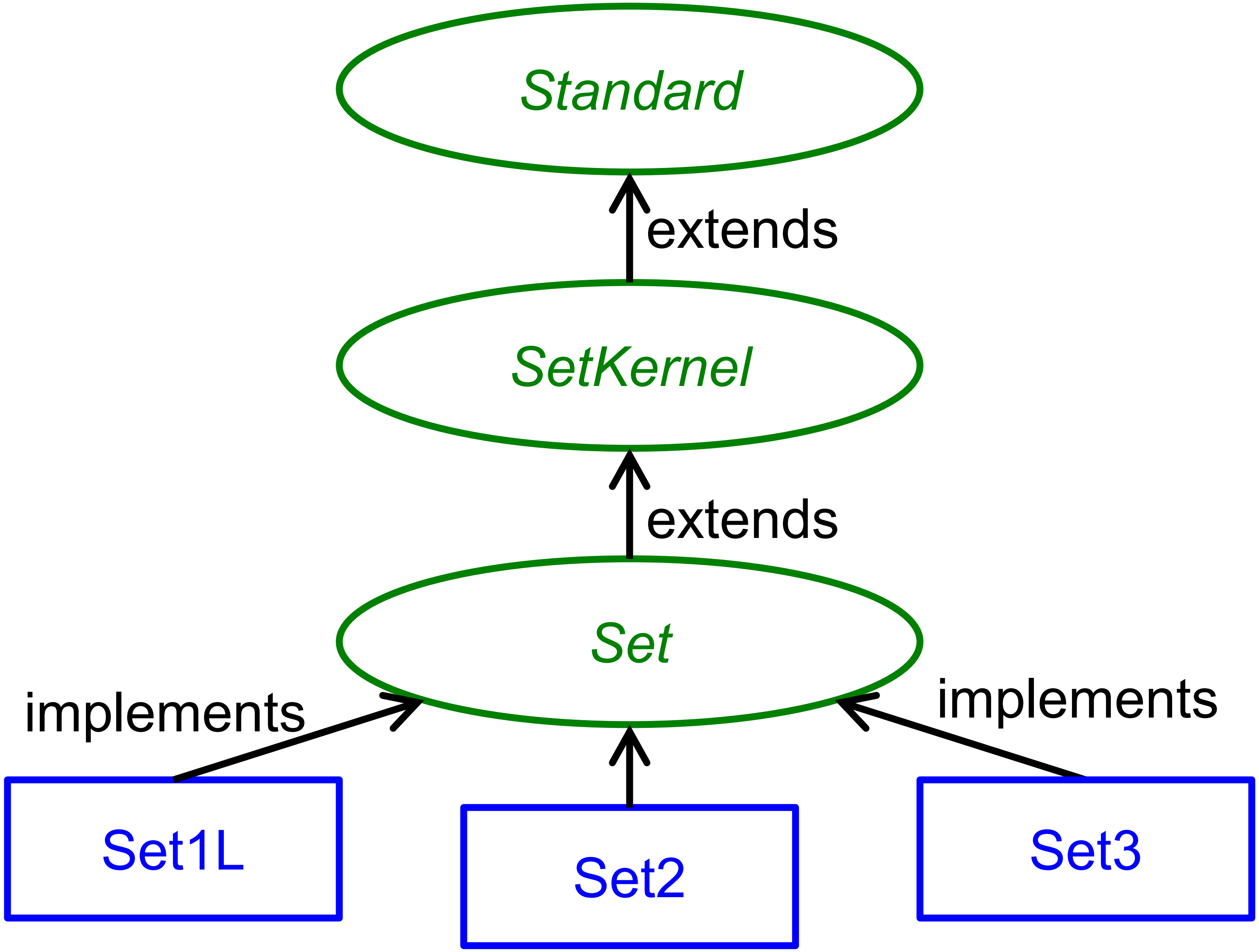
Set

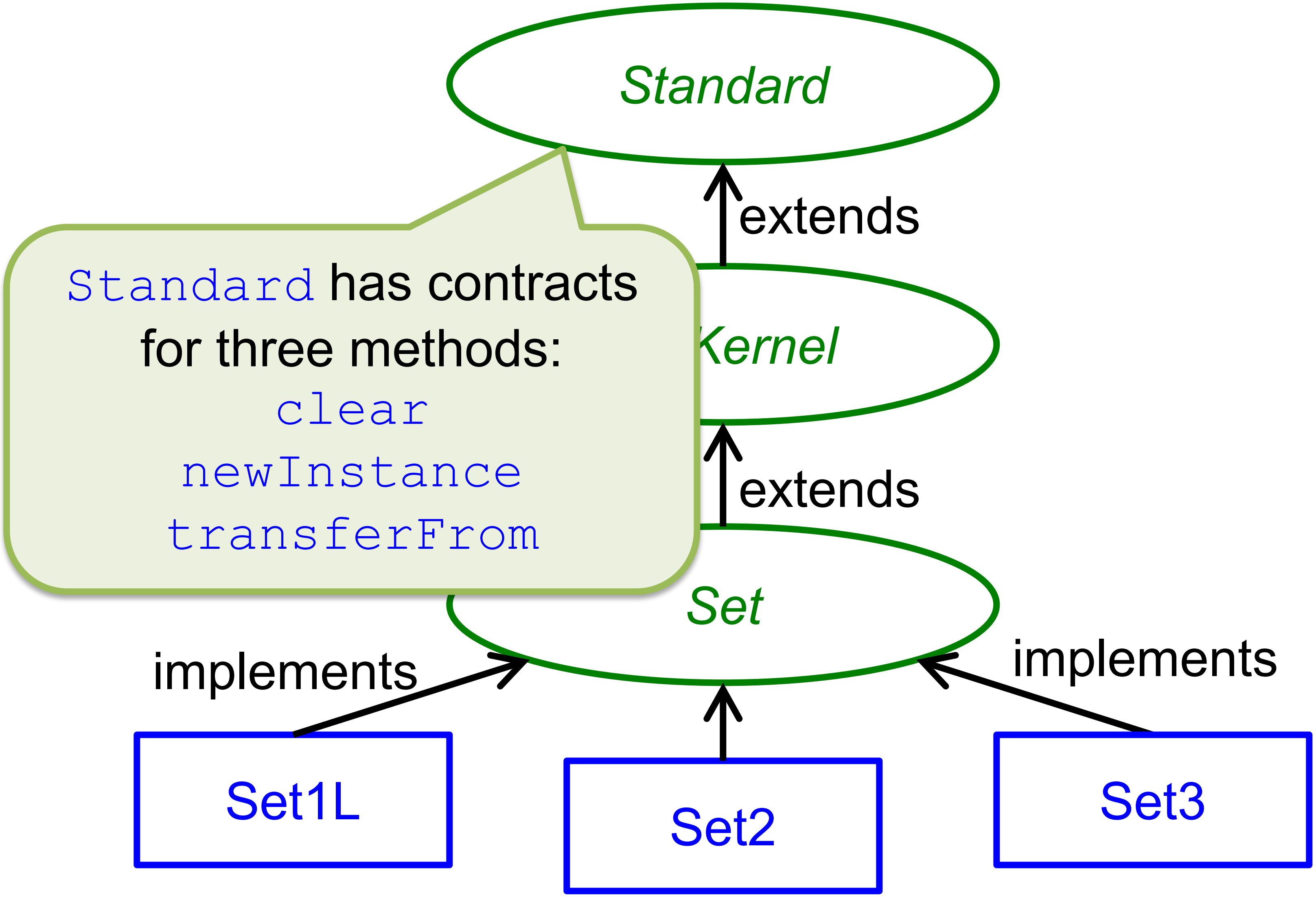
Set

- The ***Set*** component family allows you to manipulate finite sets of elements of any (arbitrary) type

