( life w/o objects )
( oop → functional design )
( with clojure )

(Osvaldas Grigas | @ogrugas)
"hybrid" languages
\[
\frac{x : \sigma \in \Gamma}{\Gamma \vdash_D x : \sigma} \quad \text{[Var]}
\]

\[
\frac{\Gamma \vdash_D e_0 : \tau \to \tau' \quad \Gamma \vdash_D e_1 : \tau}{\Gamma \vdash_D e_0 \ e_1 : \tau'} \quad \text{[App]}
\]

\[
\frac{\Gamma, \ x : \tau \vdash_D e : \tau'}{\Gamma \vdash_D \lambda \ x . \ e : \tau \to \tau'} \quad \text{[Abs]}
\]

\[
\frac{\Gamma \vdash_D e_0 : \sigma \quad \Gamma, \ x : \sigma \vdash_D e_1 : \tau}{\Gamma \vdash_D \text{let } x = e_0 \ \text{in} \ e_1 : \tau} \quad \text{[Let]}
\]

\[
\frac{\Gamma \vdash_D e : \sigma' \quad \sigma' \subseteq \sigma}{\Gamma \vdash_D e : \sigma} \quad \text{[Inst]}
\]

\[
\frac{\Gamma \vdash_D e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash_D e : \forall \alpha . \ sigma} \quad \text{[Gen]}
\]
Say "Monad" one more time!
(design)
(monolithic design)
(modular design)
(design process)

Before

After
(oo design)

- Noun-oriented
- Verb-oriented
(nouns)

- CustomerDAO
- CustomerService
- CustomerController
(verbs)

- RegisterCustomer
- PromoteCustomerToVIP
- RenderCustomerProfilePage
Domain-driven design?
(oo design principles)

- Single Responsibility
- Interface Segregation
- Dependency Inversion
(single responsibility)
public class ZipDownloadService {

    List<File> downloadAndExtract(String location) {
        ...
    }
}
public class FileDownloader {
    List<File> downloadFiles(String location) {
        ...
    }
}

class ZipExtractor {
    File extractZip(File archive) {
        ...
    }
}
Or ... just functions

```(defn download-files [location] (...))```

```(defn extract-zip [archive] (...))```
(clojure)
in a nutshell
(clojure)
in a nutshell

calculate(5, 2, 9)
(clojure)
in a nutshell

(calculate 5, 2, 9)
(clojure)
in a nutshell

(calculate 5 2 9)
(function definition)

(defn calculate [a b c]
  (+ a (* b c)))
(interface segregation)
public class ProductCatalog{
    public void Save(Product product)
    {
        ...
    }

    public Product FindById(int productId)
    {
        ...
    }
}
public class ProductSaver {
    public void Save(Product product) {
        ...
    }
}

public class ProductFinder {
    public Product FindById(int id) {
        ...
    }
}
Somethin' ain't right
public class ProductRepository
{
    public void Save(Product product)
    {
        ...
    }
}

public class ProductQuery
{
    public Product FindById(int id)
    {
        ...
    }
}
Feelin' good now
Or ... just functions

(defn save-product [product]
  (...))

(defn find-product-by-id [id]
  (...))
Applying OO design principles often leads to functional design
(what's missing)
(what's missing)

- Code organization
- Encapsulation
- Inheritance hierarchies
- Composition & Polymorphism
(code organization)
(ns my.product.repository)

(defn save [product]
  (...))

(defn find-by-id [id]
  (...))

(ns my.other.namespace
  (:require [my.product.repository :as product-repo]))

(product-repo/find-by-id 42)
(encapsulation)
(encapsulation)

- Data is *not* an object
Data is not an object
Data is immutable
(html [:body

  [:h1 "Employees"]

  [:ul

    (for [person employees]

      [:li

        [:h2 (:name person)]

        [:p (:email person)]])])
(defn handle [request]
{:status 200
 :headers {"Content-Type" "application/json"
  "X-Custom-Header" "12345"
}
 :body (json/generate-string
 {:name "John Wick"
  :age 42
  :email "john@wick.name"
  :hobbies ["guns" "dogs" "judo"]}}))
Process / Entity / State Machine
(inheritance hierarchies)
(inheritance hierarchies)

just don't
composition & polymorphism
(composition)
Avoiding hard-coded dependencies
public class ProfilePage {

    String render(Repository repo, int customerId) {
        return toHtml(repo.loadProfile(customerId));
    }
}

Repository repo = new Repository();
ProfilePage page = new ProfilePage();

String html = page.render(repo, customerId);
(defn render-page [repository-fn customer-id]
  (to-html (repository-fn customer-id)))

(defn load-from-db [customer-id]
  (...))

(render-page load-from-db customer-id)
(basic fp polymorphism)

All functions implement the "Strategy" pattern
ProfilePage page = new ProfilePage(new Repository());

page.render(customerId);
(defn inject [f arg1]
  (fn [arg2] (f arg1 arg2)))

(def render-from-db
  (inject render-page load-from-db))

(render-from-db customer-id)
(partial application)

(def render-from-db
  (partial render-page load-from-db))

(render-from-db customer-id)
"Object is a collection of partially-applied functions."

( J.B. Rainsberger )
("adapter" pattern)

(defn to-view-model [profile] (...))

(render-page (comp to-view-model load-from-db) id)
(polymorphism)
(oo polymorphism)

"Subtype Polymorphism"

(dispatch on the type of first argument)
public interface JsonObj {
    String toJson();
}

public class JsonString implements JsonObj {
    private final String value;

    public JsonString(String value) {
        this.value = value;
    }

    public String toJson() {
        return "\\" + value + "\\";
    }
}
public class JsonList implements JsonObj {
    private final List<JsonObj> items;

    public JsonString(JsonObj... items) {
        this.items = asList(items);
    }

    public String toJson() {
        return "[" + items.stream()
            .map(item -> item.toJson())
            .collect(joining("," )) + "]";
    }
}

JsonObj obj = new JsonList(
     new JsonString("a"),
     new JsonList(
         new JsonString("b"),
         new JsonString("c")
     ),
     new JsonString("d")
);

System.out.println(obj.toJson());

// ["a",["b","c"],"d"]
(limitations)

Subtyping is coupled to implementation
Subtyping is coupled to implementation...
... cannot extend existing types
(limitations)

Subtyping is coupled to implementation

... cannot extend existing types

... need wrapper classes
Too constraining!
(clojure protocols)
dispatch on the type of first argument
(defprotocol Json
  (to-json [this]))
(extend-type String Json
  (to-json [this]
    (str "\"" this "\"")))

(extend-type List Json
  (to-json [this]
    (str "[" (str/join "," (map to-json this)) "]")))

(extend-type nil Json
  (to-json [this]
    "null"))
(to-json ["a" ["b" "c"] nil "d"])

;;=> ["a", ["b", "c"], null, "d"]
Why stop there?
(multimethods)
dispatch on anything!
(defmulti greet :country)
(defmethod greet "PL" [person]
  (println "Dzień dobry," (:name person)))
(defmethod greet "FR" [person]
  (println "Bonjour," (:name person) "!"))
(defmethod greet :default [person]
  (println "Hello," (:name person)))

(greet {:name "Jacques" :country "FR"})

;;=> Bonjour, Jacques !
(the takeaway)

CLASS

POLYMORPHISM

NAMESPACE

HIERARCHY

DATA

STATE MANAGEMENT
(google "Clojure Made Simple")

(questions? "Osvaldas Grigas" @ogrigas)