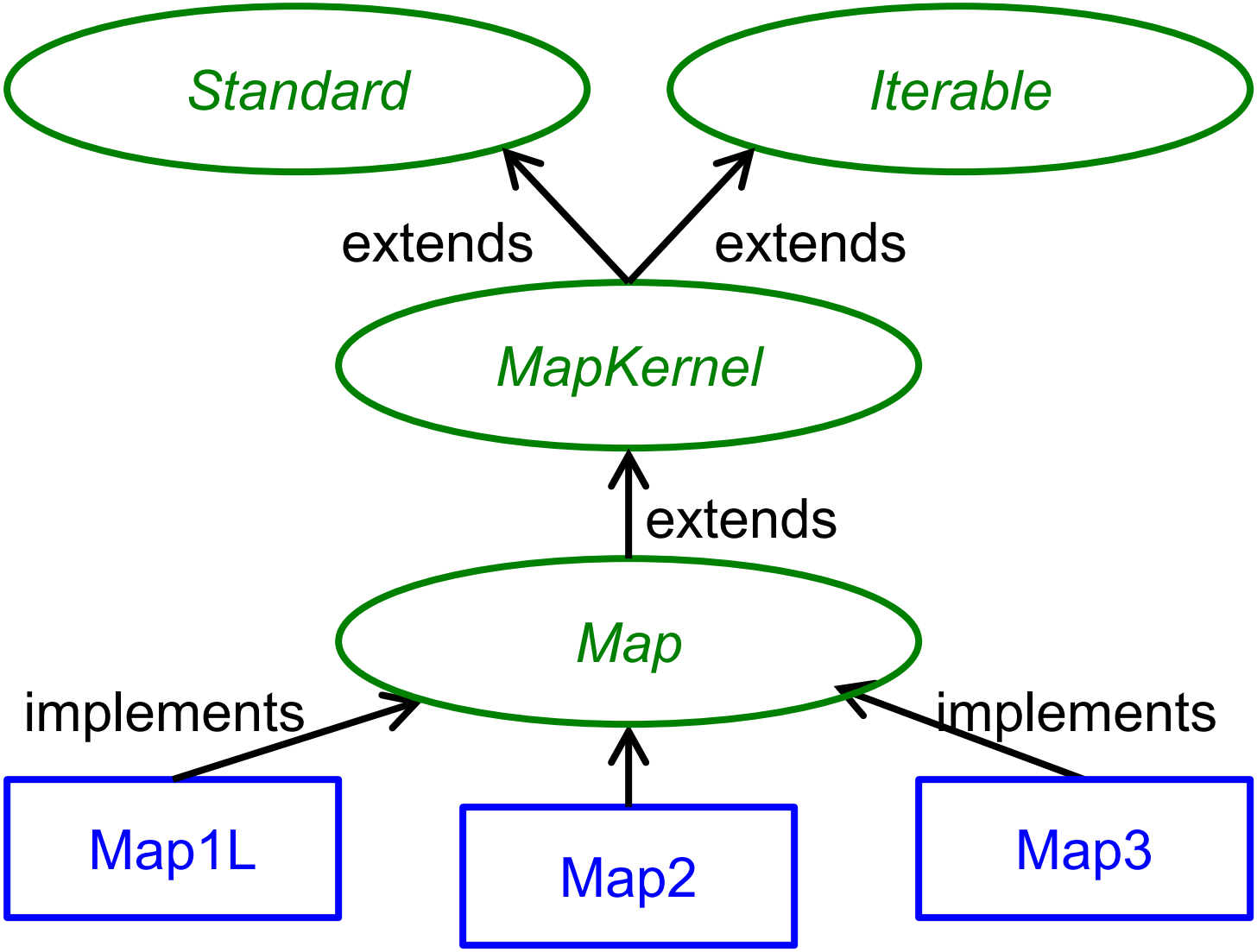
Map

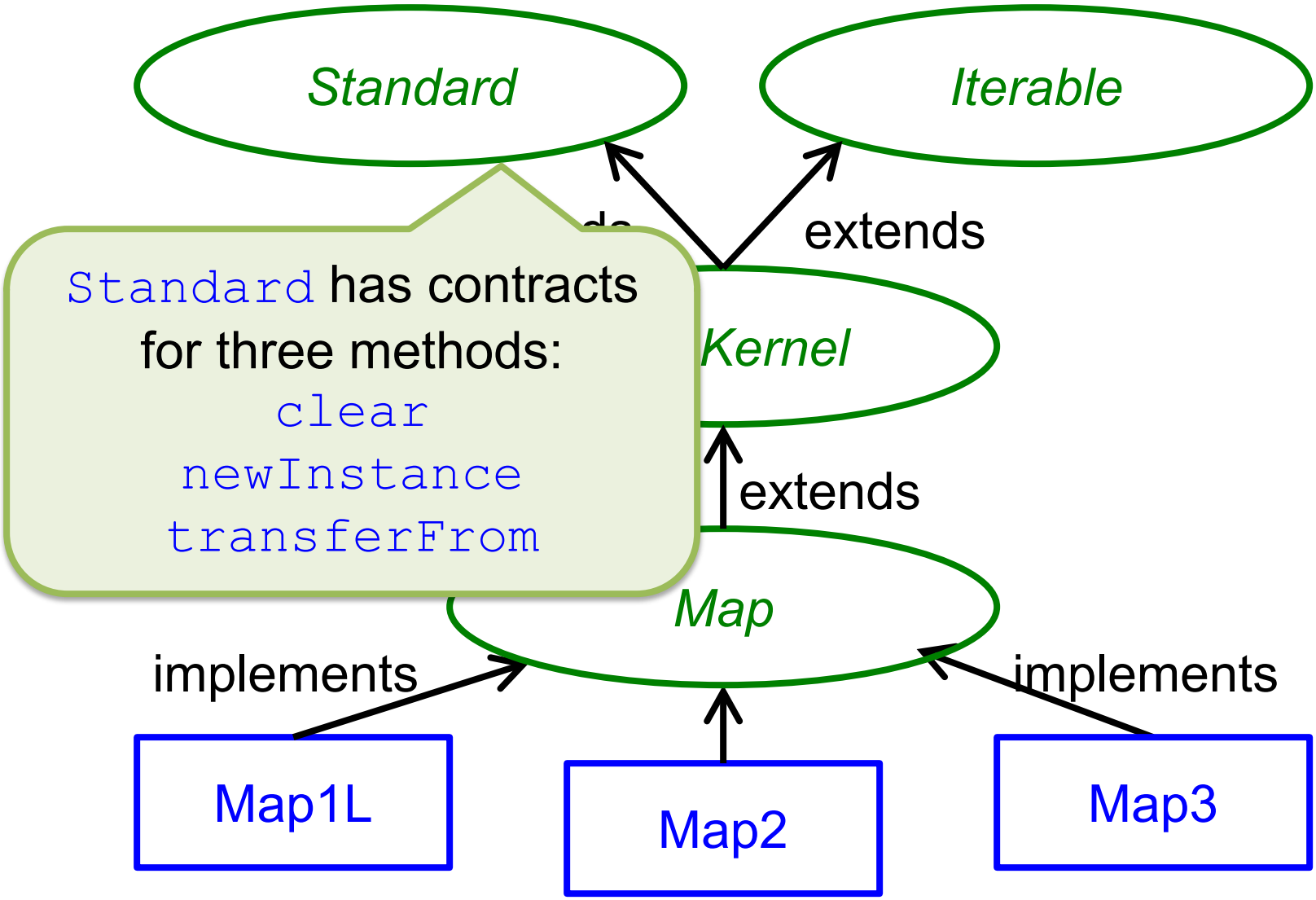
Map

- The ***Map*** component family allows you to manipulate ***mappings*** from ***keys*** (of any type `K`) to ***values*** (of any type `V`)
  - A `Map` variable holds a very simple "database" of keys and their associated values
  - Example: If you need to keep track of the exam grade for each student, you might use a `Map<String,Integer>` variable

