# clojure.spec

# Observations / problems

- Docstrings help people but aren't useful for other programs or tests
- Programatic validation libs often complect keyset and attribute specification
- Generative tests are arguably more effective than hand written example based tests, but they require declaring properties which takes more effort than examples

# Solution goals

- Minimize intrusion
- Specify scalar values, data structures, and functions using a single global system
- Decomplect specs for map keys and values
- Leverage!

# Solution goals: leverage

- Documentation
- Validation
- Error reporting
- Destructuring

# Solution goals: leverage (cont'd)

- Instrumentation
- Test-data generation
- Generative test generation

# Predicative specs

We've already got many predicates built into the language.

```
(require '[clojure.spec :as s])

(s/valid? int? 3)
(s/valid? string? "abc")
(s/valid? nil? nil)
(s/valid? odd? 5)
(s/valid? inst? (Date.))                ;; new in 1.9!
(s/valid? boolean? false)               ;; new in 1.9!
```

# Predicative specs

Any function of 1 arg that returns a truthy value is a predicate.

```
(s/valid? #(= 3 (count %)) [1 2 3])
```

Sets, functions of their members, are predicates.

```
(s/valid? #{:red :blue :green} :red)  ;; enum
```

# Conformance

```
(s/conform int? 37)    ;; 37
(s/conform int? :foo) ;; :clojure.spec/invalid
```

# Registry

Use `s/def` to associate a namespaced keyword with a spec

```
(s/def :card/suit #{:Spades :Diamonds :Clubs :Hearts})
(s/valid? :card/suit :Hearts) ;; true
(s/conform :card/suit :Clubs) ;; :Clubs
```

# Nils

Most predicates do not allow for `nil`

```
(s/valid? string? nil)
;; false

(s/valid? (s/nilable string?) nil)
;; true
```

# Ranges

```
(s/def :bowling/roll (s/int-in 0 11))

(s/def :my/foo (s/double-in :min 3.2
                            :max 5.7
                            :infinity? false
                            :NaN? false))

(s/def :time/aughts (s/inst-in #inst "2000" #inst "2010"))
```

# Predicate conjunction (and)

All and'ed predicates must pass, will short-circuit

```
(s/def ::big-even (s/and int? even? #(> % 1000)))

(def email-regex #"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,63}$")
(s/and string? #(re-matches email-regex %))
```

# Predicate disjunction (or)

All parts of an `or` are tagged

```
(s/def ::name-or-id (s/or :name string?
                          :id   int?))

(s/valid? ::name-or-id "abc")  ;; true
(s/valid? ::name-or-id 100)    ;; true
(s/valid? ::name-or-id :foo)   ;; false

(s/conform ::name-or-id "abc") ;; [:name "abc"]
(s/conform ::name-or-id 100)   ;; [:id 100]
(s/conform ::name-or-id :foo)  ;; :clojure.spec/invalid
```

# Entity maps (keys)

- Specified in terms of registered attribute keys
- `:req` and `:opt` specify required and optional keys
- `:req` can specify logical combinations of keys with `and` and `or`
- `:req-un` and `:opt-un` specify unqualified keys
- All attributes checked for conformance regardless of whether they are listed!

# Entity maps (keys), continued

```
;; generic types
(def email-regex #"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,63}$")
(s/def ::email-type (s/and string? #(re-matches email-regex %)))

;; attributes
(s/def ::first-name string?)
(s/def ::last-name string?)
(s/def ::email ::email-type)

;; entity
(s/def ::person (s/keys :req [::first-name ::last-name ::email]
                        :opt [::phone]))
```

# Sequences (regex)

- Regex ops: `cat`, `alt`, `?`, `+`, `*`, `&`
  - `?`, `+`, and `*` do exactly what you think they do!
  - `cat` and `alt` tag their parts
  - `&` passes the result of a regex op and passes it through preds

# Sequences, cont (regex)

```
(s/def ::odds-then-maybe-even (s/cat :odds (s/+ odd?)
                                     :even (s/? even?)))

(s/conform ::odds-then-maybe-even [1 3 5 100])
;;=> {:odds [1 3 5], :even 100}

(s/conform ::odds-then-maybe-even [1])
;;=> {:odds [1]}
```

# Collections (every, coll-of)

```
(s/conform (s/every :name-or-id) ["abc" 100 "def"])
;; ["abc" 100 "def"]

(s/conform (s/coll-of :name-or-id) ["abc" 100 "def"])
;; [[:name "abc"] [:id 100] [:name "def"]]
```

