# Imperative Programming 2:
# Iterator

Hongseok Yang
University of Oxford

# Linear search from Imperative Prog. I.

```
def find(a: Array[Int], p: Int=>Boolean): Option[Int] = {
  var i = 0
  while (i < a.length) {
    val x = a(i); i += 1
    if (p(x)) return Some(x)
  }
  None
}
```

Linear search from Imperative Prog. I.

Enumerates all elements.

```
def find(a: Array[Int], p: Int=>Boolean): Option[Int] = {
  var i = 0
  while (i < a.length) {
    val x = a(i); i += 1
    if (p(x)) return Some(x)
  }
  None
}
```

Linear search from Imperative Prog. I.

Enumerates all elements.

Stops as soon as a desired element is found.

```
def find(a: Array[Int], p: Int=>Boolean): Option[Int] = {
  var i = 0
  while (i < a.length) {
    val x = a(i); i += 1
    if (p(x)) return Some(x)
  }
  None
}
```

Linear search from Imperative Prog. I.

Enumerates all elements.

Stops as soon as a desired element is found.

```
def find(a: Array[Int], p: Int=>Boolean): Option[Int] = {
  var i = 0
  while (i < a.length) {
    val x = a(i); i += 1
    if (p(x)) return Some(x)
  }
  None
}
```

**Conceptually applicable to all collections**

Linear search from Imperative Prog. I.

Enumerates all elements.

Stops as soon as a desired element is found.

```
def find(a: List[Int], p: Int=>Boolean): Option[Int] = {
  var i = 0
  while (i < a.length) {
    val x = a(i); i += 1
    if (p(x)) return Some(x)
  }
  None
}
```

Conceptually applicable to all collections

Linear search from Imperative Prog. I.

Enumerates all elements.

Stops as soon as a desired element is found.

```
def find(a: List[Int], p: Int=>Boolean): Option[Int] = {
  var i = 0
  while (i < a.length) {
    val x = a(i); i += 1
    if (p(x)) return Some(x)
  }
  None
}
```

Conceptually applicable to all collections

[Q] Any problem here?

Linear search from Imperative Prog. I.

Enumerates all elements.

Stops as soon as a desired element is found.

```
def find(a: List[Int], p: Int=>Boolean): Option[Int] = {
  var i = 0
  while (i < a.length) {
    val x = a(i); i += 1
    if (p(x)) return Some(x)
  }
  None
}
```

Conceptually applicable to all collections

[Q] Any problem here?

[A] $O(n^2)$.

Linear search from Imperative Prog. I.

Enumerates all elements.

Stops as soon as a desired element is found.

```
def find(a: Set[Int], p: Int=>Boolean): Option[Int] = {
  var i = 0
  while (i < a.length) {
    val x = a(i); i += 1
    if (p(x)) return Some(x)
  }
  None
}
```

Conceptually applicable to all collections

[Q] Any problem here?

[A] $O(n^2)$. For sets, even this is not possible.

Linear search from Imperative Prog. 1.

Enumerates all elements.

Stops as soon as a desired element is found.

```
def find(a: Set[Int], p: Int=>Boolean): Option[Int] = {
  var i = 0
  while (i < a.length) {
    val x = a(i); i += 1
    if (p(x)) return Some(x)
  }
  None
}
```

Conceptually applicable to all collections

[Q] A

[A] C

```
hyang — bash — 75×15
/Users/hyang/find3.scala:3: error: value length is not a member of Set[Int]
  while (i < a.length) {
               ^
/Users/hyang/find3.scala:5: error: type mismatch;
 found   : Boolean
 required: Int
    if (p(x)) return Some(x)
          ^
/Users/hyang/find3.scala:5: error: type mismatch;
 found   : Boolean
 required: Int
    if (p(x)) return Some(x)
                      ^
three errors found
Hongseoks-MacBook-Pro:~ hyang$
```

```
def find(a: Array[Int], p: Int=>Boolean): Option[Int] = {
  var i = 0
  while (i < a.length) {
    val x = a(i); i += 1
    if (p(x)) return Some(x)
  }
  None
}
```

Iterators are abstraction for supporting efficient enumeration and early stopping.

```
def find(a: Array[Int], p: Int=>Boolean): Option[Int] = {
  var i = 0
  while (i < a.length) {
    val x = a(i); i += 1
    if (p(x)) return Some(x)
  }
  None
}
```

Iterators are abstraction for supporting efficient enumeration and early stopping.

```
def find(a: Array[Int], p: Int=>Boolean): Option[Int] = {
  var it = a.iterator
  while (it.hasNext) {
    val x = it.next()
    if (p(x)) return Some(x)
  }
  None
}
```

Iterators are abstraction for supporting efficient enumeration and early stopping.