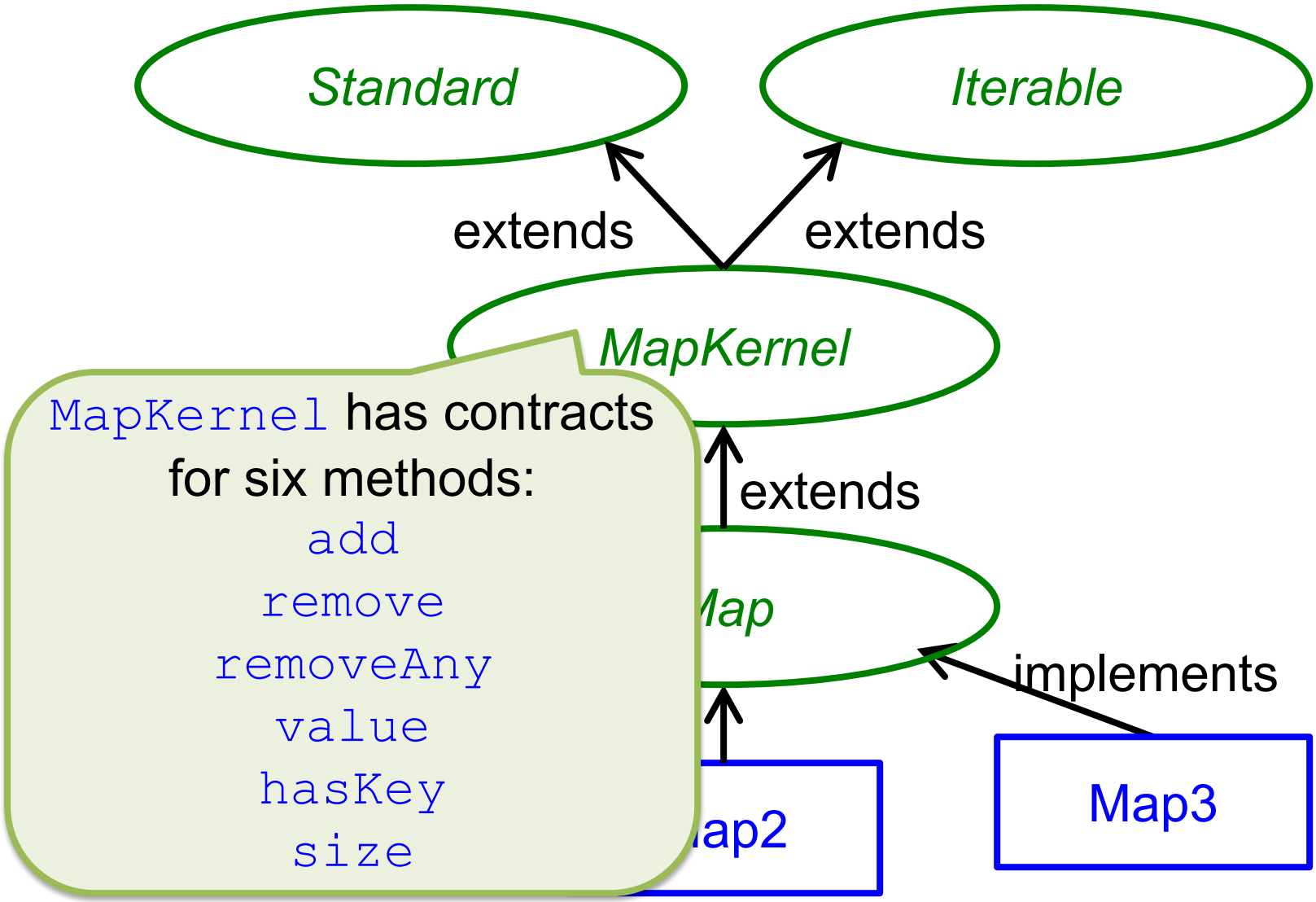
Interfaces and Classes



Interfaces and Classes

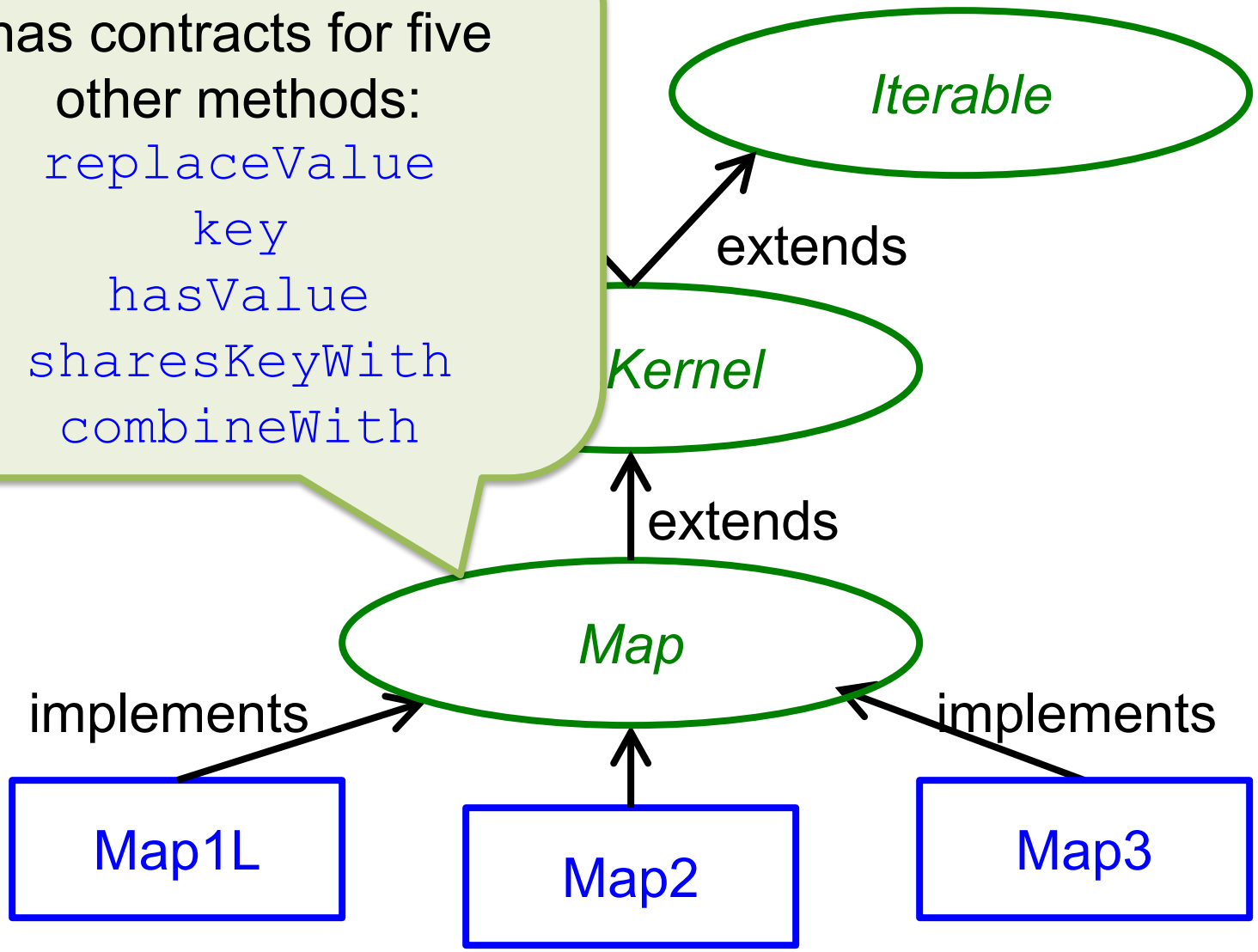


Interfaces and Classes

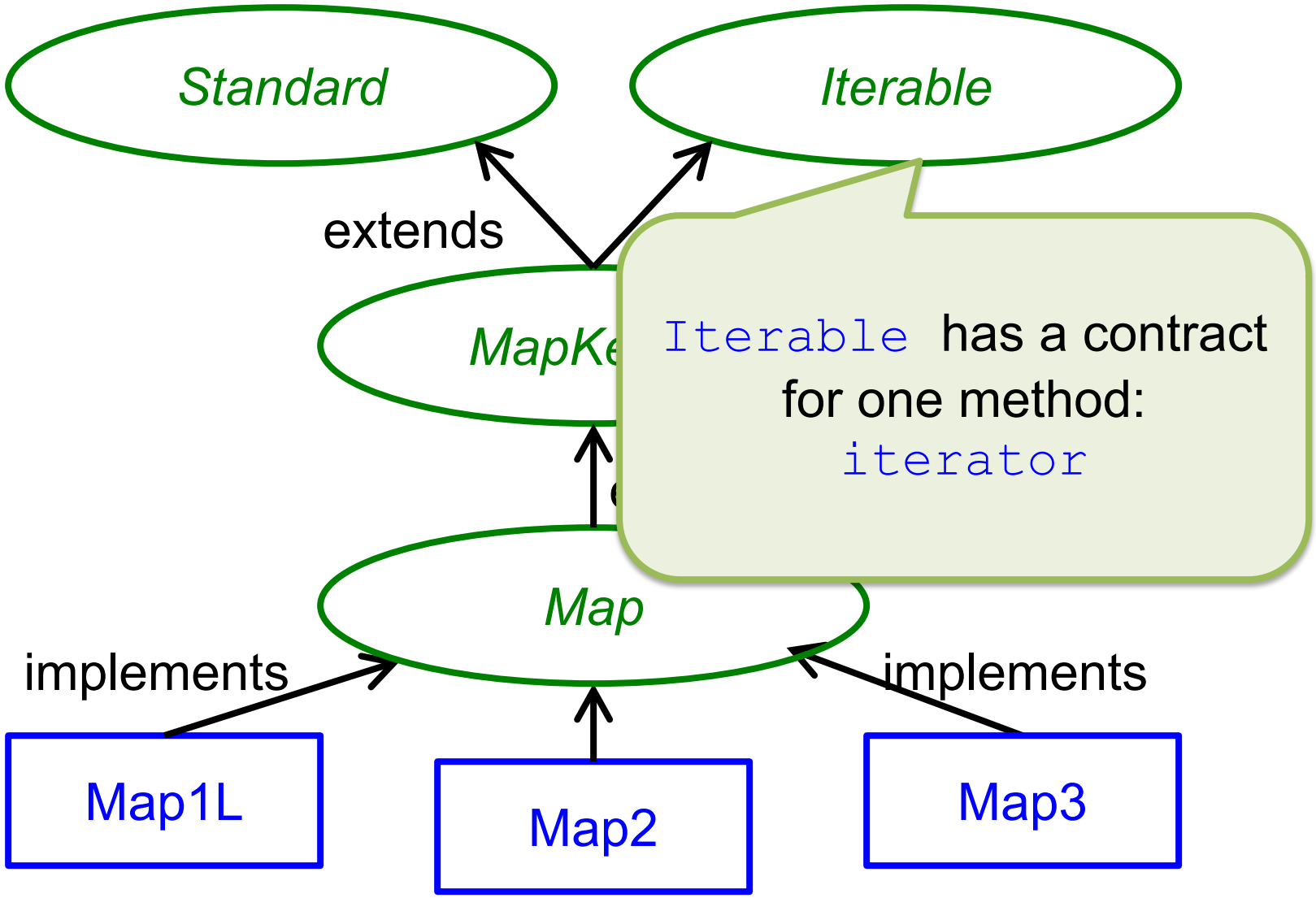


and Classes

`Map`
has contracts for five other methods:
`replaceValue`
`key`
`hasValue`
`sharesKeyWith`
`combineWith`



Interfaces and Classes



Mathematical Model

- The value of a `Map` variable is modeled as a finite set of ordered pairs of type (`K`, `V`) with "the function property", i.e., no two pairs in the set have the same `K` value
  - This is sometimes called a (finite) ***partial function from*** `K` ***to*** `V`

Partial Function

```
PARTIAL_FUNCTION is finite set of
    (key: K, value: V)
 exemplar m
 constraint
   for all key1, key2: K, value1, value2: V
       where ((key1, value1) is in m  and
                 (key2, value2) is in m)
    (if key1 = key2 then value1 = value2)
```

