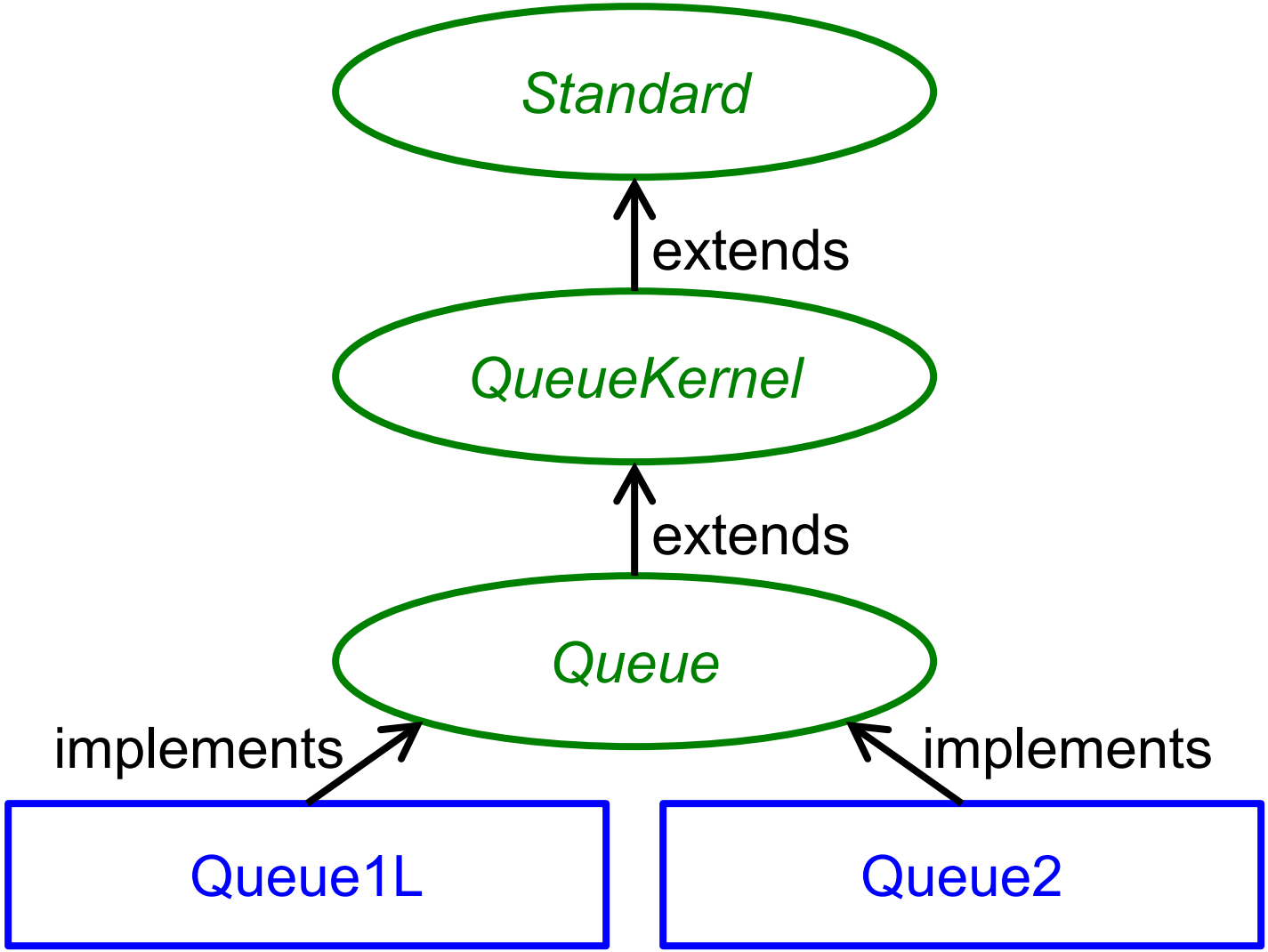
Queue

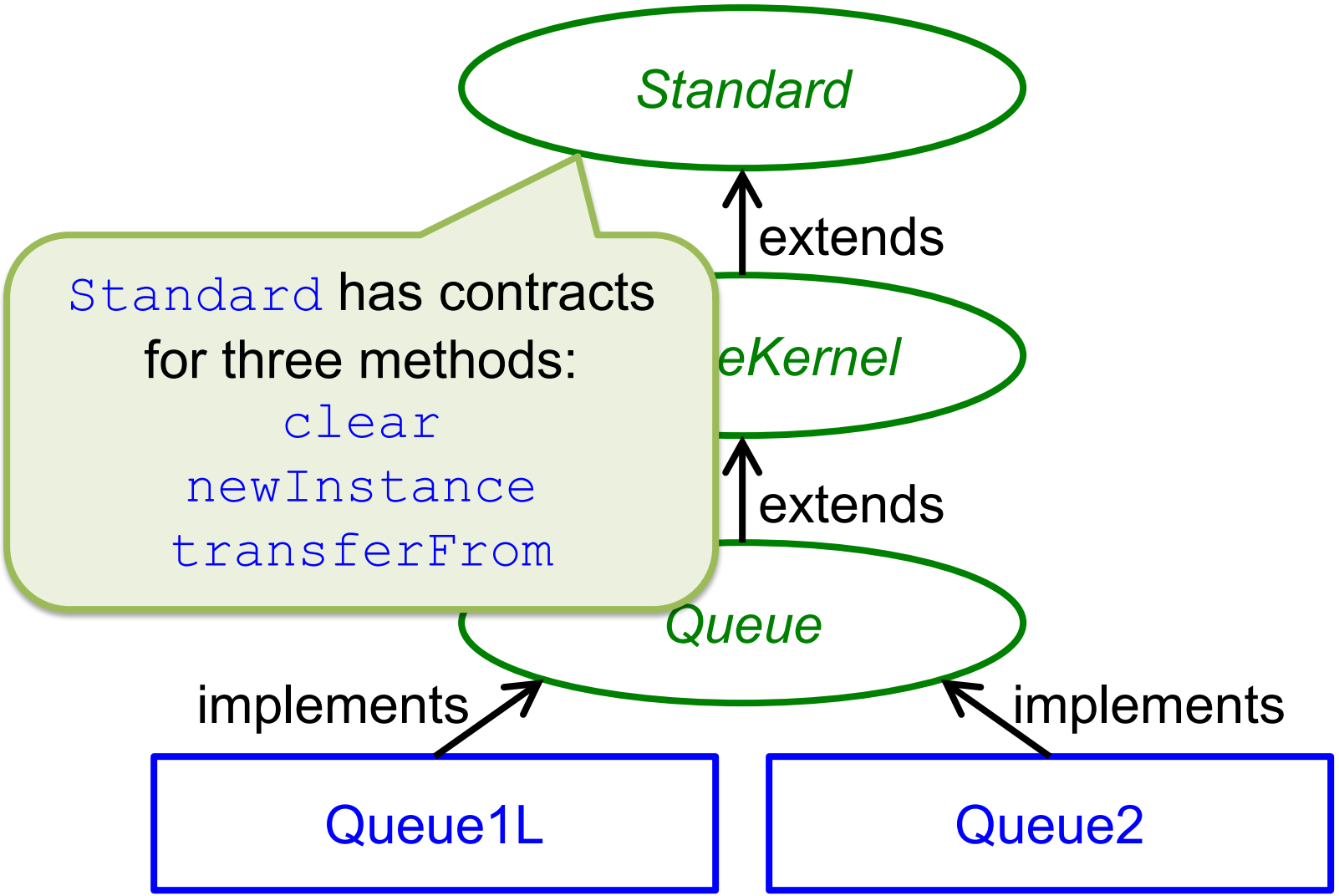
Queue

- The ***Queue*** component family allows you to manipulate strings of entries of any (arbitrary) type in ***FIFO*** (first-in-first-out) order
  - "First" here refers to the *temporal* order in which entries are put into the string and taken out of it, not about the left-to-right or right-to-left order in the string when it is written down

