id='flas-item-location-node'>
            <span id='flas-content-expander'
                  class='expansion-control'>▲</span>
            <div id='flas-content-location'
```

```
                class='location-name'>Asia</div>
            <div id='flas-container-assets'>
                            ...
            </div>
        </div>
    </div>
```

This looks remarkably similar to the previous `item` definition. Indeed, in this case, it would not be an unreasonable choice to reuse the `item` template above to show locations as well. But it will generally be the case that all of the items will be subtly different and reuse of templates for different data types will cause confusion if not outright complexity. If you choose to represent your data with different data types, it makes sense to have different templates too.

Again, this template "ends" when it reaches the `container` for `flas-container-assets`, but not before scanning it to see if there are any more nested templates.

```
    <div id='flas-container-assets'>
        <div id='flas-item-show-asset' class='asset-row'
                            >
            <div id='flas-content-what'
                class='asset-label'>Offices</div>
            <div id='flas-content-value'
                class='asset-value'>5000</div>
        </div>
    </div>
```

This template shows the label and value of the asset.

Altogether, this HTML defines four templates (one for the card, one for a `Group`, one for a `Location` and one for an `Asset`). In the HTML itself, these are nested one inside the other so that they appear to be a single web page. But during the ingestion phase, they are untangled and four separate templates are recorded and ready to use.

5.2 CSS

The CSS for this example is a little larger than previously, but nothing particularly stellar.

```
.top-table {
padding: 5px;
}
.group {
margin: 3px 5px 3px 10px;
}
.expansion-control {
width: 1.5em;
font-weight: bold;
}
.group-name {
font-weight: bold;
display: inline-block;
}
.group-table {
padding: 5px;
}
.group-table.contracted {
display: none;
}
.location-name {
display: inline-block;
color: green;
margin-bottom: 5px;
}
.asset-label {
margin: 0px 15px 0px 20px;
width: 80px;
overflow: hidden;
}
.asset-label, .asset-value {
display: inline-block;
}
.asset-table {
padding: 5px;
}
.asset-table.contracted {
display: none;
}
```

Probably of most significance are the `.group-table.contracted` and `.asset-table.contracted` styles which specify how the contracted elements are elided from the display: they are hidden by setting their CSS `display` style to `none`.

5.3 FLAS

FLAS allows developers to group their code units largely as they please. As with all programming languages, good style and readability generally suggest using separate files for significant definitions. However, data type definitions in FLAS are very short - typically 5-10 lines - so it is often common to group them. In this case, with only one complex definition in the codebase, everything has been grouped in one file, `polytree.fl`.

Data Types

There are three `struct` definitions, corresponding to the three basic concepts in our model: `Group`, `Location` and `Asset`. Hopefully these definitions should be fairly self explanatory.

```
struct Group
    Boolean expanded
    String name
    List[Item] items

struct Location
    Boolean expanded
    String location
    List[Asset] assets

struct Asset
    String what
    Number value
```

Each of `Group` and `Location` contains a `List` of nested elements. The definition specifies a *generic* type by enclosing the element type in brackets (`[...]`).

The list included in `Group` is a list of `Items`. An `Item` is defined as *either* a nested `Group` or a `Location` using a `union`:


```
union Item
    Group
    Location
```

Note that since a union is simply defining a set of values including all the values of all its element types, there is no need to provide a field name on any of these definitions; the type is enough. Specific values must be extracted from a union using functional pattern matching.

Card Outline

The card consists of three parts: a `state` definition, four `template` definitions and four `event` handlers.

```
card PolyTree
    state
        ...
    template top
        ...
    template group-node <- (Group g)
        ...
    template location-node <- (Location l)
        ...
    template show-asset
        ...
    event contractGroup (ClickEvent ev)
        ...
    event expandGroup (ClickEvent ev)
        ...
    event contractLocation (ClickEvent ev)
        ...
    event expandLocation (ClickEvent ev)
        ...
```

Each template has a name which corresponds to one of the template definitions in the HTML file. It is an error for a template to be used in FLAS which has not been defined in the HTML, or to use a template more than once in the same context.

State

The state of this card consists of just one variable and should be short. The fact that it is not comes down to the fact that being sample code, it promptly loads all the data into the state rather than fetching it from more persistent storage - such as a database or web service.

