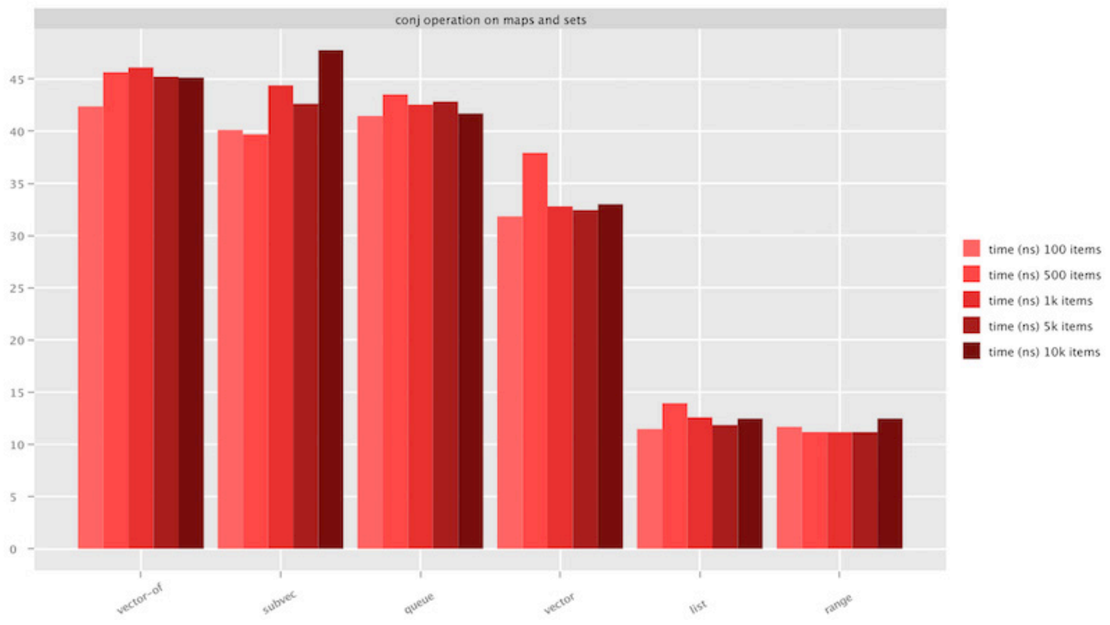
Figure 7.4. Efficiency of `conj` for sequential data structures.



The very low timings on lists and ranges are because conj is in this case is just creating a cons-cell, a small object that is linked to the rest of the collection and is not performing any other logic. The reader should also keep in mind that `conj` is not designed for bulk-inserts of many elements (for which other functions like *“into”* are better suited). When many `conj` operations are needed (for example inside a *loop-recur*) lists or vectors are the fastest options.

7.2.2 get

`function since 1.0`

Listing 7.8 → *Collection Access, Associative Lookup*

```
(get
  ([map key])
  ([map key not-found]))
```

In the group of the functions that make access to collections (the others being *“nth”* and *maps* (although working on other types too):

```
(get {:a "a" :b "b"} :a)
;; "a"
```

`get` design is to avoid throwing exception, preferring `nil` when the collection type is not supported:

```
(get (list 1 2 3) 1)
```

```
;; nil
```

`get` offers a third argument which is returned when the element is not found or the collection is not supported. This, along with the fact of returning `nil` instead of throwing exception, makes `get` quite flexible for handling mixed type input:

```
(def colls [[1 2 3] {:a 1 :b 2} '(1 2 3)])
(def keys [-1 :z 0])
(def messages ["not found" "not found" "not supported"])

(map get colls keys messages)
;; ("not found" "not found" "not supported")
```

Contract

INPUT

"map" type is not limited to *maps*. It can also be:

- *ordered*).
- *ordered* but not transient).
- *vector* (including *sub-vectors* and *native vector*).
- A record created with *"defrecord"*.
- Classes implementing the `java.util.Map` interface (such as `java.util.HashMap` object instances).

When "key" is a number, "map" can additionally be:

- Native Java array (such as those created with *int-array*).
- A string.