- `s/every` *validates* `*coll-check-limit*` vals
- `s/coll-of` *conforms* every val

# Maps (every-kv, map-of)

```
(s/conform (s/every-kv keyword? ::name-or-id) {:a "abc" :b 100 :c "def"})
;; {:a "abc", :b 100, :c "def"}

(s/conform (s/map-of keyword? ::name-or-id) {:a "abc" :b 100 :c "def"})
;; {:a [:name "abc"], :b [:id 100], :c [:name "def"]}
```

- `s/every-kv` validates `*coll-check-limit*` k/v pairs
- `s/map-of` conforms every k/v pair

# Tuples (tuple)

Fixed length and positional

```
(s/def :card/rank (s/int-in 1 14))
(s/def :card/suit #{:Spades :Hearts :Clubs :Diamonds})
(s/def ::card (s/tuple :card/rank :card/suit))

(s/valid? ::card [3 :Hearts])
```

# Multi spec (multi-spec)

```clojure
(s/def :event/type keyword?)
(s/def :event/timestamp int?)
(s/def :search/url string?)
(s/def :error/message string?)
(s/def :error/code int?)

(defmulti event-type :event/type)  ;; arbitrary dispatch function
(defmethod event-type :event/search [_]
  (s/keys :req [:event/type :event/timestamp :search/url]))
(defmethod event-type :event/error [_]
  (s/keys :req [:event/type :event/timestamp :error/message :error/code]))

(s/def :event/event (s/multi-spec event-type :event/type))
```

# Error reporting

```
(s/def ::person (s/keys :req [:person/first-name :person/last-name :person/address]))
(s/def ::address (s/keys :req [:address/line-1 :address/zipcode] :opt [:address/line-2]))
(s/def :address/line-1 string?)
(s/def :address/line-2 string?)
(s/def :address/zipcode (s/and string? #(re-matches #"\d{5}" %)))

(s/def :person/address ::address)
(s/explain ::person {:person/first-name "David" :person/address {:address/zipcode "0213"}})
;; val: #:person{:first-name "David", :address #:address{:zipcode "0213"}}
;;      fails spec: :user/person predicate: (contains? % :person/last-name)
;; In: [:person/address]
;;   val: #:address{:zipcode "0213"}
;;   fails spec: :person/address at: [:person/address] predicate: (contains? % :address/line-1)
;; In: [:person/address :address/zipcode]
;;    val: "0213"
;;    fails spec: :address/zipcode at: [:person/address :address/zipcode] predicate: (re-matches #"\d{5}" %)
```

# Generators!

spec provides data generators based on the predicates in the specs

* relies on test.check
* but you only have to depend on it if you use generators

```
(require '[clojure.spec.gen :as gen])

(gen/sample (s/gen int?))
;; (-1 -1 -1 -1 -1 0 -1 -1 -6 -2)
(gen/sample (s/gen :card/suit))
;; (:Spades :Hearts :Hearts ... :Diamonds :Clubs)
(gen/sample (s/gen ::name-or-id))
("" "" -1 0 0 -3 -4 -15 29 "99vdKmNZ3")
```

# Generators!

spec can't always build a usable generator for you ...

```
(s/def ::email (s/and string? #(re-matches email-regex %)))
(gen/sample (s/gen ::email))
;; ExceptionInfo Couldn't satisfy such-that predicate after 100 tries.
```

# Generators!

... but you can supply your own

```
(s/def ::email (s/spec (s/and string? #(re-matches email-regex %))
 :gen
 #(gen/bind
   (gen/tuple (gen/such-that not-empty (gen/string-alphanumeric))
              (gen/such-that not-empty (gen/string-alphanumeric))
              (gen/elements #{"com" "org"}))
    (fn [[a b c]] (gen/return (str a "@" b "." c))))))

(gen/sample (s/gen ::email))
;; ("V@R.org" "k@s.org" "sN@N.com" "1jC@dr.org" "7j@E6Ox.org" "sHK5h@2Tv06.org"
;;  "su@lKjh.org" "ffHilXI@K6U.com" "6hjw8@C2YF0F.com" "0@y.com")
```