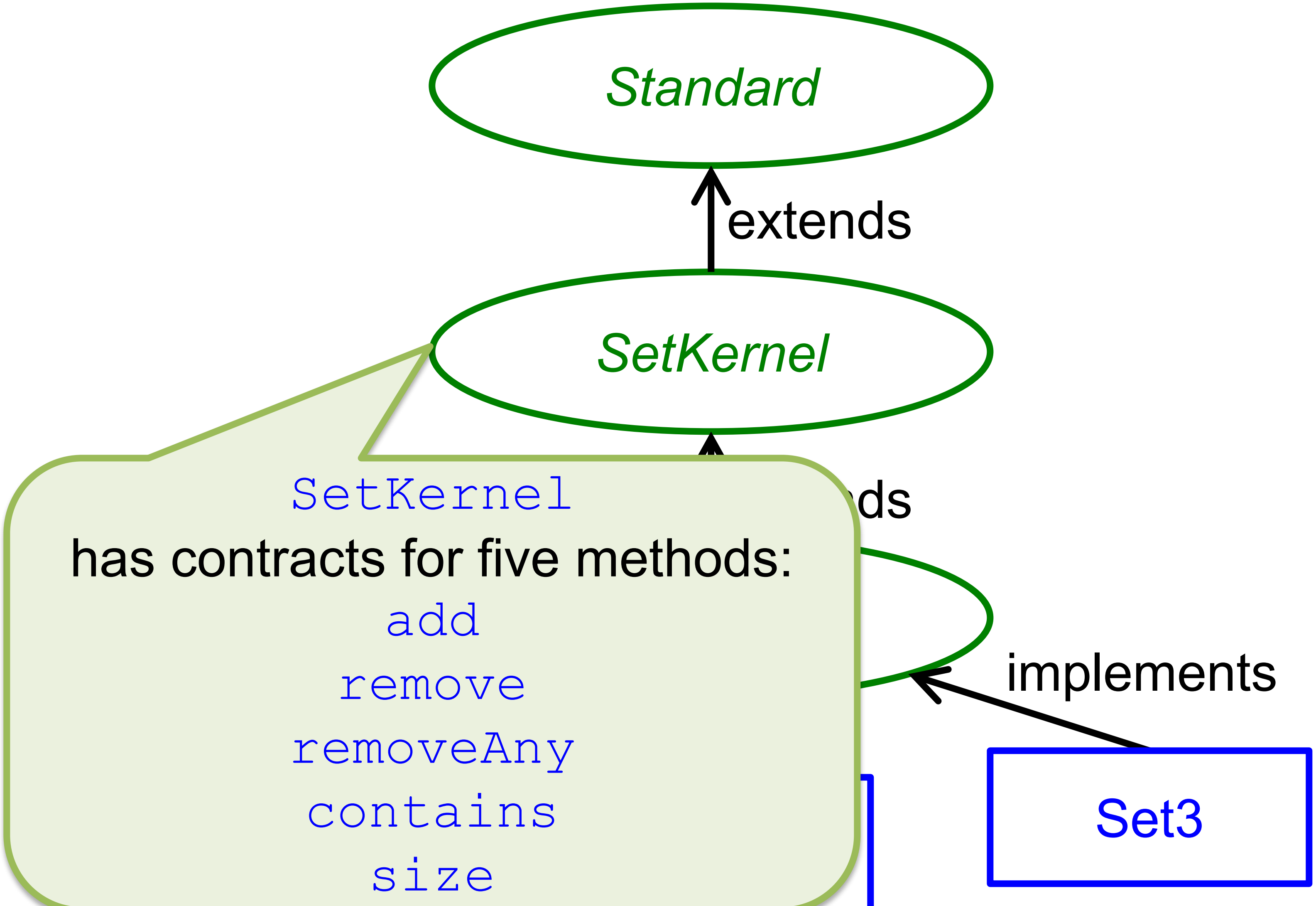
Interfaces and Classes



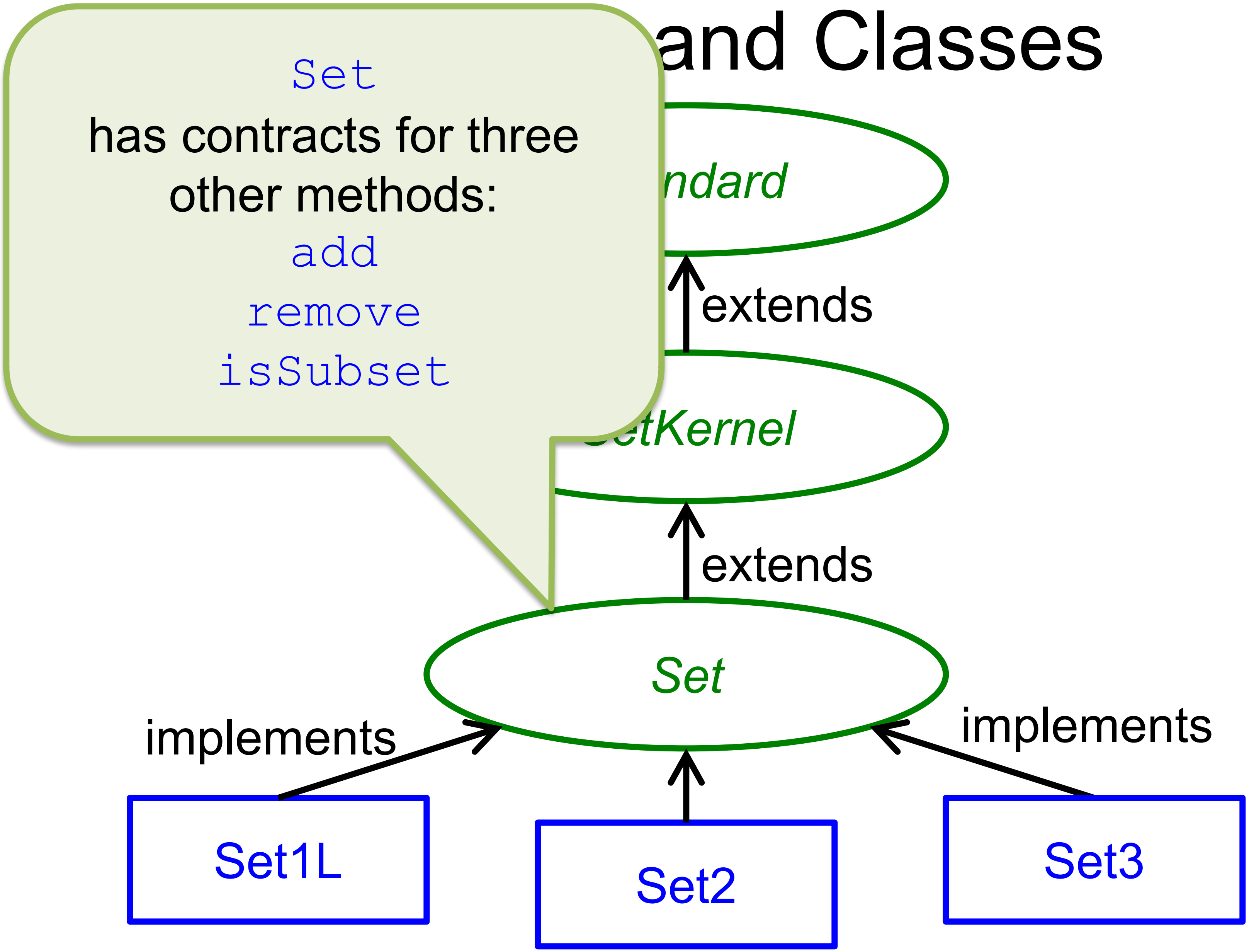
Interfaces and Classes



Interfaces and Classes



…and Classes


Set
has contracts for three
other methods:
add
remove
isSubset
…ndard
extends
…etKernel
extends
Set
implements
implements
Set1L
Set2
Set3

Mathematical Model

- The value of a `Set` variable is modeled as a (finite) set of elements of type `T`
- Formally:

type *Set* ***is modeled by***
  finite set of *T*

Constructors

- There is one ***constructor*** for each implementation class for `Set`
- As always:
  - The name of the constructor is the name of the implementation class
  - The constructor has its own contract (which is in the kernel interface `SetKernel`)

No-argument Constructor

- Ensures:

  this $= \{\ \}$

Example

Code	***State***
`Set<Integer> si =` `  new Set1L<>();`	

Example

Code	State
`Set<Integer> si =` `  new Set1L<>();`	
	$si = \{\ \}$

Methods for `Set`

- All the methods for `Set` are ***instance methods***, i.e., you call them as follows:

  `s.methodName(arguments)`

  where `s` is an initialized non-null variable of type `Set<T>` for some `T`

`add`

```
void add(T x)
```

- Adds `x` to **`this`**.
- Aliases: reference `x`
- Updates: **`this`**
- Requires:

  x is not in this
- Ensures:

  this = #this union {x}

Example

Code	State
	si = { 49, 3 } k = 70
`si.add(k);`	

Example

Code	State
	`si = { 49, 3 }` `k = 70`
`si.add(k);`	
	`si = { 49, 3, 70 }` `k = 70`

Example