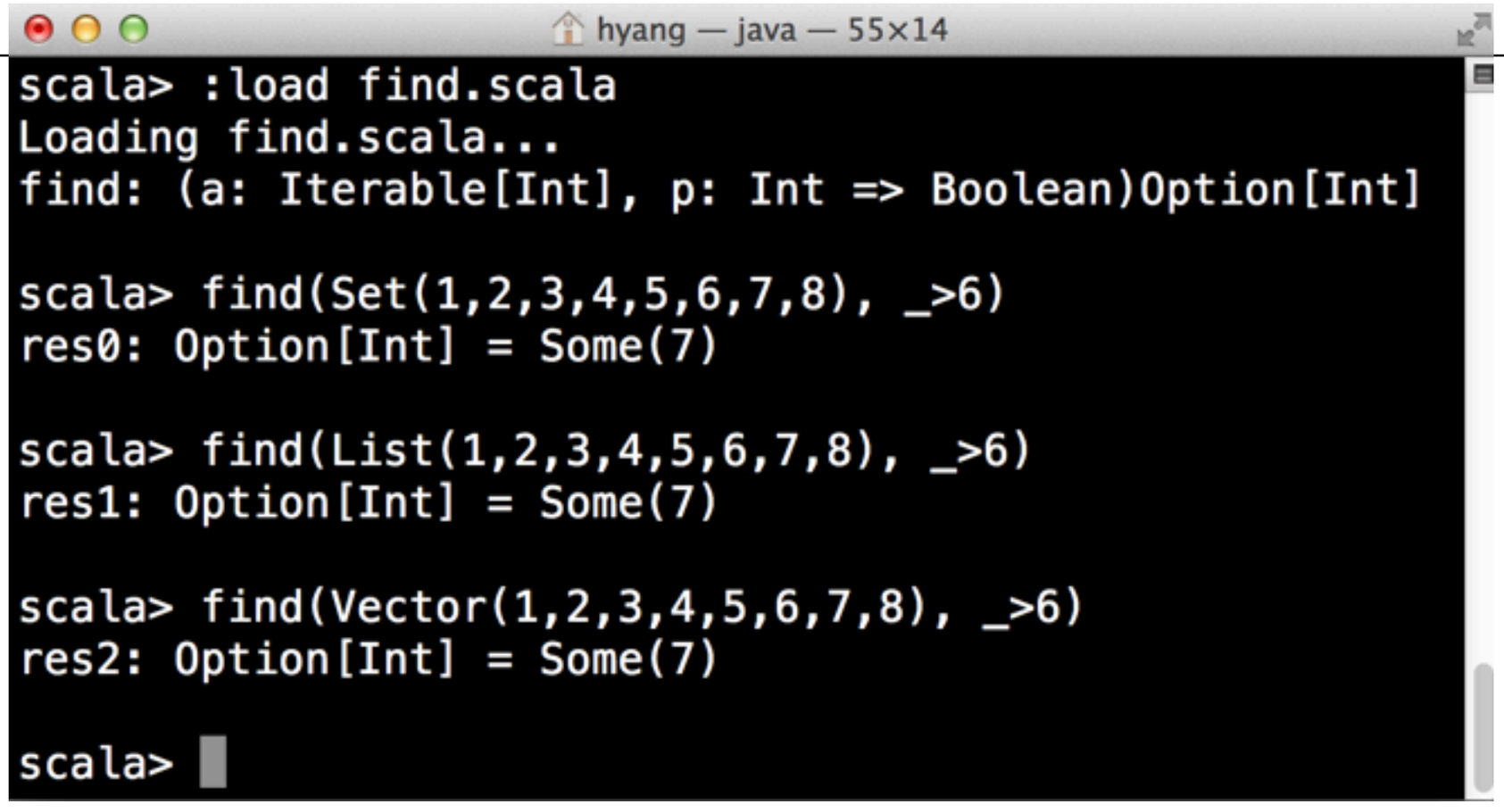
```
def find(a: Iterable[Int], p: Int=>Boolean): Option[Int] = {
  var it = a.iterator
  while (it.hasNext) {
    val x = it.next()
    if (p(x)) return Some(x)
  }
  None
}
```

The linear search works for (almost) all collections.