# Function specs

```
(defn ranged-rand
  "Returns random int in range start <= rand < end"
  [start end]
  (+ start (long (rand (- end start)))))

(s/fdef ranged-rand
  :args (s/and (s/cat :start int? :end int?)
               #(< (:start %) (:end %)))
  :ret int?
  :fn (s/and #(>= (:ret %) (-> % :args :start))
             #(< (:ret %) (-> % :args :end))))
```

# Exercise

```
(pprint (s/exercise (:args (s/get-spec `ranged-rand))))
;; ([(-1 0) {:start -1, :end 0}]
;;  [(-1 1) {:start -1, :end 1}]
;;  [(-6 -1) {:start -6, :end -1}]
;;  [(0 1) {:start 0, :end 1}]
;;  [(-3 0) {:start -3, :end 0}]
;;  [(-48 119) {:start -48, :end 119}]
;;  [(-26 -1) {:start -26, :end -1}]
;;  [(-20 -1) {:start -20, :end -1}]
;;  [(-4 -2) {:start -4, :end -2}]
;;  [(-87 69) {:start -87, :end 69}])
```

# Exercise

```
(pprint (s/exercise (:ret (s/get-spec `ranged-rand))))
;; ([-1 -1]
;;  [0 0]
;;  [-1 -1]
;;  [1 1]
;;  [0 0]
;;  [0 0]
;;  [0 0]
;;  [-1 -1]
;;  [-24 -24]
;;  [212 212])
```

# Exercise a function

```
(pprint (s/exercise-fn `ranged-rand))
;; ([(-1 1) -1]
;;  [(0 2) 0]
;;  [(-2 -1) -2]
;;  [(-1 2) 1]
;;  [(-1 1) -1]
;;  [(-1 2) 0]
;;  [(-8 31) 6]
;;  [(-2 0) -2]
;;  [(-41 -8) -13]
;;  [(-94 -5) -46])
```

# Test a function with test/check

```
(require '[clojure.spec.test :as stest])
(pprint (stest/check `ranged-rand))
;; ({:spec
;;   #object[clojure.spec$fspec_impl$reify__13789 0x670ac042 "clojure.spec$fspec_impl$reify__13789@670ac042"],
;;   :clojure.spec.test.check/ret
;;   {:result true, :num-tests 1000, :seed 1471391975232},
;;   :sym user/ranged-rand})
```

# Instrument (and unstrement)

```
(defn fn-that-calls-ranged-rand-incorrectly []
  (ranged-rand 1.2 3.4))

(fn-that-calls-ranged-rand-incorrectly)
;; 2.2
(stest/instrument `ranged-rand)

(fn-that-calls-ranged-rand-incorrectly)
;; In: [0] val: 1.2 fails at: [:args :start] predicate: int?
;; :clojure.spec/args  (1.2 3.4)
;; :clojure.spec/failure  :instrument
;; :clojure.spec.test/caller {:file "form-init9107063959735765068.clj",
;;                            :line 155, :var-scope user/fn-that-calls-ranged-rand-incorrectly}

(stest/unstrument `ranged-rand)

(fn-that-calls-ranged-rand-incorrectly)
;; 1.2
```

# Documentation with s/describe

```
(s/describe ::name-or-id)
;; (or :name string? :id int?)

(s/describe `ranged-rand)
;; (fspec :args (and (cat :start int? :end int?)
;;                   (fn* [p1__9461#] (< (:start p1__9461#) (:end p1__9461#))))
;;        :ret int?
;;        :fn (fn [{{start :start, end :end} :args, ret :ret}]
;;              (and (>= ret start) (< ret end))))
```

# Documentation with doc

```
(doc ::name-or-id)
-------------------------
:spec.examples.guide/name-or-id
Spec
  (or :name string? :id int?)

(doc ranged-rand)
-------------------------
spec.examples.guide/ranged-rand
([start end])
  Returns random int in range start <= rand < end
Spec
  args: (and (cat :start int? :end int?) (< (:start %) (:end %)))
  ret: int?
  fn: (and (>= (:ret %) (-> % :args :start)) (< (:ret %) (-> % :args :end)))
```

# Summary

Docstrings aren't enough. We want something executable that we can leverage in a number of contexts.

# Summary

clojure.spec gives us all of this from a single predicate

- Documentation
- Validation and Error reporting
- Destructuring
- Instrumentation, test-data generation, and generative test generation

# Learn more

- https://clojure.org/about/spec
- https://clojure.org/guides/spec