Note the aliasing here between the “70s”, not shown in the tracing table but visible if you draw a diagram of this situation.

	State
	`si = { 49, 3 }` `= 70`
`si.add(k);`	
	`si = { 49, 3, 70 }` `k = 70`

`remove`

`T remove(T x)`

- Removes `x` from **`this`**, and returns it.
- Updates: **`this`**
- Requires:

  `x is in this`

- Ensures:

  `this = #this \ {x} and`

  `remove = x`

Example

Code	***State***
	si = { 49, 3, 70 } *k = 3* *m = -17*
`m = si.remove(k);`	

Example

Code	State

	`si = { 49, 3, 70 }`
	`k = 3`
	`m = -17`

`m = si.remove(k);`	

	`si = { 49, 70 }`
	`k = 3`
	`m = 3`

Example

Code	State

	`si = { 49, 3, 70 }`
	`k = 3`
	`m = -17`

	`si = { 49, 70 }`
	`k = 3`
	`m = 3`

The precondition for `remove` (*x* ***is in this***) is satisfied whether or not there is aliasing involving the “3s” in this situation. Why?

`removeAny`

`T removeAny()`

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

  |***this***| > *0*

- Ensures:

  removeAny ***is in*** *#****this*** ***and***

  this *= #****this*** *\ {removeAny}*

Example

Code	State
	`si = { 49, 3, 70 }` `k = 134`
`k = si.removeAny();`	

Example

Code	State
	`si = { 49, 3, 70 }` `k = 134`
`k = si.removeAny();`	
	`si = { 3, 70 }` `k = 49`

Example

Other possible outcomes are:

`si = { 49, 70 }`

`k = 3`

or:

`si = { 49, 3 }`

`k = 70`

	State
	`i = { 49, 3, 70 }` `= 134`
	`si = { 3, 70 }` `k = 49`

`contains`

`boolean` `contains(T x)`

- Reports whether `x` is in **`this`**.
- Ensures:

  `contains = (x` ***`is in this`****`)`*

Example

Code	State
	si = { 49, 3, 70 } *k = -58*
`boolean b = si.contains(k);`	

Example

Code	State
	si = { 49, 3, 70 } *k = -58*
`boolean b = si.contains(k);`	
	si = { 49, 3, 70 } *k = -58* *b =* ***false***

Example

Code	State
	si = { 49, 3, 70 } *k = 70*
`boolean b = si.contains(k);`	

Example

Code	State
	si = { 49, 3, 70 } *k = 70*
`boolean b = si.contains(k);`	
	si = { 49, 3, 70 } *k = 70* *b =* ***true***

Example

The condition checked by `contains` (*x* ***is in this***) is satisfied whether or not there is aliasing involving the "70s" in this situation. Why?

	State
	si = { 49, 3, 70 } *k = 70*
`si.contains(k)`	
	si = { 49, 3, 70 } *k = 70* *b =* ***true***

`size`

`int` `size()`

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

  size = |***this***|

Example

Code	State
	`si = { 49, 3, 70 }` `n = -45843`
`n = si.size();`	

Example

Code	State
	`si = { 49, 3, 70 }` `n = -45843`
`n = si.size();`	
	`si = { 49, 3, 70 }` `n = 3`

Overloading

- A method with the same name as another method, but with a different ***parameter profile*** (number, types, and order of formal parameters) is said to be ***overloaded***
- A method may not be overloaded on the basis of its return type
- Java disambiguates between overloaded methods based on the number, types, and order of arguments at the point of a call

`add`

`void add(Set<T> s)`

- Adds to **`this`** all elements of `s` that are not already in **`this`**, also removing just those elements from `s`.
- Updates: **`this`**, `s`
- Ensures:

  `this = #this union #s and`

  `s = #this intersection #s`

add

void add(Set<T> s)

- Adds to **this** all e… not already in **th**… those elements fr…
- Updates: **this**, s…
- Ensures:

this** = #**this** **union** #s **and

*s = #**this** **intersection** #s*

The add method for receivers of type Set<T> is overloaded:
- one method takes an argument of type T, and
- one method takes an argument of type Set<T>.

Example

Code	*State*
	s1 = { 1, 2, 3, 4 } *s2 = { 3, 4, 5, 6}*
`s1.add(s2);`	

Example

Code	State
	`s1 = { 1, 2, 3, 4 }` `s2 = { 3, 4, 5, 6}`
`s1.add(s2);`	
	`s1 = { 1, 2, 3, 4, 5, 6 }` `s2 = { 3, 4 }`

In other words, this *moves* all elements of `#s2 \ #s1` from `s2` into `s1`; it "conserves" objects of type `T`.

	ate
	s1 = { 1, 2, 3, 4 } s2 = { 3, 4, 5, 6}
`s1.add(s2);`	
	s1 = { 1, 2, 3, 4, 5, 6 } s2 = { 3, 4 }

`remove`

`Set<T> remove(Set<T> s)`

- Removes from **`this`** all elements of `s` that are also in **`this`**, leaving `s` unchanged, and returns the elements actually removed.
- Updates: **`this`**
- Ensures:

  this *=* *#**this*** *\ s* ***and***

  *remove = #**this intersection** s*

remove

`Set<T> remove(Set<T> s)`

- Removes from **this** all elements of s that are also in **this**
- returns the ele
- Updates: **this**
- Ensures:

this = #***thi***

remove = #***this intersection*** *s*

The `remove` method for receivers of type `Set<T>` is overloaded:

- one method takes an argument of type `T`, and
- one method takes an argument of type `Set<T>`.

Example

Code	State
	s1 = { 1, 2, 3, 4 } s2 = { 3, 4, 5, 6} s3 = { 10 }
`s3 = s1.remove(s2);`	

Example

Code	State

	`s1 = { 1, 2, 3, 4 }`
	`s2 = { 3, 4, 5, 6}`
	`s3 = { 10 }`

`s3 = s1.remove(s2);`	

	`s1 = { 1, 2 }`
	`s2 = { 3, 4, 5, 6}`
	`s3 = { 3, 4 }`

In other words, this “conserves” all elements of *#s1* and *#s2*; they all wind up in some `Set<T>` rather than being “lost”.