Other arguments are:

- "key" can be any positive integer up to `(Integer/MAX_VALUE)`. When "key" is beyond that range, the result of `get` can be difficult to predict, as "key" is truncated to an integer with potential loss of precision.
- "not-found" is optional and can be of any type. "not-found" is returned when the requested index is not present.
- `nil` is accepted as a degenerated collection type. `get` of `nil` always returns `nil` unless "not-found" is provided.

NOTABLE EXCEPTIONS

- `ClassCastException` is thrown when "coll" is a *comparator*. See the examples for additional information.

OUTPUT

`get` returned value has a different meaning depending on the type of "coll":

- The value at key "key" when coll is a *"array-map"* or their transient variation (with the exception of a sorted map that cannot be transient).

- "key" when coll is a *sorted-set* and "key" is one of the elements in the set.
- The item at index "key" when "coll" is a *"vector"*, *native vector*. "key" is assumed numeric in this case.
- The value at field "key" in a *record* instance, similarly to map access.
- In the absence of the requested index, value or item, the "not-found" parameter (when present).
- `nil` when "coll" is `nil` and no default value is given.
- `nil` in all other cases.

Examples

`get` contract contains a few exceptional situation worth highlighting as examples. The first has to do with sorted collections. When elements are added to a sorted collection, types are constrained by the existence of a suitable comparator to compare them. The reason for the constraint comes from the binary tree search algorithm implemented for sorted collections in Clojure. The search compares the target element against the element at the current node and when not equal, uses the result to descend the right or left branch. The trade-off allows O(logN) average search time at the price of reduced flexibility:

```
(get (sorted-map :a 1 :b 2) :c "not-found")     ; ❶
;; not-found

(get (sorted-map :a 1 :b 2) "c" "not-found")    ; ❷
;; ClassCastException clojure.lang.Keyword cannot be cast to java.lang.String
```

❶ `get` is searching for a non-existent key. The default value is thus returned.

❷ Another case of a non existent key, but with a type that cannot be compared to the keywords in the map.

The second exception in `get` contract has to do with[126] :

```
(get (transient {:a 1 :b 2}) :a)    ; ❶
;; 1

(get (transient [1 2 3]) 0)         ; ❷
;; 1

(get (transient #{0 1 2}) 1)        ; ❸
;; nil
```

❶ `get` is used on a *transient map*.

❷ `get` is used on a *transient vector*.

❸ `get` used on a *transient set* is unable to find an existing element.

The last surprising behavior happens on numerical keys exceeding the maximum value

[126] The problem with some transients being not compatible with collection operations is described in the following Jira ticket: dev.clojure.org/jira/browse/CLJ-700

allowed for integers. get implementation makes use of a lossy integer truncation with (Number/intValue) that can return surprising results. Numerical keys are allowed for vectors, strings and arrays:

```
(+ 2 (* 2 (Integer/MAX_VALUE)))  ; ❶
;; 4294967296

(.intValue 4294967296)
;; 0

(get ["a" "b" "c"] 4294967296)   ; ❷
;; "a"

(get "abcd" 4294967296)
;; \a

(get (int-array [0 1 2]) 4294967296)
;; 0
```

❶ A sufficiently large number can mistakenly return valid indexes when coerced into an integer.

❷ get is used on vectors, strings and arrays using a sufficiently large number. The expectation would be for get to return nil instead.

get is the more resilient of the group of functions dedicated to element lookup in a collection. It works on every type (even scalars) at the cost of returning nil instead of throwing exception. It also accepts a nil collection as input, making it a good candidate when the target collection could potentially be nil:

```
(def mixed-bag [{1 "a"} [0 2 4] nil "abba" 3 '(())])

(map #(get % 1) mixed-bag)
```

get also accepts objects implementing the java.util.Map interface, so it works with native Java maps that sometimes results from Java interoperation. Java methods like System/getProperties or System/getenv are useful to gather information about the running environment and they return Java maps. The following example shows how we can search for interesting environment properties using get:

```
(defn select-matching [m k]                                 ; ❶
  (let [regex (re-pattern (str ".*" k ".*"))]
    (->> (keys m)
         (filter #(re-find regex (.toLowerCase %)))         ; ❷
         (reduce #(assoc %1 (keyword %2) (get m %2)) {}))))  ; ❸

(defn search [k]                                            ; ❹
  (merge (select-matching (System/getProperties) k)
         (select-matching (System/getenv) k)))

(search "user")                                             ; ❺

{:USER "reborg"
 :user.country "GB"
 :user.language "en"
 :user.name "reborg"}
```

❶ `select-matching` searches for the given key (or portion of it) inside a Java map.
❷ The regular expression is built from the given key name and used to filter the matching keys regardless of the case.
❸ The following reduce operation builds a new Clojure map with the matching keys. The use of `get` is essential to access the Java map for the related value.
❹ `search` wraps access to the system properties and environment, merging them together before the actual selection.
❺ We can see the result of searching for "user", producing a Clojure *map* containing the matching keys. The content could be different on other systems.