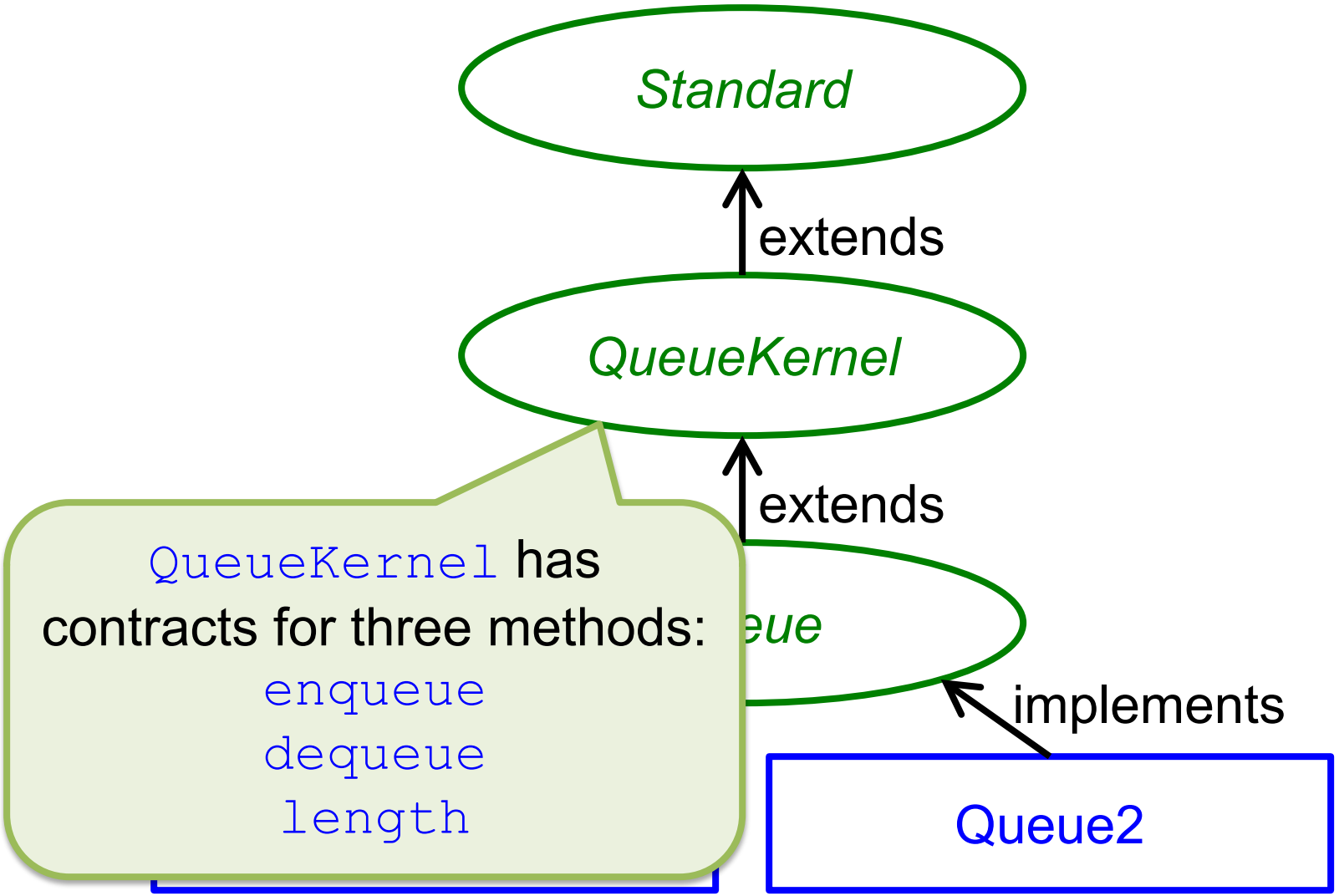
Interfaces and Classes



Interfaces and Classes



Interfaces and Classes

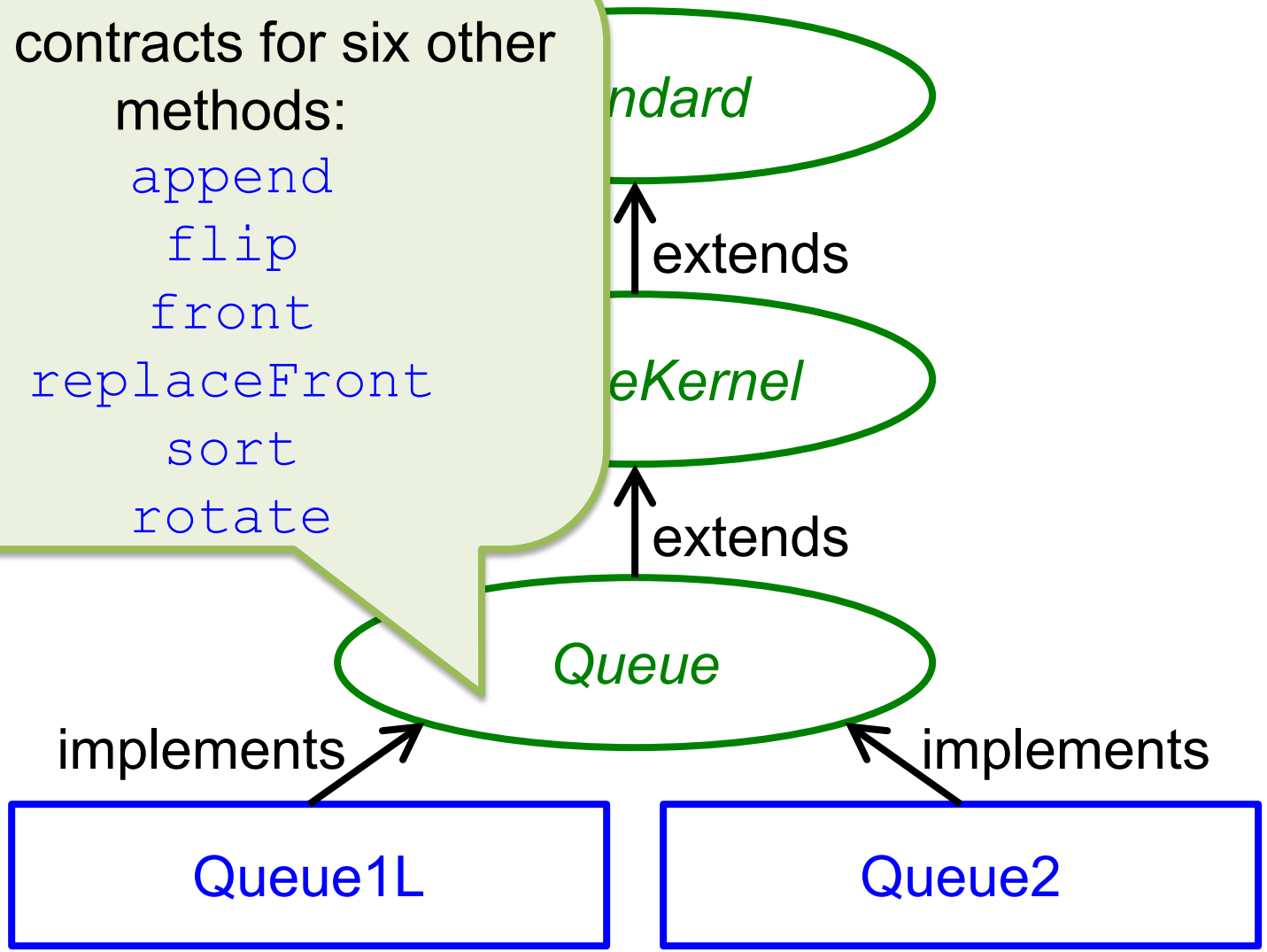


Interfaces and Classes

`Queue`

has contracts for six other methods:

`append`
`flip`
`front`
`replaceFront`
`sort`
`rotate`



Mathematical Model

- The value of a `Queue` variable is modeled as a string of entries of type `T`
- Formally:

```
type Queue is modeled by
    string of T
```

Generics

- Note that `Queue` is a ***generic type*** (also called a ***parameterized type***)
- The actual type of the entries is selected only later by the client when the type `Queue` is used to declare or instantiate a variable, e.g.:

```
Queue<Integer> qi =
   new Queue1L<Integer>();
```

Generics

The ***formal type parameter*** was called `T`; here, the ***actual type*** or ***argument type*** is `Integer`.

a ***generic type*** (also ***ized type***)

ne entries is selected

nt when the type

`Queue` is u o declare or instantiate a variable, e.g.,

```
Queue<Integer> qi =
   new Queue1L<Integer>();
```

Generics

As of Java 7, generic arguments in a constructor call are inferred from the declared type...

a ***generic type*** (also
ized type)

ne entries is selected
nt when the type
`Queue` is u o declare or instantiate a
variable, e.g..

```
Queue<Integer> qi =
   new Queue1L<Integer>();
```

Generics

... so this ***diamond operator*** means the same thing as the constructor with explicit generic arguments.

a ***generic type*** (also
ized type)

ne entries is selected
nt when the type

`Queue` is u o declare or instantiate a variable, e.g..

```
Queue<Integer> qi =
    new Queue1L<>();
```

Wrapper Types

