A5 concepts

- abbreviated addresses
- walking the commit parent chain
- Clojure deps
- Web servers in Clojure
- HTML and Hiccup

(See A5 writeup for detailed notes and code snippets.)
Interfaces

interface (noun):

. A surface forming a common boundary between adjacent regions, bodies, substances, or phases.

. A point at which independent systems or diverse groups interact.

. A system of interaction or communication between a computer and another entity such as a printer, another computer, a network, or a human user.
Interfaces

Examples

The interface between the ocean and the air.

An application programming interface, or API: you write a program that interacts with an existing program through an API, e.g. GitHub's API.

Interaction modes for computers and humans:

- graphical user interface (GUI)
- command line interface (CLI)

I tend to like keyboard interfaces, which is why I like:
Interfaces

Why should we care?

- Remember our limited ability to hold multiple things in our attention?

  Our puny minds are helped by considering a thing apart from its details

  More specifically: interfaces allow us to separate *specification* from *implementation*

    - Specification: *what* the thing does
    - Implementation: *how* it does it Interfaces are one way to achieve polymorphism (which we'll talk about)

  Interfaces decouple things and enable composability
Interfaces

Traits

Interfaces describe a set of related things you can do (i.e. *verbs*) to some noun

A decent word for this idea is "trait". Some nouns have a given trait (i.e. "implement the interface"); others don't.

Traits in nature

- Can grasp (distinguishes apes, monkeys, and humans)

- Can move oneself (distinguishes fauna from flora)

- Can see/can hear (distinguishes the more sophisticated animals) Notice that we can talk about the trait by itself, disconnected from a noun.


Interfaces

Constraints and freedom

From the noun's perspective, an interface is a constraint: it submits to the rules of the interface. The more interfaces it participates in, the more it has to do.

Point of philosophy: what does ultimate freedom look like?

- It's awfully hard to get anything done when you refuse to submit to anything. A world without friction is difficult to deal with.
Interfaces

Interpersonal examples

People (and companies) use interfaces to deal with people

Are you a student? You implement a student interface. You must:

- `(attend-class a-class)`
- `(do-assignment an-assignment)`
- `(take-test a-test)`
- `(pay-tuition a-big-sum)`

As a professor, I implement an interface too. I must teach stuff, create assignments, create tests, and manage the LAs.
Interfaces

Towards polymorphism

Foreshadowing: these constraints enable us to deal with variation in a consistent way.

All of you are unique, probably with some remarkable differences of skills, preferences, and opinion.

If I had to consider the full complexity and variation of each of you to do my job, I wouldn't be able to! Likewise, you with your various professors.

But we have an interface that lets us get things done despite our differences. This is close to the essence of polymorphism, as we'll see.
Interfaces

Design considerations

Fine-grained interfaces enable flexibility, like ordering *a la carte* from a menu.

Tradeoff: smaller interfaces is (slightly) more work for the implementer (but not much)

Recommendation: go as fine-grained as possible, but ensure each trait actually *means* something.

Difficulty: breaking constraints down into meaningful parts is hard mental work. E.g. how would you break down the student interface into parts?

Thankfully, Rich Hickey (Clojure creator) has gone ahead of us here...
Interfaces

An example in Clojure

From the Clojure source code:

```java
public interface Associative extends IPersistentCollection, ILookup{
  boolean containsKey(Object key);
  IMapEntry entryAt(Object key);
  Associative assoc(Object key, Object val);
}
```

Notice that these verbs are related, as part of the trait described by the word `Associative`. 
Interfaces

The **Associative** interface

Things that implement **Associative** in Clojure:

- IPersistentMap (i.e. an immutable hashmap)
  - (There are actually lots of kinds of hashmaps in Clojure: PersistentStructMap, PersistentArrayMap, PersistentTreeMap, PersistentHashMap.)
- IPersistentVector (i.e. an immutable vector) Did you know you can `assoc` with a vector?

```
(assoc [3 6 7] 1 5) ; => [3 5 7]
```
Interfaces

Another example: \texttt{ISeq}

```java
public interface ISeq extends IPersistentCollection {
    Object first();
    ISeq next();
    ISeq more();
    ISeq cons(Object o);
}
```

This is an interface for sequential things.