Partial Function

```
PARTIAL_FUNCTION i
    (key: K, value
 exemplar m
 constraint
   for all key1, key2: K, value1, value2: V
       where ((key1, value1) is in m  and
              (key2, value2) is in m)
    (if key1 = key2 then value1 = value2)
```

This formally states “the function property” for a set of ordered pairs.

Domain of a (Partial) Function

```
DOMAIN (
  m: PARTIAL_FUNCTION
): finite set of K
  satisfies
    for all key: K
      (key is in DOMAIN(m)  iff
       there exists value: V
         ((key, value) is in m))
```

Range of a (Partial) Function

```
RANGE (
  m: PARTIAL_FUNCTION
): finite set of V
  satisfies
    for all value: V
      (value is in RANGE(m)  iff
       there exists key: K
         ((key, value) is in m))
```

Mathematical Model

- Formally:

```
type Map is modeled by
    PARTIAL_FUNCTION
```

No-argument Constructor

- Ensures:

  this = { }

Example

Code	State
`Map<String,Integer> m = new Map1L<>();`	

Example

Code	State
`Map<String,Integer> m = new Map1L<>();`	
	m = { }

add

void add(K key, V value)

- Adds the pair (key, value) to **this**.
- Aliases: references key, value
- Updates: **this**
- Requires:

  key ***is not in*** *DOMAIN(****this****)*

- Ensures:

  this *= #****this union*** *{(key, value)}*

Example

Code	State
	m = {("PB", 99), *("BW", 17)}* *k = "PS"* *v = 99*
`m.add(k, v);`	

Example

Is the requires clause satisfied? What is *DOMAIN(m)*?

	State

	`m = {("PB", 99), ("BW", 17)}`
	`k = "PS"`
	`v = 99`

`m.add(k, v);`	

Example

Code	State

	`m = {("PB", 99),`
	`("BW", 17)}`
	`k = "PS"`
	`v = 99`

`m.add(k, v);`	

	`m = {("PB", 99),`
	`("BW", 17),`
	`("PS", 99)}`
	`k = "PS"`
	`v = 99`

Example

Note the aliases created here, which you cannot see in the tracing table; you should be able to draw the appropriate diagram showing them.