	State
	s1 = { 1, 2, 3, 4 } *s2 = { 3, 4, 5, 6}* *s3 = { 10 }*
`s3 = s1.remove(s2);`	
	s1 = { 1, 2 } *s2 = { 3, 4, 5, 6}* *s3 = { 3, 4 }*

`isSubset`

```
boolean isSubset(Set<T> s)
```

- Reports whether **`this`** is a subset of `s`.
- Ensures:

  isSubset = ***this is subset of*** *s*

Example

Code	State
	s1 = { 2, 4 } s2 = { 1, 2, 3, 4 }
`boolean b = s1.isSubset(s2);`	

Example

Code	State
	s1 = { 2, 4 } *s2* = { 1, 2, 3, 4 }
`boolean b = s1.isSubset(s2);`	
	s1 = { 2, 4 } *s2* = { 1, 2, 3, 4 } *b* = ***true***

Example

Code	***State***
	s1 = { 3, 4, 5 } *s2 = { 1, 2, 3, 4 }*
`boolean b =` `  s1.isSubset(s2);`	

Example

Code	State
	s1 = { 3, 4, 5 } *s2 = { 1, 2, 3, 4 }*
`boolean b = s1.isSubset(s2);`	
	s1 = { 3, 4, 5 } *s2 = { 1, 2, 3, 4 }* *b = **false***

Iterating Over a `Set`

- Suppose you want to do something with each of the elements of a `Set<T> s`
- How might you do that?

Iterating With `removeAny`

```
Set<T> temp = s.newInstance();
temp.transferFrom(s);
while (temp.size() > 0) {
  T x = temp.removeAny();
  // do something with x
  s.add(x);
}
```

Iterating With `removeAny`

```
Set<T> temp = s.newInstance();
temp.transferFrom(s);
while (temp.size() > 0) {
  T x = tem
  // do so
  s.add(x)
}
```

Recall that `newInstance` returns a new object of the same ***object type*** (***dynamic type***) as the receiver, as if it were a no-argument constructor; but we don't need to *know* the object type of `s` to get this new object.

Iterating With `removeAny`

```
Set<T> temp = s.newInstance();
temp.transferFrom(s);
while (temp.s       () > 0) {
   T x = te
   // do so
   s.add(x)
}
```

Why `transferFrom` rather than `copyFrom`?

- Performance: there is no need for a copy, and `transferFrom` is far more efficient.
- We really want `s` to be empty to start the iteration, and this does it.

Iterating With `removeAny`

- This code has the following properties:
  - It introduces no dangerous aliases, so it is relatively easy to reason about; just think about values, not references
  - If what you want to do with each element is to change it, then the approach works because you may change the value of `x` each time through the loop body
  - It is reasonably efficient (making no copies of elements of type `T`, though it does use `removeAny` and `add`, and these could be slow)

Iterating With `removeAny`

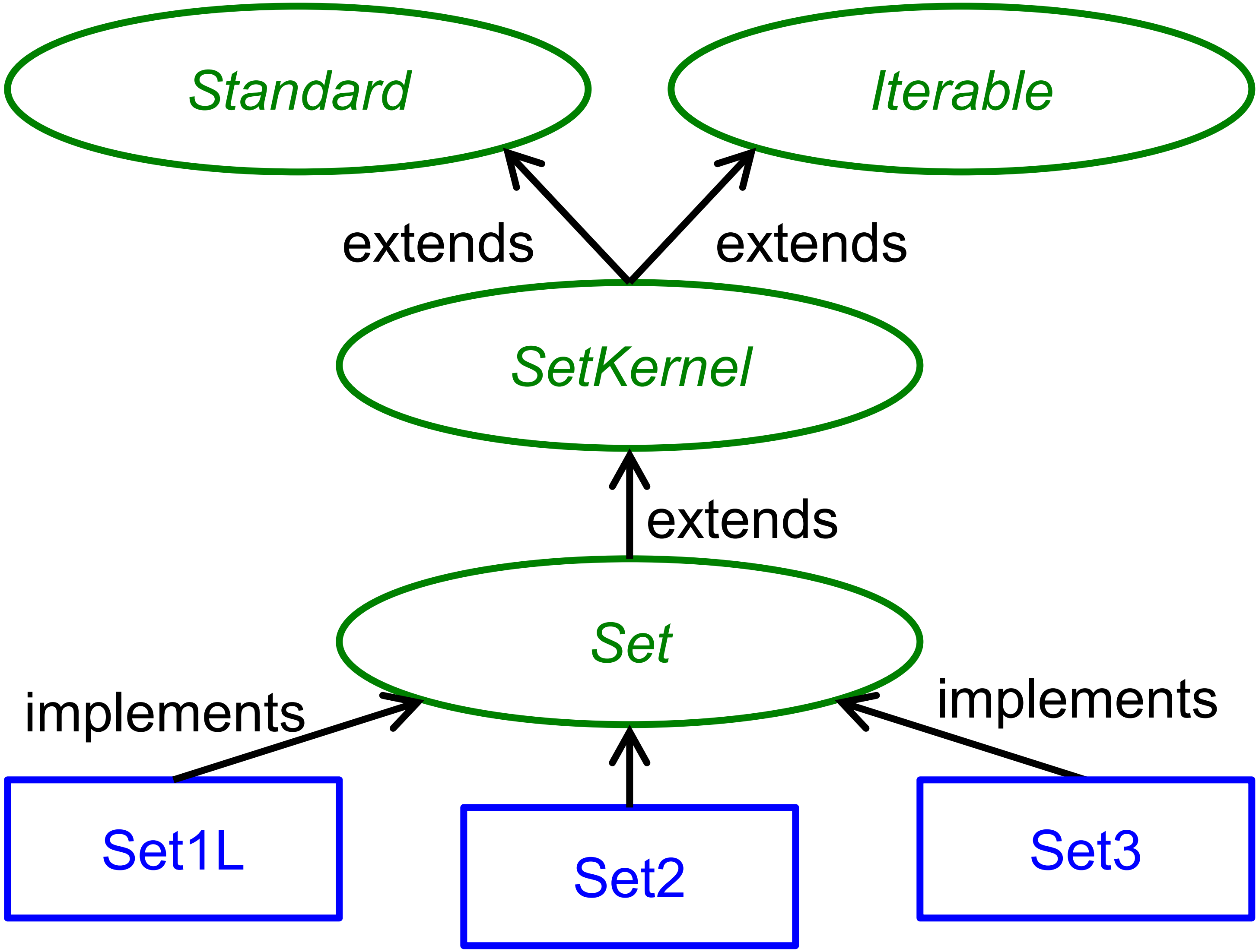
- This code has the following properties:
  - It introduces no dangerous aliases, so it is relatively easy to reason about; ju t think about values, not references
  - If what you want to … nge it, then the approac … nge the value of `x` each
  - It is reasonably effi … elements of type `T`, … y and `add`, and these could be slow)