- Note the use of `Integer` here, not **`int`**
- Java demands that generic arguments must be ***reference types***
- Each of the primitive types has a corresponding ***wrapper type*** that is a reference type (in part to satisfy this requirement)

Wrapper Types

primitive type	*wrapper type*
boolean	Boolean
char	Character
int	Integer
double	Double

Wrapper Types

- Each wrapper type is an ***immutable type***
- There is a constructor from the corresponding primitive type
- Java includes features called ***auto-boxing*** and ***auto-unboxing*** so wrapper types can be used with primitive-type syntax *almost* as if they were primitive types
  - Details later (for now, look it up if it seems to matter to your code)

Constructors

- There is one ***constructor*** for each implementation class for `Queue`
- 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 `QueueKernel`)

No-argument Constructor

- Ensures:

  this = < >

Example

Code	State
`Queue<Integer> qi =` `  new Queue1L<>();`	

Example

Code	State
`Queue<Integer> qi =` `  new Queue1L<>();`	
	qi = < >

Methods for `Queue`

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

  ```
  q.methodName(arguments)
  ```

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

enqueue

void enqueue(T x)

- Adds x at the back (right end) of **this**.
- Aliases: reference x
- Updates: **this**
- Ensures:

  this = #***this*** * <x>

enqueue

void enqueue(T x)

- Adds x at the back (right end) of **this**.
- Aliases: reference x
- Updates: **this**
- Ensures:

  this = #***this***

The list of references that might be ***aliases*** upon return from the method is ***advertised*** here, because aliasing is important and otherwise is not specified.