	State

	`m = {("PB", 99), ("BW", 17)}`
	`k = "PS"`
	`v = 99`

	`m = {("PB", 99), ("BW", 17), ("PS", 99)}`
	`k = "PS"`
	`v = 99`

Another Interface

- The `Map` interface includes an interface for another related generic type, `Map.Pair`
- Its mathematical model is simply an ordered pair of a key and a value
- Formally:

```
type Map.Pair is modeled by
    (key: K, value: V)
```

`Map.Pair` Methods

- This (immutable) type has only a constructor (taking a `K` and a `V`) and a ***getter*** method for each pair component
  - `K key()`
    - Returns the first component of **`this`**
    - Aliases: reference returned by `key`
  - `V value()`
    - Returns the second component of **`this`**
    - Aliases: reference returned by `value`

remove

`Map.Pair<K,V> remove(K key)`

- Removes from **`this`** the pair whose first component is `key` and returns it.
- Updates: **`this`**
- Requires:

  *`key` **`is in`** `DOMAIN(`**`this`**`)`*
- Ensures:

  `remove.key = key` **`and`**

  *`remove` **`is in`** `#`**`this`** **`and`**

  `this` *`=`* `#`**`this`** *`\ {remove}`*

Example

Code	State
	`m = {("PB", 99), ("BW", 17)}` `k = "BW"`
`Map.Pair<String,Integer> p = m.remove(k);`	

Example

Code	State
	`m = {("PB", 99), ("BW", 17)}` `k = "BW"`
`Map.Pair<String,Integer> p = m.remove(k);`	

	`m = {("PB", 99)}`
	`k = "BW"`
	`p = ("BW", 17)`

removeAny

`Map.Pair<K,V> removeAny()`

- Removes and returns an arbitrary pair from **`this`**.
- Updates: **`this`**
- Requires:

  `|this| > 0`
- Ensures:

  `removeAny is in #this and`

  `this = #this \ {removeAny}`

Example

Code	State
	`m = {("PB", 99), ("BW", 17), ("PS", 99)}`
`Map.Pair<String,Integer> p = m.removeAny();`	

Example

Code	State
	`m = {("PB", 99), ("BW", 17), ("PS", 99)}`
`Map.Pair<String,Integer> p = m.removeAny();`	
	`m = {("PB", 99), ("BW", 17)}` `p = ("PS", 99)`

value

`V value(K key)`

- Reports the value associated with `key` in **`this`**.
- Aliases: reference returned by `value`
- Requires:

  `key` ***`is in`*** *`DOMAIN(`****`this`****`)`*
- Ensures:

  `(key, value)` ***`is in this`***

Example

Code	State

	`m = {("PB", 99), ("BW", 17)}`
	`k = "PB"`
	`v = -423`

`v = m.value(k);`	

Example

Code	State

	`m = {("PB", 99),`
	`("BW", 17)}`
	`k = "PB"`
	`v = -423`

`v = m.value(k);`	

	`m = {("PB", 99),`
	`("BW", 17)}`
	`k = "PB"`
	`v = 99`

Example

Note the alias created here, which you cannot see in the tracing table; you should be able to draw the appropriate diagram showing it.

	State

	`m = {("PB", 99), ("BW", 17)}`
	`k = "PB"`
	`v = -423`

	`m = {("PB", 99), ("BW", 17)}`
	`k = "PB"`
	`v = 99`

hasKey

boolean hasKey(K key)

- Reports whether there is a pair in **this** whose first component is key.
- Ensures:

  hasKey =

  (key ***is in*** *DOMAIN(****this****))*

Example

Code	State
	m = {("PB", 99), *("BW", 17)}* *k = "PB"*
boolean b = m.hasKey(k);	

Example

Code	State
	`m = {("PB", 99), ("BW", 17)}` `k = "PB"`
`boolean b = m.hasKey(k);`	

	`m = {("PB", 99), ("BW", 17)}`
	`k = "PB"`
	`b = true`

`size`

`int` `size()`

- Reports the size (cardinality) of **`this`**.
- Ensures:

  `size` = |***`this`***|

replaceValue

V replaceValue(K key, V value)

- Replaces the value associated with `key` in **`this`** by `value`, and returns the old value.
- Aliases: reference `value`
- Updates: **`this`**
- Requires:

  key ***is in*** *DOMAIN(****this****)*
- Ensures:

  this *= (#****this*** *\ {(key, replaceValue)})*
  union *{(key, value)}* ***and***
  (key, replaceValue) ***is in*** *#****this***