The many ways to access maps in Clojure

`get` is possibly the most formal way to access a map in Clojure. Unless you care specifically about `get` handling of `nil` or Java interoperation, developers prefer others (usually more concise) ways. Those are illustrated below.

"Map as a function" is one of the feature distinguishing Clojure from other languages. It works by using the map as the first item in a list, so it can be interpreted as a function call along with its parameters:

```
({:a 1 :b 2} :b) ; ❶
;; 2

({:a 1 :b 2} :c "Not found") ; ❷
;; "Not found"
```

❶ **Using the hash-map `{:a 1 :b 2}` as a function to access the `:b` key.**
❷ **The optional second argument is returned when the key is not found.**

hash-maps implement the `clojure.lang.IFn` interface, thus defining an `invoke` method that is used when the map appears in a callable position. It supports a second argument to be used as a default when the key is not found, exactly like the `get` function. `get` and "map as a function" even share the same implementation, both ending up calling the method `valAt()` from the `clojure.lang.ILookup` interface.

keywords are also invokable functions similarly to maps. They accept a map as their first argument (and also other type of associative data structures like vectors). The keyword will then lookup inside the map keys for an instance of itself:

```
(:b {:a 1 :b 2}) ; ❶
;; 2

(:c {:a 1 :b 2} "Not found") ; ❷
;; "Not found"
```

❶ *Keywords* **behave like maps when in callable position.**
❷ **Like `get` they also support an optional second argument to return when the key is not found.**

Keywords also implement the `clojure.lang.IFn` interface and delegate to the same `valAt()` method.

Clojure `hash-map` are also instances of `java.util.Map` Java interface, so you can also access them with the `get(Object key)` Java method:

```
(.get {:a 1 :b 2} :b) ; ❶
;; 2

(.getOrDefault {:a 1 :b 2} :c "Not found") ; ❷
;; "Not found"
```

❶ **Note the "." dot before "get". It is syntactic sugar to invoke the Java method "get" on the object** `{"a 1 "b 2}`.

❷ **Unlikely** `get`**, "map as a function" and "keyword as a function",** `java.util.Map` **interface doesn't have an overload of the** `get()` **method taking the optional second argument. There is intstead a specific method** `getOrDefault()`.

find is similar to the other methods seen so far but wrapping results in a `java.util.Map.Entry` object (which Clojure extends in its own `clojure.lang.IMapEntry` interface). Apart from wrapping the final result in a newly created map entry object, it shares the same implementation as `get`:

```
(find {:a 1 :b 2} :b) ; ❶
;; [:b 2]

(type (find {:a 1 :b 2} :b)) ❷
;; clojure.lang.MapEntry
```

❶ Making access to the key in a map using *"find"*.

❷ *"find"* does not support the optional second argument for default values. We can see that the returned type, which is printed like a vector, is instead a `MapEntry` object.

Choosing between one of the possible ways to access a key in a Map is a matter of how hash-maps are used by different applications. Performances is less of a concern in this case, as Map access is overall a very fast operation independently from the method used.

See Also

- *"find"* is similar to `get`, but it returns the entry map tuple (a vector of two element) instead of just the value. If you need to use the value but maintain the relationship to its key, `find` is the perfect choice.
- *"select-keys"* can access multiple keys at once and returns a map with those key-value pairs. Use `select-keys` to pick multiple values at once with their corresponding keys.
- *"nth"* accesses an element by index. `get` works on vectors too, but `nth` is specifically dedicated to that goal. `get` actually uses `nth` when the argument is a vector. Unless you are interested in `get` flexibility with `nil`, prefer using `nth` with vectors.