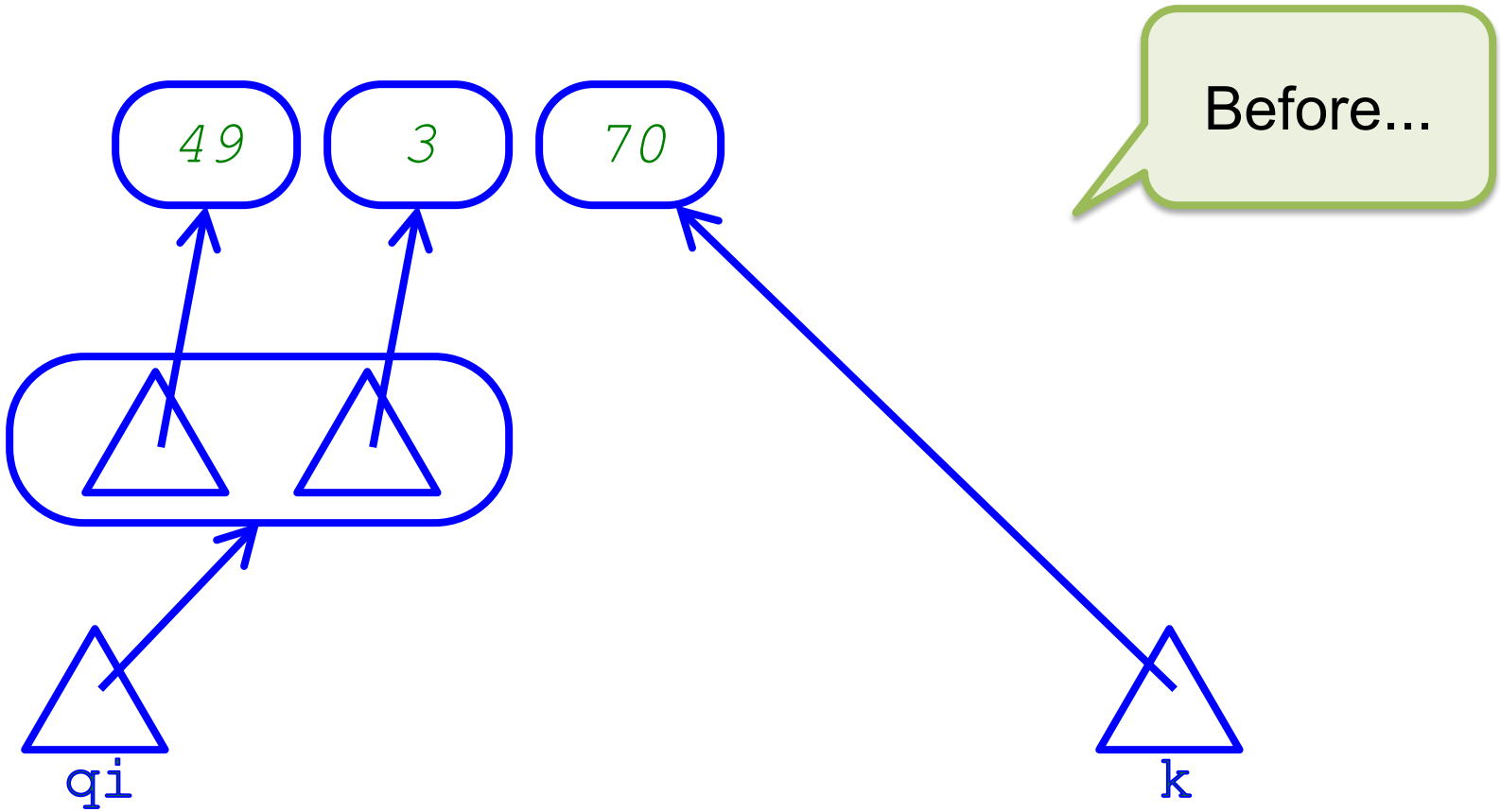
Example

Code	State
	qi = < 49, 3 > *k = 70*
`qi.enqueue(k);`	

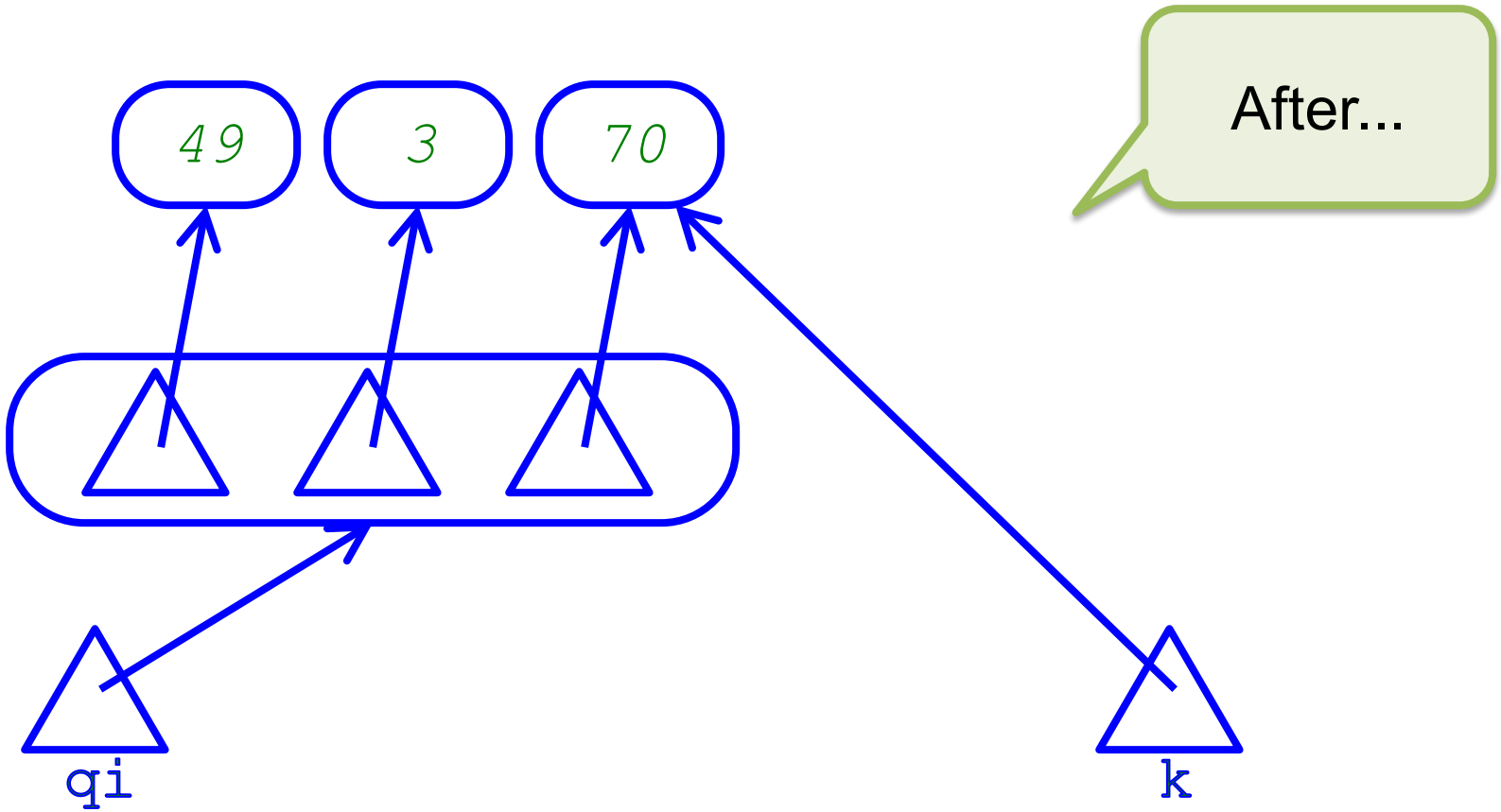
Example

Code	State
	qi = < 49, 3 > *k = 70*
`qi.enqueue(k);`	
	qi = < 49, 3, 70 > *k = 70*

Meaning of “Aliases: …”



Meaning of “Aliases: …”



Meaning of "Aliases: ..."

- The tracing table notation with → gives us no easy way to describe this situation
  - The picture is, however, a handy way to see what's going on, so draw pictures!
- Since `Integer` is immutable, there is ***no consequence*** to this case of aliasing
  - But consider:

```
Queue<NaturalNumber> qn = ...
```

dequeue

`T dequeue()`

- Removes and returns the entry at the front (left end) of **`this`**.
- Updates: **`this`**
- Requires:

  `this` *`/= < >`*
- Ensures:

  *`#`****`this`*** *`= <dequeue> *`* ***`this`***

Example

Code	State
	qi = < 49, 3, 70 > *k = -584*
`k = qi.dequeue();`	

Example

Code	State
	qi = < 49, 3, 70 > *k = -584*
`k = qi.dequeue();`	
	qi = < 3, 70 > *k = 49*

`length`

`int` `length()`

- Reports the length of **`this`**.
- Ensures:

  `length` = |***`this`***|

Example

Code	State
	qi = < 49, 3, 70 > *n = -45843*
`n = qi.length();`	

Example

Code	State
	qi = < 49, 3, 70 > *n = -45843*
`n = qi.length();`	
	qi = < 49, 3, 70 > *n = 3*

front

`T front()`

- Returns the entry at the the front (left end) of **`this`**.
- Aliases: reference returned by `front`
- Requires:

  `this` *`/= < >`*
- Ensures:

  `<front>` ***`is prefix of this`***

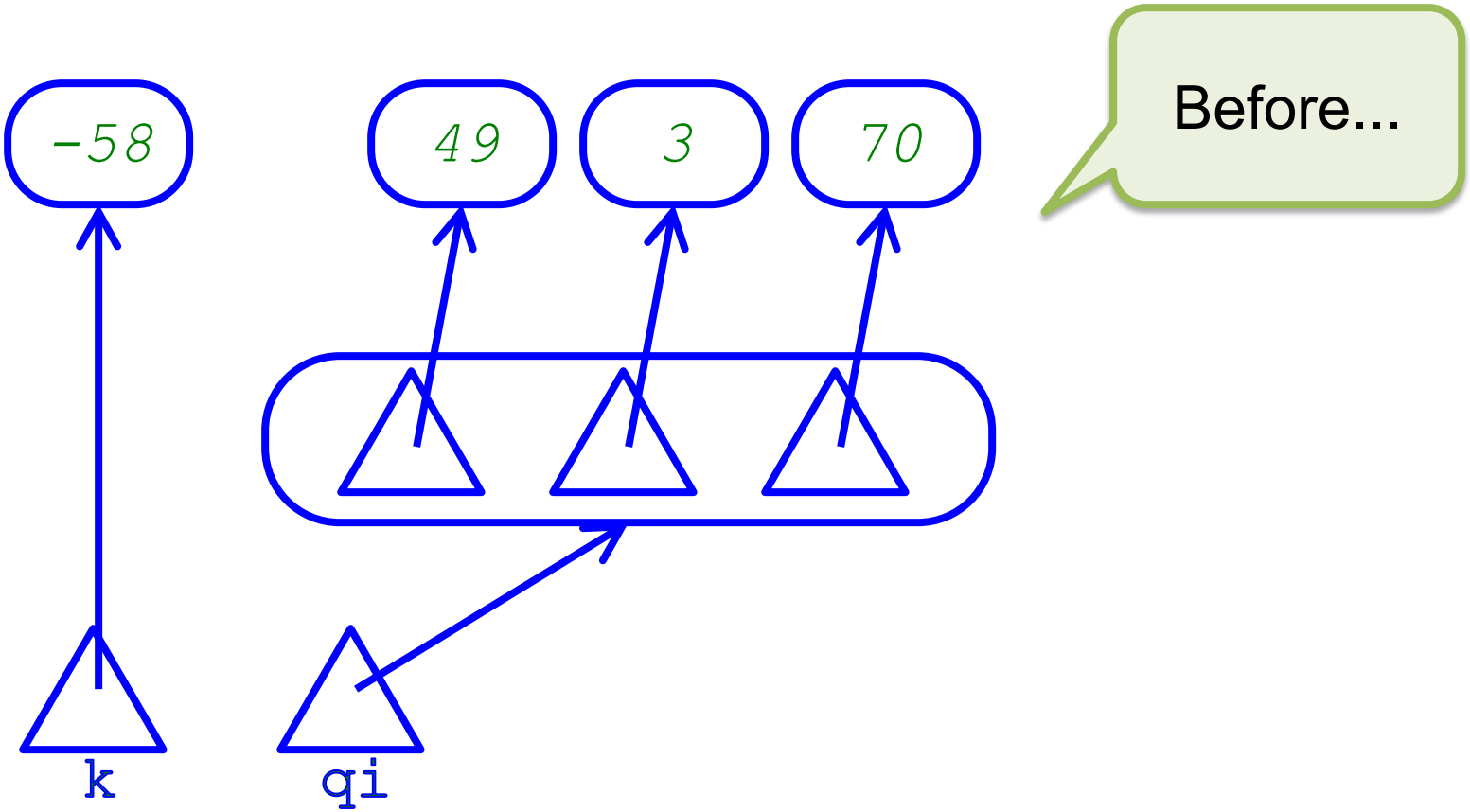
Example

Code	State
	qi = < 49, 3, 70 > *k = -58*
`k = qi.front();`	

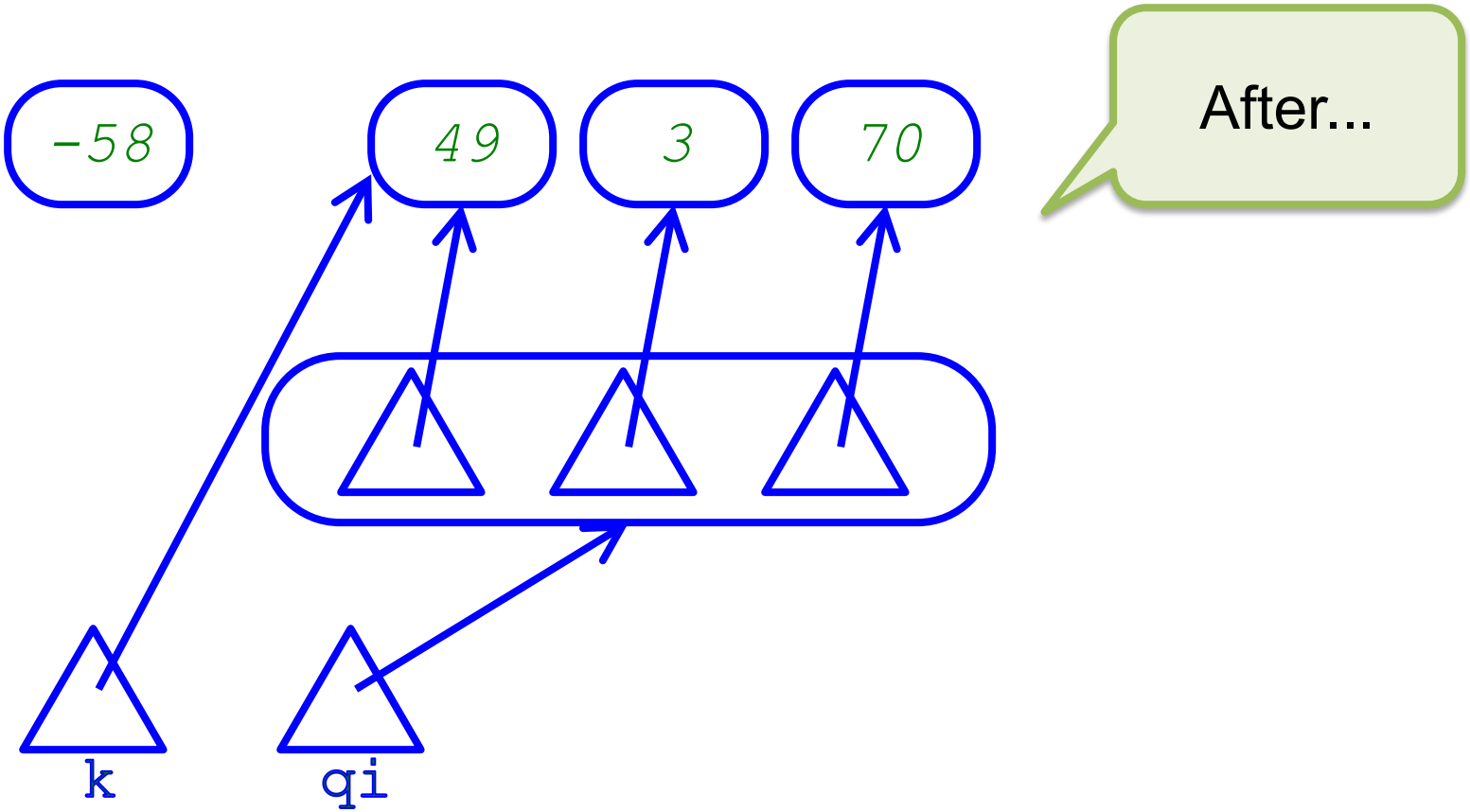
Example

Code	State
	qi = < 49, 3, 70 > *k = -58*
`k = qi.front();`	
	qi = < 49, 3, 70 > *k = 49*

Meaning of “Aliases: …”



Meaning of “Aliases: …”



replaceFront

`T replaceFront(T x)`

- Replaces the front of **`this`** with `x`, and returns the old front.
- Aliases: reference `x`
- Updates: **`this`**
- Requires:

  this */= < >*
- Ensures:

  <replaceFront> ***is prefix of #this and***

  this *= <x>* * ***#this****[1, |****#this****|)*

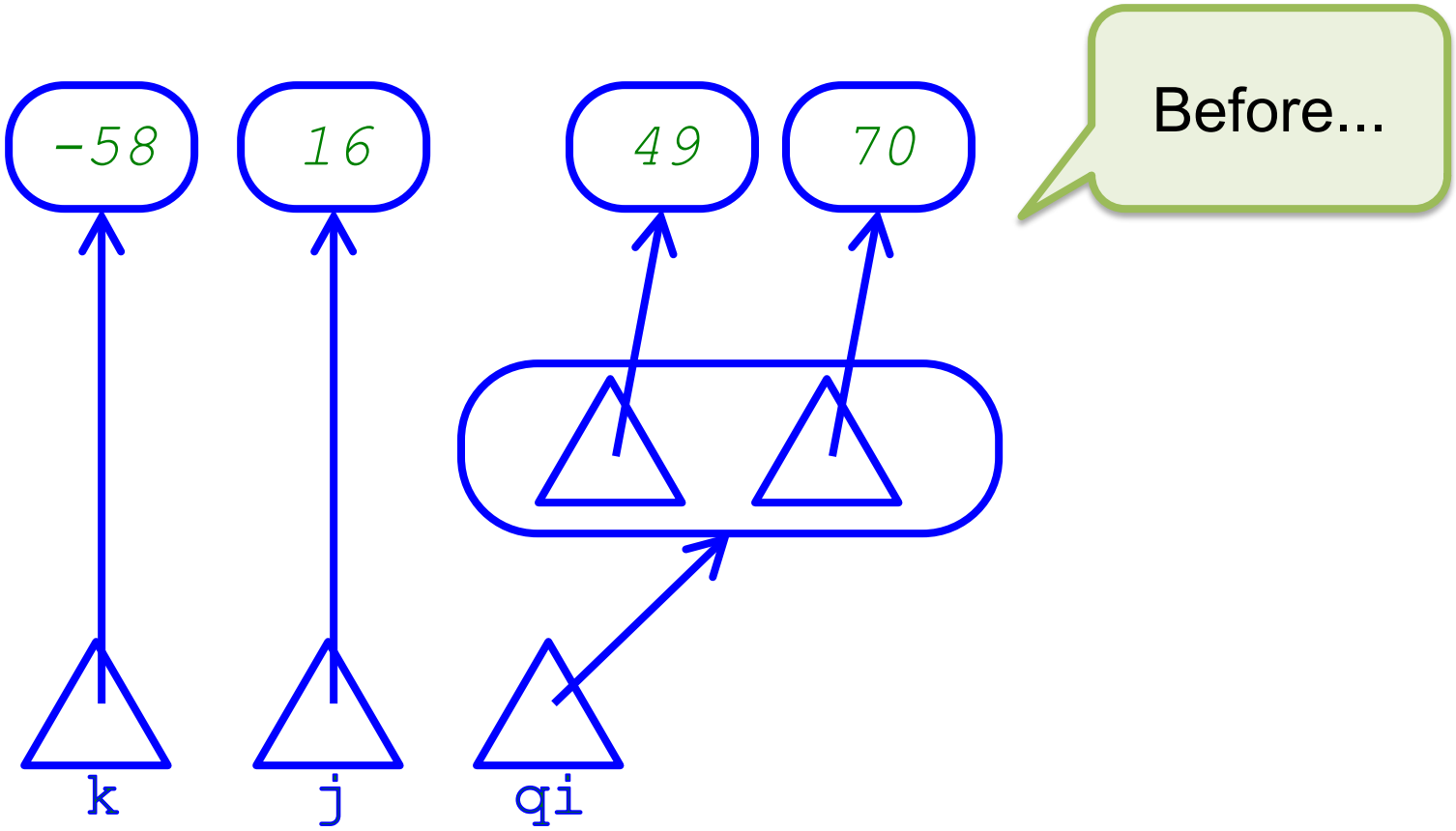
Example

Code	State
	qi = < 49, 70 > *k = -58* *j = 16*
`k = qi.replaceFront(j);`	

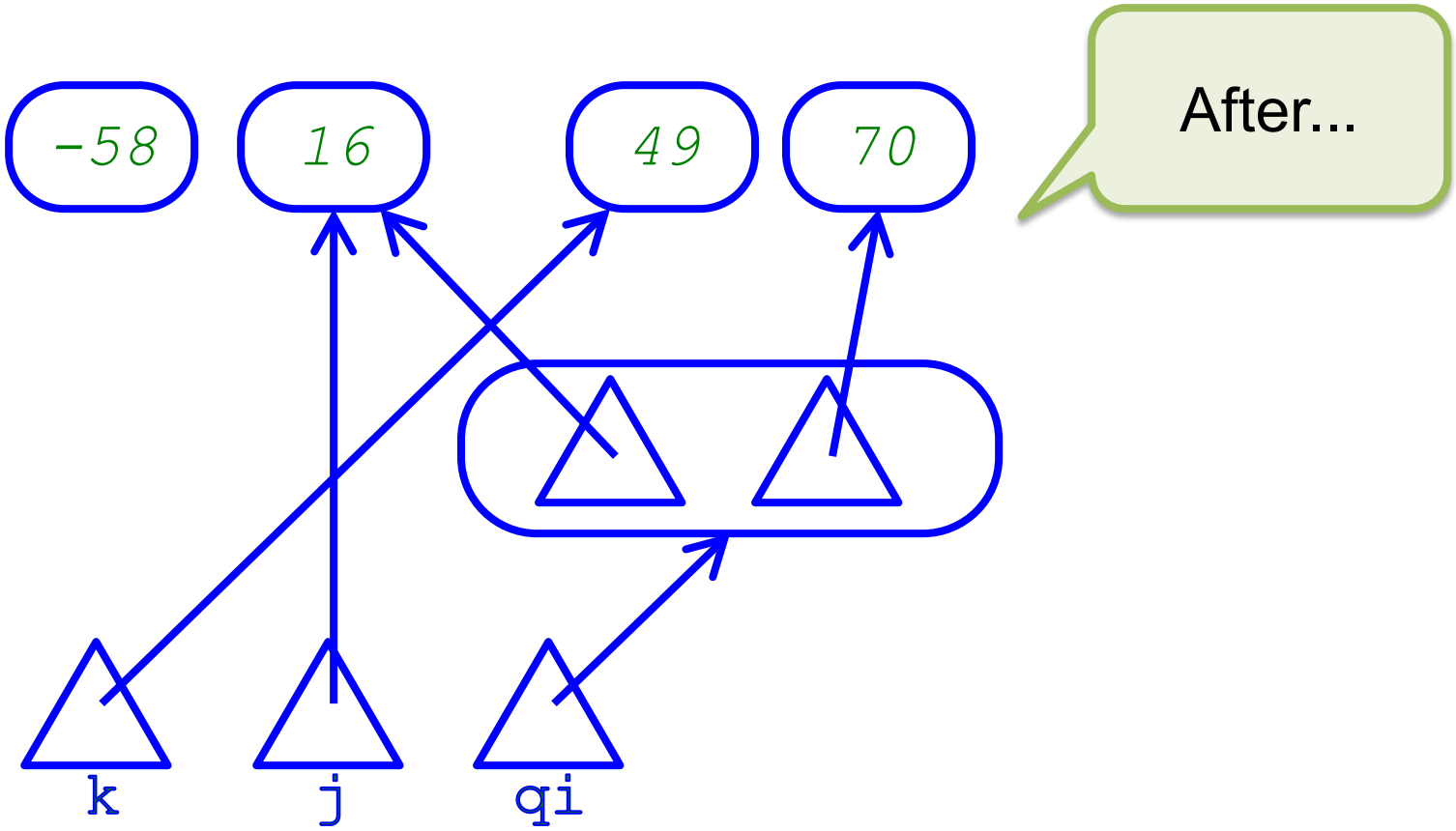
Example

Code	State
	qi = < 49, 70 > *k = -58* *j = 16*
`k = qi.replaceFront(j);`	
	qi = < 16, 70 > *k = 49* *j = 16*

Meaning of “Aliases: …”



Meaning of “Aliases: …”



Another Example

Code	State
	qi = < 49, 70 > *j = 16*
`j = qi.replaceFront(j);`	

Another Example

Code	State
	qi = < 49, 70 > *j = 16*
`j = qi.replaceFront(j);`	
	qi = < 16, 70 > *j = 49*

Another Example

This use of the method avoids creating an alias: it ***swaps*** `j` with the entry previously at the front.

	State
	qi = < 49, 70 > *j = 16*
`j = qi.replaceFront(j);`	
	qi = < 16, 70 > *j = 49*

append

void append(Queue<T> q)

- Concatenates ("appends") q to the end of **this**.
- Updates: **this**
- Clears: q
- Ensures:

  this = #***this*** * #q

Example

Code	State
	q1 = < 4, 3, 2 > *q2 = < 1, 0 >*
`q1.append(q2);`	

Example

Code	State
	q1 = < 4, 3, 2 > *q2 = < 1, 0 >*
`q1.append(q2);`	
	q1 = < 4, 3, 2, 1, 0 > *q2 = < >*

flip

void flip()

- Reverses ("flips") **this**.
- Updates: **this**
- Ensures:

this = ***rev***(#***this***)

Example

Code	State
	qi = < 18, 6, 74 >
`qi.flip();`	

Example

Code	State
	qi = < 18, 6, 74 >
`qi.flip();`	
	qi = < 74, 6, 18 >

sort

void sort(Comparator<T> order)

- Sorts **this** according to the ordering provided by the compare method from order.
- Updates: **this**
- Requires:

  [the relation computed by order.compare is a total preorder]

- Ensures:

  this *= [#**this** ordered by the relation computed by order.compare]*

Comparators

- The Java interface `Comparator<T>` is:

```
public interface Comparator<T> {
   /**
    * Returns a negative integer, zero, or
    * a positive integer as the first
    * argument is less than, equal to, or
    * greater than the second.
    */
   int compare(T o1, T o2);
}
```

Comparators

- The notion of “less than” and “greater than” is quite flexible
- The notion of “equal to” is not flexible
  - It is based on mathematical equality, not on a flexible notion of being “equivalent to”
- There are important technicalities...

sort

A ***total preorder*** means that any two values of type `T` are comparable, and that there are no "cycles", e.g., nothing like *`a < b < c < a`*.

`> order)`

ordering provided