```
card PolyTree
    state
        List[Item] items <- [
            Group False "company" [
              Group False "sales" [
                Location False "USA" [
                  Asset "Offices" 500000,
                  Asset "Cars" 200000,
                  Asset "Inventory" 109000
                ],
                Location False "Europe" [
                  Asset "Coffee" 20000,
                  Asset "Magazines" 15000
                ]
              ],
              Group False "technology" [
                Group False "new" [
                  Location False "Palo Alto" [
                  ]
                ],
                Group False "current" [
                ],
                Group False "legacy" [
                ]
              ],
              Group False "admin" [
                Location False "New York" [
                ]
              ]
            ]
        ]

    ...
```


This creates a "list" of exactly one item, the top-level Organizational Unit or `Group`. Although the code supports a list of items, only one is provided because that matches my understanding of how organizations (should) work. But the code would be perfectly happy to handle a conglomerate with no central management.

Templates

The card defines a top-level template which uses the template defined by the HTML element `flas-card-top`. This has a single container `flas-container-entries`, for which there is a *binding*. This binding says that the container will be filled with the elements of the list `items`.

```
card PolyTree
    ...
    template top
        entries <- items

    ...
```

How does this work? FLAS knows that `entries` is a container and that `items` is a list. It renders each element in the list and places the result into the container. How does it know how to render the items in the list? It looks at their type and then tries to find a "corresponding" definition among the card's templates.

```
card PolyTree
    ...
    template group-node <- (Group g)
        expander
            | g.expanded <- '▼'
                => contractGroup
              <- '▲'
                => expandGroup
        name <- g.name
        items <- g.items
            | => 'group-table'
            | !g.expanded => 'contracted'

    template location-node <- (Location l)
        expander
            | l.expanded <- '▼'
```

```
                => contractLocation
            <- '▲'
                => expandLocation
        location <- 1.location
        assets <- 1.assets => show-asset
            | => 'asset-table'
            | !1.expanded => 'contracted'

    ...
```

There is an additional complication. The elements of the `items` list are all members of a union, so it is not clear *exactly* what type they are; they could be nested `Group` elements or they could be `Location` elements.

FLAS resolves this confusion by allowing the user to define one template for each member of the `union`, so long as they are precise about *which* type is intended to be covered by the case. FLAS ensures that all the cases have been covered and that there is no ambiguity.

As noted when discussing the HTML, these two cases are very similar; the only real difference being the contained lists. In each case, there is an expander which is a visual clue as to whether the node is expanded or contracted and is set based on the `expanded` member of the `struct`.

Clicking on the expander will invoke the appropriate method to contract or expand the group.

`expander` is bound *conditionally*; that is, there are two cases (more may be specified) which are considered in turn until one of the conditions evaluates to `True`. In this case, the second condition is "default" which means that it will *always* be `True`. For any given binding, only one default is allowed and it must come last.

The label (`name` or `location`) is a simple, non-conditional binding.

Finally, the nested list is shown. In the `Group` case it is much the same as with the top-level card: it puts a list of items in the `items` container. But the `Location` case is different. Because the `Location` knows exactly what type its elements are, it can specifically name the template it wants to use, in this case `show-asset`.

Finally, each of the lists is styled. In every case, the list is given the `group-table` or `asset-table` style, and then, if the `expanded` flag is not set, it is also given the `contracted` style. As we saw above, this causes the CSS to not display the table giving it the "contracted" feel.

The final template shows the name and value of an asset.

```
card PolyTree
    ...
    template show-asset
        what <- what
        value <- (show value)


    ...
```

This is relatively simple compared to the two that have gone before. Note the use of the fields `what` and `value` which seem to appear randomly. In fact, their type is inferred from the earlier use of the template - they are fields of the inferred `struct Asset`.

Event Handlers

Finally, we have the definitions of the event handlers. These were bound in the templates to enable the expanders to be clicked to expand and contract the branches of the tree.