- *get-in* allows fetching values from withing nested maps (or the different kind of collections supported by `get`) without the need of nesting `get` invocations:

```
(def m {:a "a" :b [:x :y :z]})

(get (get m :b) 0) ; ❶
;; :x

(get-in m [:b 0])) ; ❷
;; x
```

❶ `get` of `get` is used to access the vector at key ":b".

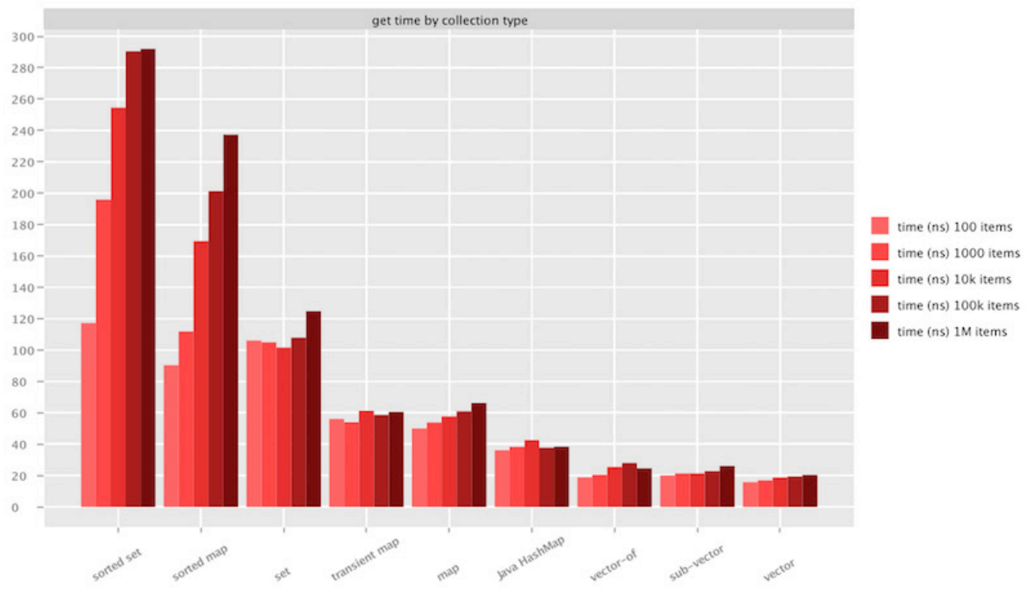
❷ The same element in the vector can be reached with `get-in`

Performance Considerations and Implementation Details

⇒ `O(1)` ***Constant time (best case, unsorted collections)*** ⇒ `O(log n)` ***Logarithmic time (worst case, sorted collections)***

`get` performances are dependent on the input collection type. The following *benchmark chart* shows `get` used against different types of collections at increasingly bigger sizes. The benchmark is obtained by creating a map of randomized keyword keys (for example: `:12376`). The target key to access is selected to be roughly in the middle for sorted collections.

Figure 7.5. Access to several collection types using `get`.