n `order`.

- Requires:

  `[the relatio computed by order.compare is a total preorder]`

- Ensures:

  `this` *`= [#`* ***`this`*** *`ordered by the relation computed by order.compare]`*

Creating a Comparator

```
private static class IntegerLT
    implements Comparator<Integer> {
  @Override
  public int compare(Integer o1, Integer o2) {
    if (o1 < o2) {
      return -1;
    } else if (o1 > o2) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

Creating a Comparator

```
private static class IntegerLT
    implements Comparator<Integer> {
  @Override
  public int compare(Integer o1, Integer o2) {
    if (o1 < o2)
      return -1;
    } else if (o1 > o2)
      return 1;
    } else {
      return 0;
    }
  }
}
```

A class that implements `Comparator` is usually a ***nested class*** (i.e., declared for local use inside another class), and if so should be declared **`private static`**.

An Easy Comparator

```
private static class IntegerLT
    implements Comparator<Integer> {
  @Override
  public int compare(Integer o1, Integer o2) {
    return o1.compareTo(o2);
  }
}
```

Since a generic parameter must be a ***reference*** type, and each ***wrapper*** type `T` (here, `Integer`) implements the interface `Comparable<T>`, each has a `compareTo` method that can be called like this to simplify the code for `compare` in a `Comparator<T>` implementation.

Example

Code	State
	qi = < 8, 6, 92, 1 >

`Comparator<Integer> ci =`
`  new IntegerLT ();`
`qi.sort(ci);`	

Example

Code	State
	qi = < 8, 6, 92, 1 >

`Comparator<Integer> ci =`
`  new IntegerLT ();`
`qi.sort(ci);`	

	qi = < 1, 6, 8, 92 >

rotate

void rotate(**int** distance)

- Rotates **this**.
- Updates: **this**
- Ensures:

```
if #this = <> then
  this = #this
else
  this =
    #this[distance mod |#this|, |#this|) *
    #this[0, distance mod |#this|)
```

Example

Code	State
	qi = < 8, 6, 92, 3 >
`qi.rotate(1);`	

Example

Code	State
	qi = < 8, 6, 92, 3 >
`qi.rotate(1);`	
	qi = < 6, 92, 3, 8 >

Example

Code	State
	qi = < 8, 6, 92, 3 >
`qi.rotate(3);`	

Example

Code	State
	qi = < 8, 6, 92, 3 >
`qi.rotate(3);`	
	qi = < 3, 8, 6, 92 >

Example

Code	State
	qi = < 8, 6, 92, 3 >
`qi.rotate(-1);`	

Example

Code	State
	qi = < 8, 6, 92, 3 >
`qi.rotate(-1);`	
	qi = < 3, 8, 6, 92 >

Resources

- OSU CSE Components API: `Queue`
  - http://web.cse.ohio-state.edu/software/common/doc/
- Java Libraries API: `Comparator`, `Comparable`
  - http://docs.oracle.com/javase/8/docs/api/