```
card PolyTree
    ...
    event contractGroup (ClickEvent ev)
        ev.source.expanded <- False
    event expandGroup (ClickEvent ev)
        ev.source.expanded <- True
    event contractLocation (ClickEvent ev)
        ev.source.expanded <- False
    event expandLocation (ClickEvent ev)
        ev.source.expanded <- True
```

These four event handlers are incredibly similar, and could in fact be replaced by a single one which inverted the value of `ev.source.expanded`. They have been listed separately to increase clarity in this example. Again, your experience will vary with the tradeoffs you have to make in any given situation. In every case, `ev.source` refers back to the `struct` used to render the target area that was clicked. A *target area*, in this sense means the smallest block rendered by a template in

The upshot of course, is that any time one of these event handlers is fired, its consequence is to invert the current value of the appropriate

---

Commentary

5.1c Card and Item Templates

It may seem that having separate templates for cards and items is an unnecessary complexity; sadly, it is an essential complexity.

In popular parlance, *card* is shorthand for "a small area of the screen that looks separate". In FLAS, they are fundamental building blocks - actors. Meanwhile, the need to manage "a small area of the screen" by itself having not gone away, there is a need for a concept to handle it - the *item*.

It is possible to nest cards within cards, but in doing so it is necessary to introduce new actors into the system which, by definition, must be independent of their parents. There are significant complexities in doing this to which we will return ina later chapter. For now, it is simply necessary to note that the *first* template referenced in the card definition must be a `card` template; *all* the others must be `item` templates.

5.2c Naming Templates and Handlers

The rules for naming elements in FLAS are somewhat complicated because of the number of different models that are coming together in one place. The rules are discussed thoroughly in the reference manual.

In essence, function, constant and field names in FLAS must start with a lowercase letter and then contain letters, numbers and underscores. Type names must start with an uppercase letter, be at least three characters long, and contain letters, numbers and underscores. By convention, camelcase is used in both types of name to indicate the start of a word.

Event handlers are considered functions and so follow this same rule.

Template elements are different, though. Because template names are *defined* in HTML, they follow the HTML naming convention of lowercase letters with words separated by hyphens. This obviously applies to both specifying a template and those occasions when templates are referenced.

5.3c The Template Nesting Chain

A card template needs to be "context free"; that is, it needs to be able to be rendered with no information other than what can be found in the state of the card.

This is not true for nested items: each item template is rendered in a specific context, depending on where it is referenced; it is the referencing site that defines the context.

Once again, the rules by which this works are quite complex, but the important point is that it is possible to "inherit" context variables in a template as well as to acquire the value of a loop variable. Additional context variables are just listed after the loop variable (if any).

Context variables are, by default, matched by type, so only one such variable can be passed for each type. In this sense the mechanism used by `location-node`, of explicitly specifying the template to use is more general, because it allows different templates to be used for the same item type.

More complex situations may also arise, in which case, specific naming can be used to address them, but that is beyond the scope of this guide.

When the `location-node` template specifically references `show-asset`, this ensures that the loop-element variable in `show-asset` is defined to match the list element type (in this case `Asset`). Because the template does not explicitly set a template nesting chain, it is automatically populated. If `show-asset` is referenced on more than one occasion, all of the references will be checked to ensure that they all specify the same type.

If the element type is specified, the element must be given a variable name and this must be used to dereference the fields in the template. If the element type is not specified, then the field members can be used directly.

5.4c Event Sources and Traits

A `ClickEvent` is defined as a `struct` in a standard way. However, it is incredibly useful to be able to find the element which was the "target" for the event. This can be accessed by referencing the `source` parameter of the event.

But what type is it? Well, that's tricky and TBH, it's hacked at the moment, but I have a plan to define "traits" to sort this out. What you want is for it to be the same as the "element type" that was used to render the template.

A trait is kind of like an extra thing that gets tacked on to the struct at compile/reference time that knows because of the context what's going on.

So you can reference your data item's members as if `ev.source` just were that type. Cool, right?

6 Objects and the Lifecycle Contract

So far, we have mainly been discussing how FLAS works using Actors, although we have also seen examples of data types, both primitives such as `String` and the more complex data structures introduced in the last chapter.

But there is a need for something in between - an *object* that can combine data and behavior while not providing an independent lifecycle and maintaining strict functional semantics.

It's the last requirement that makes FLAS objects "feel" very different from objects in traditional object-oriented languages - and in this context, even JavaScript might be considered "traditional".

Consider then, a list of items such as is commonly found in websites: say a list of message headers in a mail reader; a list of servers in a cluster; or a list of posts in a forum. Each of these has some data, together with some controls, such as a "select" control and a "favorite" control. There are also some operations - such as delete - which can either operate on individual items or on all the items that are "currently selected".

We are going to build this using a single card encapsulating everything you see here with each item represented by a separate instance of a single object. These objects can be created or deleted at will