Iterators are abstraction for supporting efficient enumeration and early stopping.

```
def find(a: Iterable[Int], p: Int=>Boolean): Option[Int] = {
  var it = a.iterator
  while (it.hasNext) {
    val x = it.next()
    if (p(x)) return Some(x)
  }
  None
}
```

The linear search works for (almost) all collections.



# Learning outcome

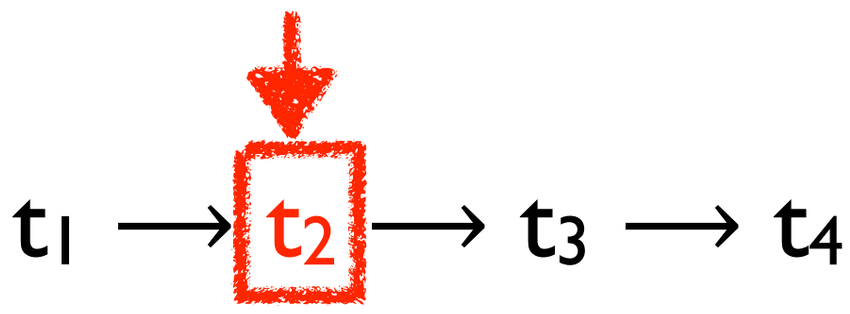
- Can explain what the iterator abstraction is and why it is useful.
- Can develop iterators.
- Can write programs using iterators.

# Iterator for T objects

- An object with (at least) following methods:

```
def hasNext : Boolean
def next() : T
```

- Intuition 1 -- a generator of elements of type T.
- Intuition 2 -- a sequence of T objects with the current position.



Odd numbers

$1 \longrightarrow 3 \longrightarrow 5 \longrightarrow 7 \longrightarrow 9 \ldots. \longrightarrow 21$

Odd numbers

1 → 3 → 5 → 7 → 9 .... → 21

```
class Odd(private val uBound: Int) {
  private var v = 1
  def hasNext: Boolean = (v <= uBound)
  def next(): Int =
    if (v <= uBound) { val t=v; v += 2; t }
    else { throw new NoSuchElementException }
}
```

Odd numbers

$1 \longrightarrow 3 \longrightarrow 5 \longrightarrow 7 \longrightarrow 9 \ldots. \longrightarrow 21$

```
class Odd(private val uBound: Int) {
  private var v = 1
  def hasNext: Boolean = (v <= uBound)
  def next(): Int =
    if (v <= uBound) { val t=v; v += 2; t }
    else { throw new NoSuchElementException }
}
```

[Q] Build an iterator for Fibo. seq.

$0 \longrightarrow 1 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 5 \longrightarrow 8 \ldots$

$x_0 = 0 \quad x_1 = 1$

$x_{n+2} = x_n + x_{n+1}$

Odd numbers

$1 \longrightarrow 3 \longrightarrow 5 \longrightarrow 7 \longrightarrow 9 \ldots. \longrightarrow 21$

```
class Odd(private val uBound: Int) {
  private var v = 1
  def hasNext: Boolean = (v <= uBound)
  def next(): Int =
    if (v <= uBound) { val t=v; v += 2; t }
    else { throw new NoSuchElementException }
}
```

[Q] Build an iterator for Fibo. seq.

$x_0 = 0 \quad x_1 = 1$

$x_{n+2} = x_n + x_{n+1}$

$0 \longrightarrow 1 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 5 \longrightarrow 8 \ldots$

```
class Fibonacci {


  def hasNext: Boolean = ...
  def next(): Int = ...




}
```

Odd numbers

$1 \longrightarrow 3 \longrightarrow 5 \longrightarrow 7 \longrightarrow 9 \ldots. \longrightarrow 21$

```
class Odd(private val uBound: Int) {
  private var v = 1
  def hasNext: Boolean = (v <= uBound)
  def next(): Int =
    if (v <= uBound) { val t=v; v += 2; t }
    else { throw new NoSuchElementException }
}
```

[Q] Build an iterator for Fibo. seq.

$x_0 = 0 \quad x_1 = 1$

$x_{n+2} = x_n + x_{n+1}$

$0 \longrightarrow 1 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 5 \longrightarrow 8 \ldots$

```
class Fibonacci {
  private var v0 = 0
  private var v1 = 1
  def hasNext: Boolean = true
  def next(): Int = {
    val res = v0
    v0 = v1; v1 += res
    res
  }
}
```

Odd numbers

$1 \longrightarrow 3 \longrightarrow 5 \longrightarrow 7 \longrightarrow 9 \ldots. \longrightarrow 21$

```
class Odd(private val uBound: Int)
        extends Iterator[Int] {
  private var v = 1
  def hasNext: Boolean = (v <= uBound)
  def next(): Int =
    if (v <= uBound) { val t=v; v += 2; t }
    else { throw new NoSuchElementException }
}
```

[Q] Build an iterator for Fibo. seq.