Implemented by lazy sequences and most collection types (lists, vectors, hashmaps, sets)
Interfaces

Another example: **IFn**

- Just one verb (though many arities): **invoke**

  Implemented by lots of things
  - functions (of course)
  - keywords (e.g. `(age kiddo)` or `(map :age [johnny susan billy])`)
  - sets (e.g. `#{"-h" "--help"}` next-argument), returns `nil` if no match or the match if there was one)
  - vectors (e.g. `[[12 15 18] 2]` returns the item at index 2, i.e. `18`)
Interfaces

Another example: IFn (continued)

aps also implement IFn

(def error-type->message
  {:invalid-argument "Error: invalid argument"
   :missing-database "Error: no database found. Did you `init` one?"
   ,,,})
**Interfaces**

Defining your own interfaces

Here's a simple example:

```lisp
(defprotocol Geometry
  "Basic geometrical calculations on 2d shapes"
  (area [_])
  (perimeter [_]))
```

This defines the functions for `area` and `type` in this namespace that dispatch based on type.

Remember that this is abstract. It needs to be connected to a noun to actually do anything.
Interfaces

Implementing an interface

```lisp
(defrecord Circle [radius]
  Geometry
  (area [this] (* Math/PI (:radius this) (:radius this))))
  (perimeter [this] (* 2 Math/PI (:radius this))))

(area (->Circle 3)) ; => 28.274333882308138
(perimeter (->Circle 3)) ; => 18.84955592153876
```

If this record was in a different namespace, we'd have to refer to Geometry to implement the protocol and area and perimeter to invoke the functions.
Implementing an interface (2)

So you can extend types to implement a protocol after the type is already defined.

```
(defrecord Rect [height width])

;; => No implementation of method: :area of protocol: #'Geometry found

(extend-type Rect
  Geometry
  (area [this] (* (:height this) (:width this))))

(perimeter [this] (* 2 (+ (:height this) (:width this)))))

(area (->Rect 3 5)) ; => 15

(perimeter (->Rect 3 5)) ; => 16
```
Interfaces

Implementing an interface (3)

```
(area
 (let [square-size 5]
  (reify Geometry
    (area [_] (* square-size square-size))
    (perimeter [_] (* 4 square-size)))))
;; => 25
```

You can even provide anonymous, inline implementations of interfaces
Interfaces

Open for extension

I was using some unusual data type in my database one time.

I was using a Clojure adapter for the database, but it didn't support the data type I was using.

I was able to supply my own implementation for that data type; problem solved.

No forking the adapter's source code. No pull request. Just extend the protocol to the other type and be done.
Polymorphism

Concepts

Etymology: from "polymorph", meaning "many forms"

A polymorph describes a set of things that are similar and different.

Alternately, it describes various concrete expressions of the same abstract idea, each with a different shape.
Polymorphism

Polymorphism and interfaces

Interfaces are a *mechanism* to achieve polymorphism.

An interface specifies exactly how the various forms are the same.

They can be arbitrarily different in other ways. Remember: the interface is the constraint, and I don't need to care what you do in your spare time.
Polymorphism

An example

Example: git objects are polymorphic: they have some shared identity as a git object, but blobs, trees, and commits have different forms.

What's different between these forms? The relevant data needing to be stored.

What's the same? What is the constraint?

Why is the constraint important?
Polymorphism

Multimethods

We saw protocols already as a mechanism for polymorphism in Clojure.

Another one exists: multimethods.

Multimethods separate the questions:

- What type of thing(s) are we dealing with here?
- How do we handle that type of thing(s)?

Protocols only dispatch based on class; multimethods can dispatched based on anything; in fact, you write the dispatch function yourself.
Polymorphism

Multimethods example

```
(defmulti mm-area
  "Compute the area for a shape"
  (fn [shape] (:type shape)))

(defmethod mm-area :circle [circle]
  (* Math/PI (:radius circle) (:radius circle)))
(mm-area {:type :circle, :radius 3}) ; => 28.274333882308138

(defmethod mm-area :rect [rect]
  (* (:height rect) (:width rect)))
(mm-area {:type :rect, :height 3, :width 5}) ; => 15
```
Polymorphism

Multimethods example

```clojure
(mm-area {:type :square, :width 5})
;; => No method in multimethod 'mm-area' for dispatch value: :square

(defmethod mm-area :default [something] nil)
(mm-area {:type :square, :width 5}) ; => nil

(defmethod mm-area :square [square] (* (:width square) (:width square)))
(mm-area {:type :square, :width 5}) ; => 25
```

So multimethods can have a default "catch-all" method

And you can implement methods after the multimethod has been defined