6.1 HTML

As part of the abstraction offered by an object, it knows how to render itself - possibly in multiple ways - using the same template-based mechanism used by the card itself. Thus, in addition to an HTML template for the overall card, we need an HTML template for the object. But in line with the "bring your own HTML" model (in which the user delivers HTML which makes sense by itself), we don't expect users to provide two separate HTML files; rather, the user provides a single file which contains one template nested within the other: it is up to FLAS to figure it out.

At the top level, the HTML looks like this:

Note the element with `id` `flas-container-list`. When the compiler ingests this, it will remove all its contents, and then this will be the "container" into which the object items will be put.

However, the ingestor looks inside the container, searching for other templates that it might find. And it finds this one with `id` `flas-item-header`. This identifies the element as a template, similar, but not identical to a `card` template. The difference is simple: it can be used to render complex content *within* a card, but cannot be used for a card itself. This is the template that we will use for the object.

Note that the original HTML may include more headers here, which look like they are the same as the template. This allows the input HTML to contain multiple versions of the header line and to be reviewed as such; but only the first header has the magic `id` which tells the compiler to ingest it and use it as a template - all the others are simply thrown away.

6.2 CSS

6.3 FLAS

Now we need to actually write some code

Headers

This is the cardNeeds a lifecycle handler to create the objectNeeds the event handler for "delete selected"Needs logic to handle 'remove an item' from the list

SingleHeader

This is the objectNeeds event handler for 'select' and 'favorite'Needs to handle 'delete'

7 Bringing It All Together

Having worked through all the building blocks of the basic FLAS system[1] it is now time to put them all together in an application of reasonable complexity.

We are going to implement a game of Patience (or Solitaire as it is often called). While the rules often vary, since we are mainly focused on the didactic task of explaining *how* to build a web application, not *what* to build, we have chosen rules that are fairly simple to implement. It is left as an exercise to the reader to improve and adapt the work that we have done. Feel free to submit branch pull requests with your great ideas.

In short, the game we are going to design has the following properties:

- It is played with a standard deck of 52 playing cards which are shuffled before play starts;
- 28 of the cards are dealt out in a right-justified triangle with 7 columns (called the "tableau");
- The leftmost card in each column is dealt face up; the others are dealt face down;
- Each row of cards overlaps the row above;
- The remaining cards are placed face down in the top left corner (the "deck");
- There are five (as yet empty) piles:
  - A "discard" pile where cards are placed when turned over from the deck;
  - 4 piles (one for each suit) where the player attempts to collect the cards in rank order (from A to K) as they become available;
- On each turn a user may:
  - Move any uncovered, face up Ace to start the pile for that suit;
  - Move any uncovered, face up card onto the pile for its suit providing it is the rank above the top of that pile;

---

[1] There are, in fact, more building blocks in FLAS but using them effectively requires a Ziniki back end and that is beyond the scope of this document. For more information, see the Ziniki Programming Guide and the Ziniki Reference.

    - Move any entire "run" of face up cards in a single column to the bottom of another column provided that the rank of the top card is one less than the rank of the bottom card in the destination run AND that these two cards are of opposite colors;
    - Move the current card at the top of the discard pile into the tableau obeying the same rule above for moving cards within the tableau;
    - Turn over the top card in the deck and place it on the discard pile;
    - If the deck is empty, the discard pile may be turned over to make a new deck;
- The game ends when either all the cards are in the ranked piles (victory) or no further cards can be moved, except for endless dealing of the deck into the discard pile (defeat).

Hopefully, these rules are not too controversial for dedicated players[2] out there. One thing that hopefully is clear at this point is that although this is a relatively simple game, it is most certainly *not* a trivial one. There is quite a lot of what is often called "business logic" here, as well as all the mechanics of keeping track of the current game state, displaying it and reacting to user interactions.

We feel the need to apologize right now for any confusion that may arise between the use of the word "Card" to describe an organizational concept in FLAS and the word "card" to describe an element of this game. While unintentional, it does reflect the fact that the terminology of FLAS comes from the physical realm where you can imagine building a UI from a tessellation of index cards of various sizes.

Unlike previous samples, where we have presented all of the HTML followed by the FLAS code, this code is sufficiently large that it makes sense to present it in a more modular form.

[2] On the other hand, hopefully the description we have given is not too confusing for those unfamiliar with the game. For those who are, we would suggest that you head to `samples.zinapps.org/patience` and give it a few rounds to understand how the game is actually played, which will hopefully clarify the rules.

7.1 Outline HTML

```
<html>
<head>
    <link rel='stylesheet' type='text/css' href='patience.cs
                                        >
</head>
<body>
    <div id='flas-card-patience'>
        <div class='toprow'>
            ...
        </div>
        <div id='flas-content-tableau' class='tableau'
                                    >
            <div id='flas-item-layout'>
                <div id='flas-container-rows'></div>
            </div>
        </div>
    </div>
    ...
</body>
</html>
```

then present the main card and Lifecycle initializer together with its constructor and repeater, although I think that is in Deck, and then present Deck in its entirety ….