Example

Code	State

	`m = {("PB", 99), ("BW", 17)}`
	`k = "PB"`
	`v = 85`

`Integer oldV = m.replaceValue(k, v);`	

Example

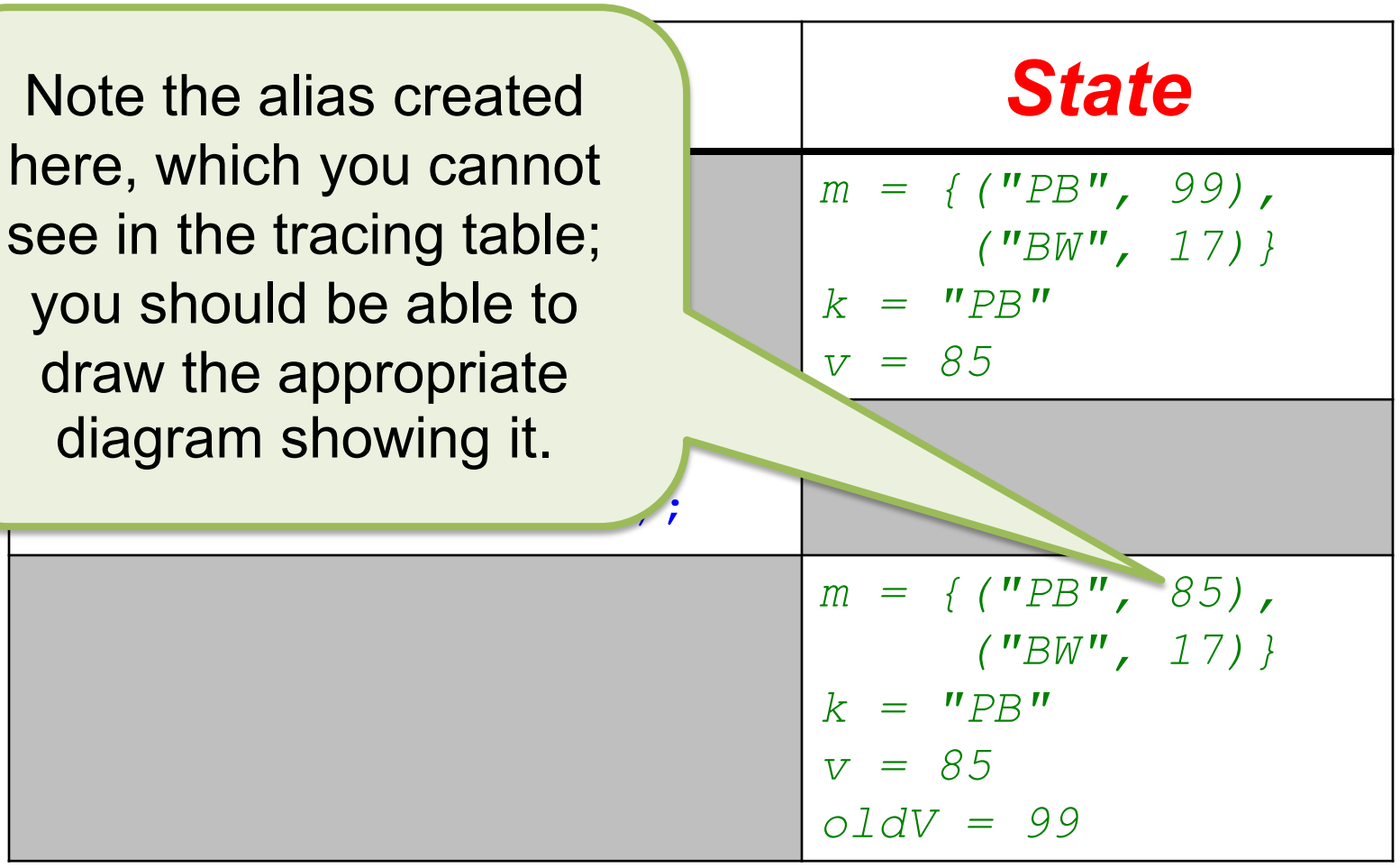
Code	State

	`m = {("PB", 99), ("BW", 17)}`
	`k = "PB"`
	`v = 85`

`Integer oldV = m.replaceValue(k, v);`	

	`m = {("PB", 85), ("BW", 17)}`
	`k = "PB"`
	`v = 85`
	`oldV = 99`

Example


State
m = {("PB", 99),
("BW", 17)}
k = "PB"
v = 85
m = {("PB", 85),
("BW", 17)}
k = "PB"
v = 85
oldV = 99
Note the alias created here, which you cannot see in the tracing table; you should be able to draw the appropriate diagram showing it.

Another Example

Code	State

	`m = {("PB", 99), ("BW", 17)}`
	`k = "PB"`
	`v = 85`

`v = m.replaceValue(k, v);`	

Another Example

Code	State

	`m = {("PB", 99), ("BW", 17)}`
	`k = "PB"`
	`v = 85`

`v = m.replaceValue(k, v);`	

	`m = {("PB", 85), ("BW", 17)}`
	`k = "PB"`
	`v = 99`

Another Example

This use of the method avoids creating an alias: it ***swaps*** `v` with the value in `m` that was previously associated with `k`.

	State

	`m = {("PB", 99), ("BW", 17)}`
	`k = "PB"`
	`v = 85`

`v = m.replaceValue(k, v);`	

	`m = {("PB", 85), ("BW", 17)}`
	`k = "PB"`
	`v = 99`

key

`K key(V value)`

- Reports some key associated with `value` in **`this`**.
- Aliases: reference returned by `key`
- Requires:

  `value` ***`is in`*** *`RANGE(this)`*
- Ensures:

  `(key, value)` ***`is in this`***

Example

Code	State

	`m = {("PB", 99),`
	`("BW", 17)}`
	`k = "xyz"`
	`v = 99`

`k = m.key(v);`	

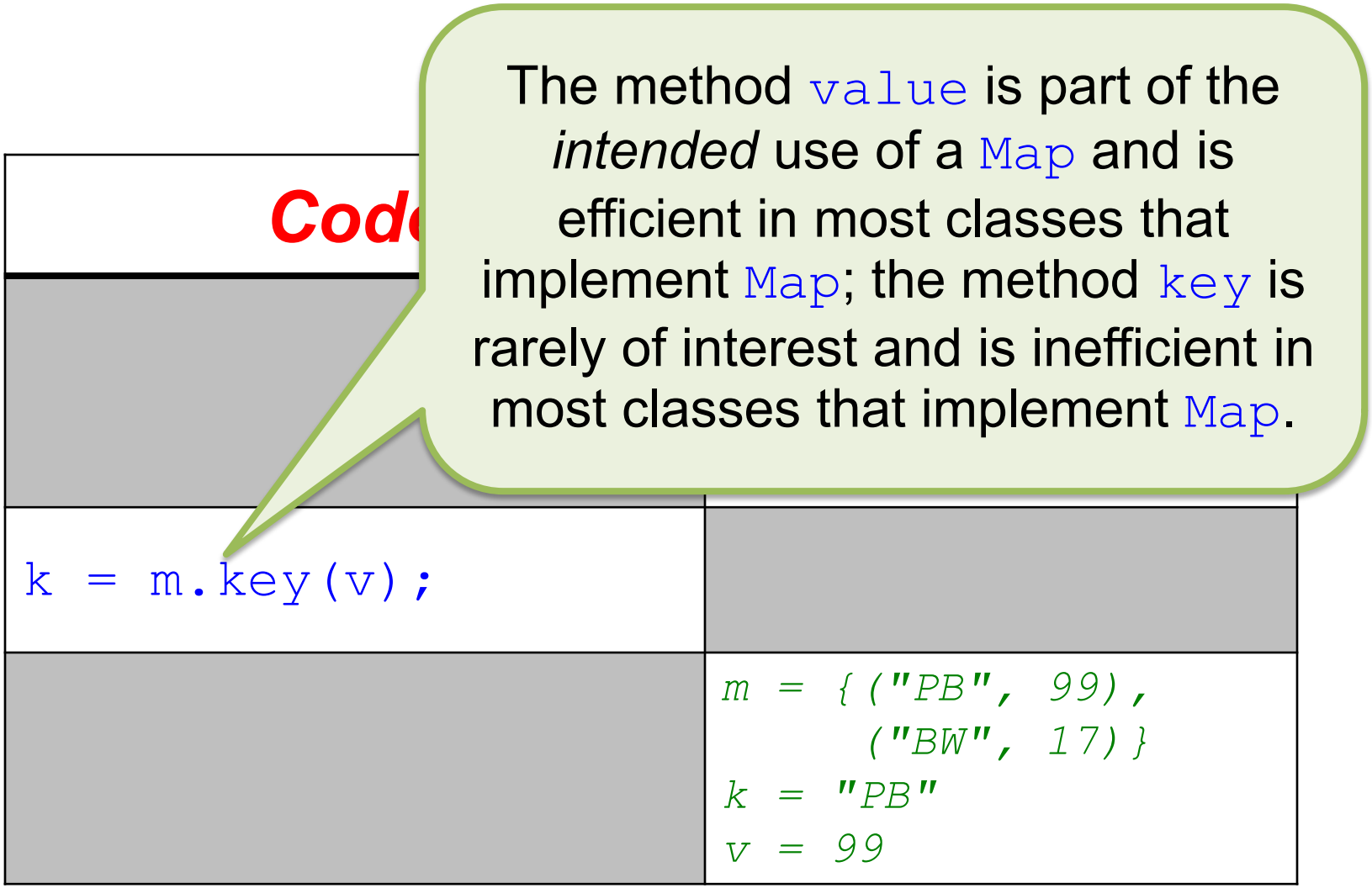
Example

Code	State

	`m = {("PB", 99), ("BW", 17)}`
	`k = "xyz"`
	`v = 99`

`k = m.key(v);`	

	`m = {("PB", 99), ("BW", 17)}`
	`k = "PB"`
	`v = 99`


Code
k = m.key(v);
m = {("PB", 99),
("BW", 17)}
k = "PB"
v = 99
The method value is part of the intended use of a Map and is efficient in most classes that implement Map; the method key is rarely of interest and is inefficient in most classes that implement Map.

hasValue

boolean hasValue(V value)

- Reports whether there is a pair in **this** whose second component is value.
- Ensures:

hasValue =

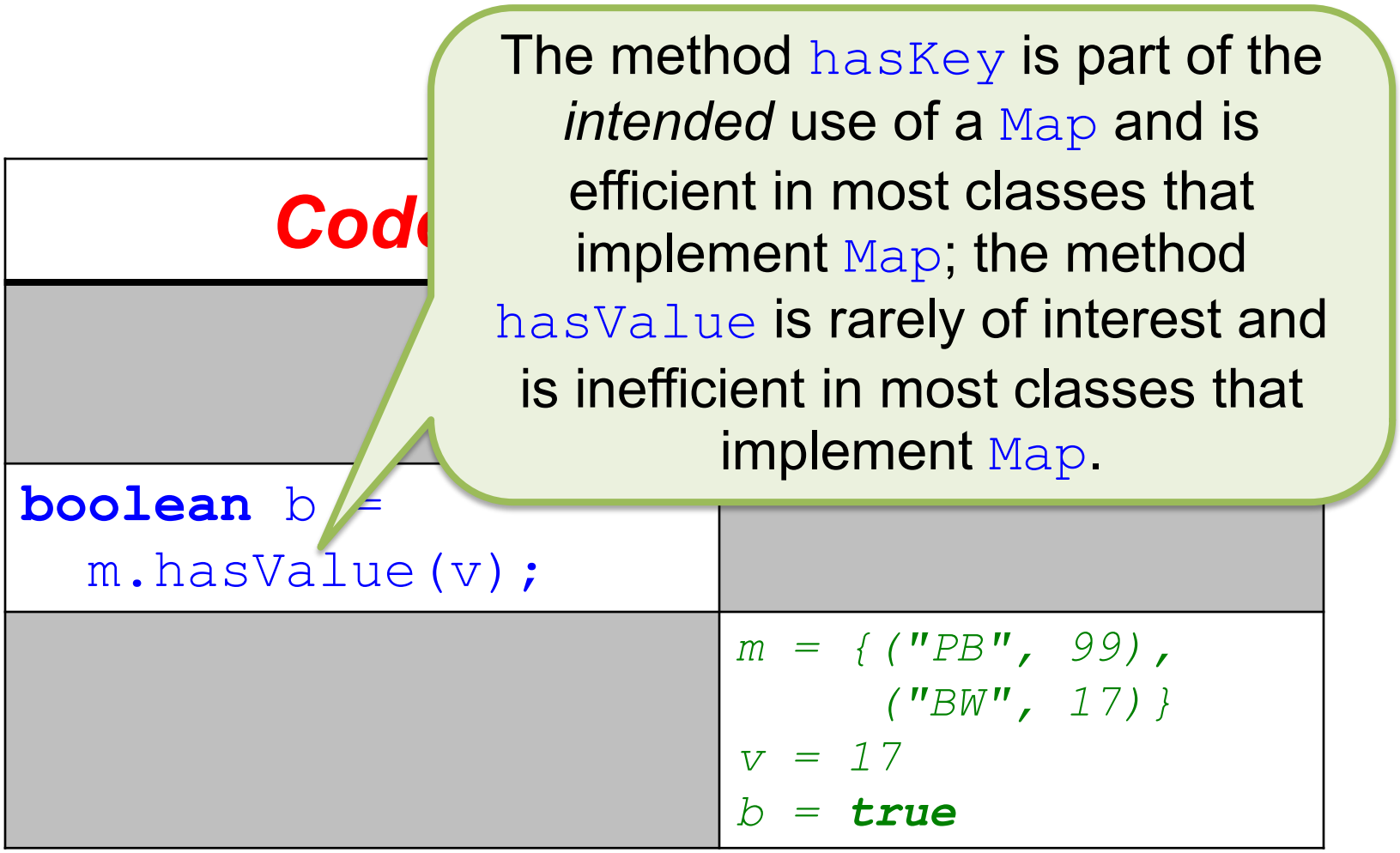
(value ***is in*** *RANGE(****this****))*