It does introduce an alias (where?)
but it is of no consequence (why?).

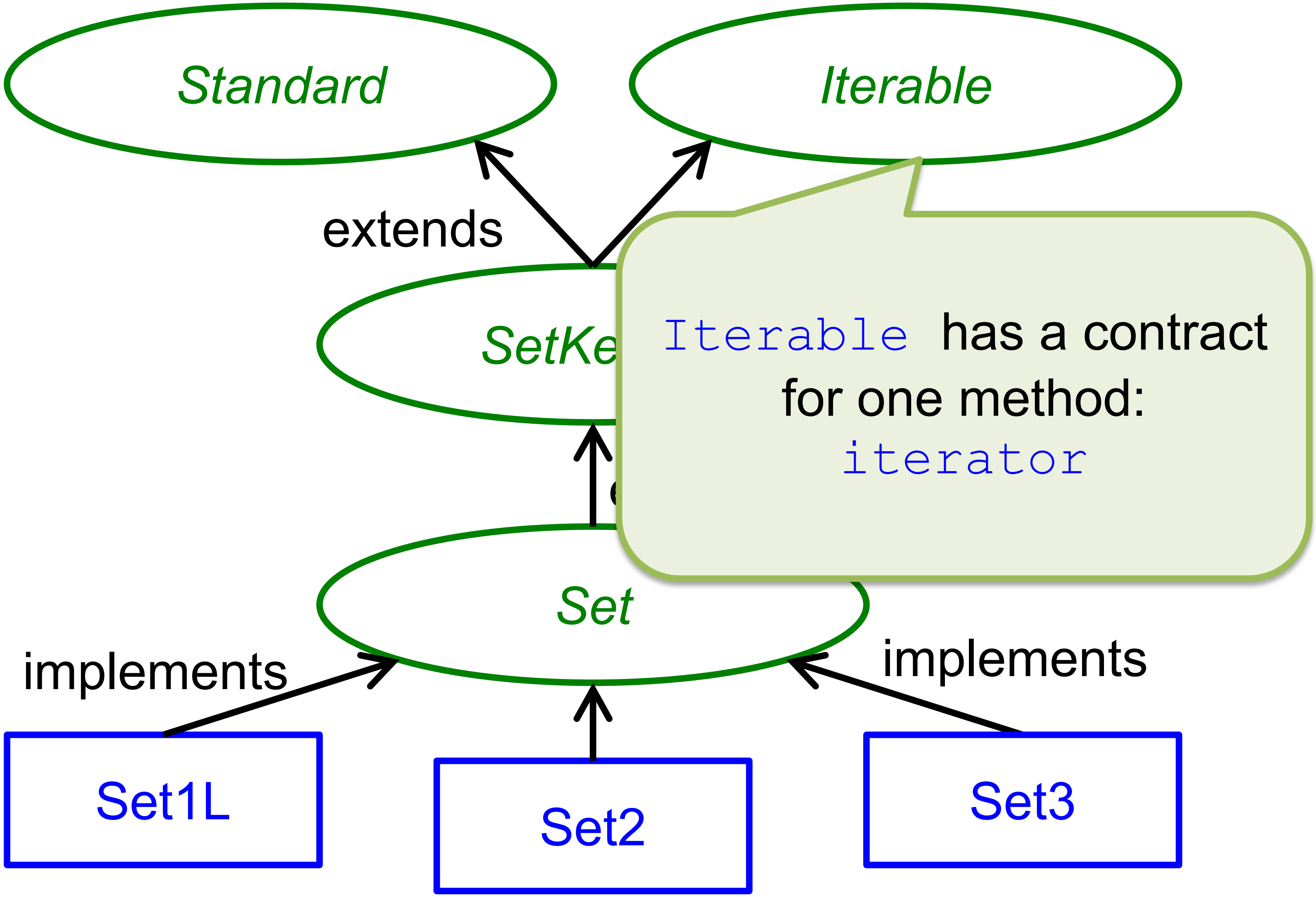
Iterators

- Conventional Java style for iterating over a "collection" like a `Set` is to use an ***iterator*** so you can do this without taking the collection apart and reconstituting it

One More Interface



One More Interface



iterator

`Iterator<T> iterator()`

- Returns an iterator over a set of elements of type `T`.
- Ensures:

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

iterator

`Iterator<T> iterator()`

- Returns an iterator over a set of elements of type `T`.
- Ensures:

  entries(~this.seen ... ***is***
  and
  |~this.seen * ~this

`Iterator` is yet another interface in the Java libraries (in the package `java.util`).

iterator

`Iterator<T> iterat`

- Returns an iterator ov
  of type `T`.
- Ensures:

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

We will return to decipher the contract after seeing the easiest way for this method to be used...

For-Each Loops

- Since `Set<T>` extends the interface `Iterable` (so it inherits the `iterator` method), you may write a ***for-each loop*** to "see" all elements of `Set<T> s` :

```
for (T x : s) {
  // do something with x, but do
  // not call methods on s, or
  // change the value of x or s
}
```

For-E

- Since `Set<T>` e `Iterable` (so it method), you may a ***for-each loop*** to "see" all elements of `Set<T> s` :

```
for (T x : s) {
   // do something with x, but do
   // not call methods on s, or
   // change the value of x or s
}
```

This declares `x` as a local variable of type `T` in the loop; on each iteration, `x` is ***aliased*** to a different element of `s`.

For-Each Loop Example

- Count the number of strings of length 5 in a `Set<String>`:

```
Set<String> dictionary = …
...
int count = 0;
for (String word : dictionary) {
  if (word.length() == 5) {
    count++;
  }
}
```

In Which Order?

- The kernel interface (`SetKernel` in this case) contains the contract for the `iterator` method, as specialized for the type `Set<T>`
- This contract specifies the *order* in which the elements are seen

`iterator` Contract