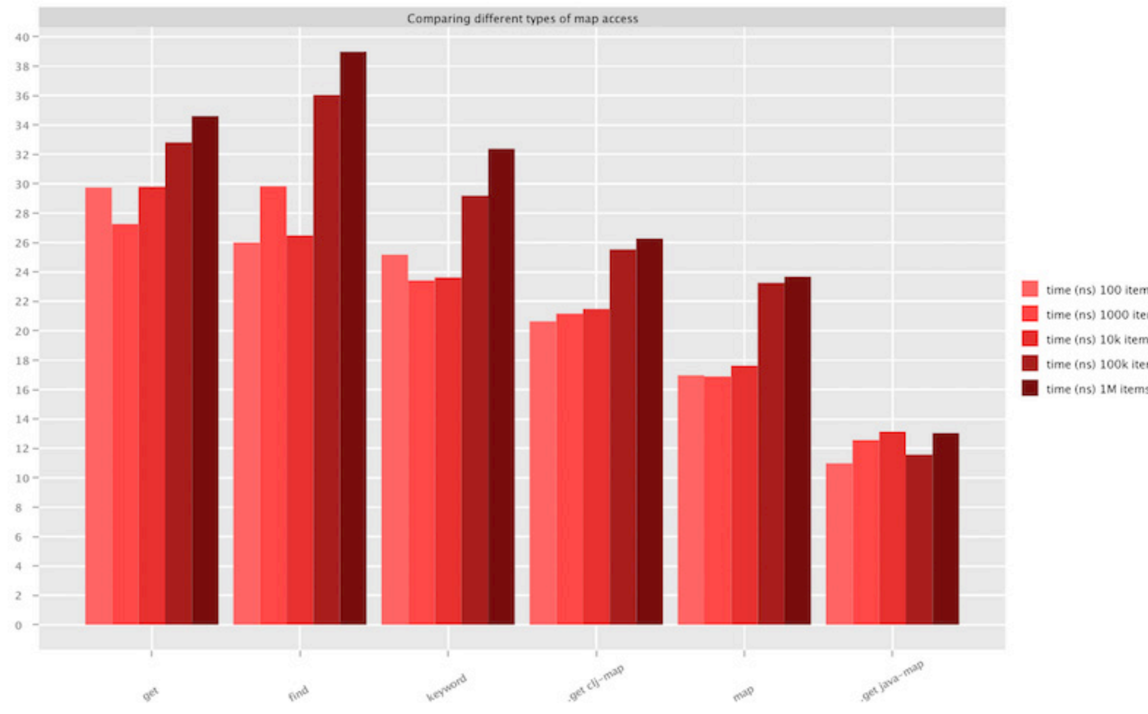
Overall `get` is a fast operation that shouldn't raise particular concerns (times are in nanoseconds). Sorted collections are penalized, because access requires to compare the key at each branch of a balanced tree[127]

In terms of `get` compared to other ways to access maps (we illustrated the different ways to access a map before in the chapter) please see the *following chart*.

The chart compares 6 different ways to access a key in a map at increasingly bigger sizes (up to 1 million keys). Please note again that times are nanoseconds and overall we are talking about very fast operations. The bars in the chart, left to right, are:

- "get" is showing `get` access to the map. It scores around 30 ns average access.
- "find" is using the function *"find"* compared to `get`.
- "keyword" is using the key itself to access the map. It is 10% faster than `get`.
- ".get clj-map" is using the Java interop ".get" Java method to access a persistent hash-map in Clojure.
- "map" is using the map itself as a function to access the key.
- ".get java-map" is again using Java interoperation, but instead of accessing a Clojure map, it's showing what happens when we access a native mutable `java.util.HashMap` object.

Figure 7.6. Comparing different ways to access a map.



[127] Red-black trees are used in Clojure to implement sorted collections. Please see the following Wikipedia entry for more information: vectors are 2 times faster for `get` access.

The chart essentially shows how good Clojure is performing against mutable Java HashMaps (for read-access). It also shows that using a map object directly as a function on keys is one of the best choices both for readability and speed. `get` remains a very good option when we want to take care of potentially `nil` maps without using an explicit condition or we want to access Java HashMaps.

7.2.3 contains?

`function since 1.0`

Listing 7.9 → *Collection Search, Item Inclusion*

```
(contains? [coll key])
```

`contains?` verifies the presence of a "key" (element or index) in a collection supporting direct access lookup. Based on the kind of lookup implementation, it returns `true` or `false` if an element is found which is equal to the key, or an element is found at the index indicated by the key. Here's the most common use:

```
(contains? {:a "a" :b "b"} :b) ; ❶
;; true

(contains? #{:x :y :z} :z) ; ❷
;; true

(contains? [:a :b :c] 1) ; ❸
;; true
```

❶ A key equals to ":b" is found in the hash-map.
❷ An element equals to ":z" is found in the set.
❸ An element is present at index "1" in a vector.

Other types are supported, although their use is less common. The contract section goes into deeper details.

Contract

`contains?` main goal is to check for the presence of an element inside an "associative" data structures. In Clojure "associative" is a broad category including *Sets* are not implementing the `clojure.lang.Associative` interface, but they are supported by `contains?`. The following is an exhaustive list of all supported collections for the "coll" argument and related restrictions:

- *transient*).
- *transient*).
- *vector* (including *sub-vectors* and *native vector*).
- A record created with *"defrecord"*.
- Classes implementing the `java.util.Map` interface (such as `java.util.HashMap` object instances).