Example

Code	State
	`m = {("PB", 99), ("BW", 17)}` `v = 17`
`boolean b = m.hasValue(v);`	

Example

Code	State
	m = {("PB", 99), *("BW", 17)}* *v = 17*
boolean b = m.hasValue(v);	
	m = {("PB", 99), *("BW", 17)}* *v = 17* *b =* ***true***


Code
boolean b =
m.hasValue(v);
m = {("PB", 99),
("BW", 17)}
v = 17
b = true
The method hasKey is part of the intended use of a Map and is efficient in most classes that implement Map; the method hasValue is rarely of interest and is inefficient in most classes that implement Map.

combineWith

void combineWith(Map<K,V> m)

- Combines m with **this**.
- Updates: **this**
- Clears: m
- Requires:

  *DOMAIN(**this**) **intersection***

  DOMAIN(m) = {}
- Ensures:

  ***this** = #**this** **union** #m*

Example

Code	State
	`m1 = {("PB", 99), ("BW", 17)}` `m2 = {("PS", 99)}`
`m1.combineWith(m2);`	

Example

Code	State
	`m1 = {("PB", 99), ("BW", 17)}` `m2 = {("PS", 99)}`
`m1.combineWith(m2);`	
	`m1 = {("PB", 99), ("BW", 17), ("PS", 99)}` `m2 = { }`

sharesKeyWith

boolean sharesKeyWith(Map<K,V> m)

- Reports whether **this** and m have any keys in common.
- Ensures:

sharesKeyWith=

*(DOMAIN(**this**) **intersection***

DOMAIN(m) /= {})

Example

Code	State
	`m1 = {("PB", 99), ("BW", 17)}` `m2 = {("PS", 99)}`
`boolean b = m1.sharesKeyWith(m2);`	

Example

Code	State
	m1 = {("PB", 99), ("BW", 17)} *m2 = {("PS", 99)}*
boolean b = m1.sharesKeyWith(m2);	
	m1 = {("PB", 99), ("BW", 17)} *m2 = {("PS", 99)}* *b =* ***false***

iterator

`Iterator<Map.Pair<K,V>> iterator()`

- Returns an iterator over a set of elements of type `Map.Pair<K,V>`.
- Ensures:

```
entries(~this.seen * ~this.unseen) = this
  and
|~this.seen * ~this.unseen| = |this|
```

Example

- Suppose you have a `Map` that keeps track of the names and associated salaries of all employees in the company:

```
Map<String,NaturalNumber> m =
   new Map1L<>();
...
```

Sample For-Each Loop: Danger!

- Here's how you *might try* to give every employee a $10,000 raise:

```
NaturalNumber raise =
  new NaturalNumber2(10000);
for (Map.Pair<String,NaturalNumber> p : m) {
  NaturalNumber salary = p.value();
  salary.add(raise);
}
```

Sample For-Each Loop:

-

```
Na
   new
for (Map.Pair<String,Natura      ber> p : m) {
  NaturalNumber salary = p.value();
  salary.add(raise);
}
```

Draw this diagram: `p` holds aliases to some key and its associated value in `m`; the method `value` returns an alias to a `NaturalNumber` that is also in the `Map m`; so, changing that `NaturalNumber` incidentally changes the values of *both* `p` and `m` (even though no `Map` method is called in the loop).

Sample For-Each Loop: Danger!

- Here's how you *might try* to give every employee a $10,000 raise:

```
NaturalNumber raise =
  new NaturalNumber2(10000);
for (Map.Pair<String,NaturalNumber>          m) {
  NaturalNumber salary = p.value();
  salary.add(raise);
}
```

Danger!

This violates the rules for using iterators and for-each loops!

The Safe Way

- Here's how you *should* give every employee a $10,000 raise:

```
NaturalNumber raise = new NaturalNumber2(10000);
Map<String, NaturalNumber> temp = m.newInstance();
temp.transferFrom(m);
while (temp.size() > 0) {
  Map.Pair<String, NaturalNumber> p =
    temp.removeAny();
  p.value().add(raise);
  m.add(p.key(), p.value());
}
```

Draw this diagram: `p` holds references to some key and its associated value, but now they are *not* in any `Map` and `p` is *not* in any `Map`; the method `value` returns an alias to a `NaturalNumber` in the `Map.Pair p`; so, changing that `NaturalNumber` does *not* incidentally change the value of `m` or `temp` (even though that actually would be OK for this loop).

-

```
Nat
Map
temp.transferFrom(m);
while (temp.size() > 0) {
  Map.Pair<String, NaturalNumber> p =
    temp.removeAny();
  p.value().add(raise);
  m.add(p.key(), p.value());
}
```

Resources

- OSU CSE Components API: `Map`
  - http://web.cse.ohio-state.edu/software/common/doc/