- Two new ***mathematical variables*** are involved in the contract:
  - The ***`string of`*** *T* called ***`~this.seen`*** contains, in order, those values *already* “seen” in the for-each loop iterations up to any point
  - The ***`string of`*** *T* called ***`~this.unseen`*** contains, in order, those values *not yet* “seen” in the for-each loop iterations up to that point

iterator

`Iterator<T> iterator()`

- Returns an iterator over a set of elements of type `T`.
- Ensures:

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

it

The concatenation of the ***string of*** *T* values already seen and the values not yet seen...

Iterator<T> i

- Returns an itera... s of type T.
- Ensures:

entries(~this.seen * ~this.unseen) = this

and

|~this.seen * ~this.unseen| = |this|

`it`

`Iterator<T> i`

- Returns an itera ... s of type `T`.
- Ensures:

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

The ***finite set of T*** of values already seen and not yet seen...

it

`Iterator<T> i`

- Returns an itera... of type `T`.
- Ensures:

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

The ***finite set of*** *T* of values already seen and not yet seen...
is equal to the entire set ***this***.

`iterator`

`Iterator<T`

What else must be said?
What does the second clause mean?
Why is it important?

- Returns an of type `T`.
- Ensures:

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

Iterating With `iterator`

- The ***for-each*** code has the following properties:
  - It introduces aliases, so you must be careful to “follow the rules”; specifically, the loop body should not call any methods on `s`
  - If what you want to do to each element is to change it (when `T` is a mutable type), then the approach does not work because the loop body should not change `x`
  - It may be more efficient than using `removeAny` (i.e., it also makes no copies of elements of type `T`, though it does use `iterator` methods to carry out the for-each loop, and these could be slow)

Resources

- OSU CSE Components API: `Set`
  - http://web.cse.ohio-state.edu/software/common/doc/
- Java Libraries API: `Iterable` and `Iterator`
  - http://docs.oracle.com/javase/8/docs/api/