$x_0 = 0 \quad x_1 = 1$

$x_{n+2} = x_n + x_{n+1}$

$0 \longrightarrow 1 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 5 \longrightarrow 8 \ldots$

```
class Fibonacci extends Iterator[Int] {
  private var v0 = 0
  private var v1 = 1
  def hasNext: Boolean = true
  def next(): Int = {
    val res = v0
    v0 = v1; v1 += res
    res
  }
}
```

~50 methods for free.

# Recipe for equipping a collection class with iterator

1. Add a method for creating an iterator.
2. Typically, the method uses an inner class, which can access private fields of the outer class.

```
class Stack {
  private[this] var i: Int = -1
  private[this] var s: Array[Int] = new Array(100)
  def push(x: Int) { i += 1; s(i) = x }
  def pop(): Int = { i -= 1; s(i+1) }




}
```

# Recipe for equipping a collection class with iterator

1. Add a method for creating an iterator.
2. Typically, the method uses an inner class, which can access private fields of the outer class.

```
class Stack {
  private[this] var i: Int = -1
  private[this] var s: Array[Int] = new Array(100)
  def push(x: Int) { i += 1; s(i) = x }
  def pop(): Int = { i -= 1; s(i+1) }
  def iterator : Iterator[Int] = ...



}
```

# Recipe for equipping a collection class with iterator

1. Add a method for creating an iterator.
2. Typically, the method uses an inner class, which can access private fields of the outer class.

```
class Stack {
  private[this] var i: Int = -1
  private[this] var s: Array[Int] = new Array(100)
  def push(x: Int) { i += 1; s(i) = x }
  def pop(): Int = { i -= 1; s(i+1) }
  def iterator : Iterator[Int] = new Iterator[Int] {

    def hasNext = ...
    def next() = ...
  }
}
```

# Recipe for equipping a collection class with iterator

1. Add a method for creating an iterator.
2. Typically, the method uses an inner class, which can access private fields of the outer class.

```
class Stack {
  private[this] var i: Int = -1
  private[this] var s: Array[Int] = new Array(100)
  def push(x: Int) { i += 1; s(i) = x }
  def pop(): Int = { i -= 1; s(i+1) }
  def iterator : Iterator[Int] = new Iterator[Int] {

    def hasNext = ...
    def next() = ...
  }
}
```

[Q] Complete the def'n.

# Recipe for equipping a collection class with iterator

1. Add a method for creating an iterator.
2. Typically, the method uses an inner class, which can access private fields of the outer class.

```
class Stack {
  private[this] var i: Int = -1
  private[this] var s: Array[Int] = new Array(100)
  def push(x: Int) { i += 1; s(i) = x }
  def pop(): Int = { i -= 1; s(i+1) }
  def iterator : Iterator[Int] = new Iterator[Int] {
    var j: Int = i
    def hasNext = (j >= 0)
    def next() = { j -= 1; s(j+1) }
  }
}
```

[Q] Complete the def'n.

# Recipe for equipping a collection class with iterator

3. A collection and an iterator share mutable state.
4. Assumption -- a collection does not get modified while it is enumerated by an iterator.

```
class Stack {
  private[this] var i: Int = -1
  private[this] var s: Array[Int] = new Array(100)
  def push(x: Int) { i += 1; s(i) = x }
  def pop(): Int = { i -= 1; s(i+1) }
  def iterator : Iterator[Int] = new Iterator[Int] {
    var j: Int = i
    def hasNext = (j >= 0)
    def next() = { j -= 1; s(j+1) }
  }
}
```

```
val st = new Stack; st.push(1)
val it = st.iterator
st.pop(); st.push(2); st.push(3)
println(it.next())



2
```

3. A collection and an iterator share mutable state.

4. Assumption -- a collection does not get modified while it is enumerated by an iterator.

```
class Stack {
  private[this] var i: Int = -1
  private[this] var s: Array[Int] = new Array(100)
  def push(x: Int) { i += 1; s(i) = x }
  def pop(): Int = { i -= 1; s(i+1) }
  def iterator : Iterator[Int] = new Iterator[Int] {
    var j: Int = i
    def hasNext = (j >= 0)
    def next() = { j -= 1; s(j+1) }
  }
}
```

```
val st = new Stack; st.push(1)
val it = st.iterator
st.pop(); st.push(2); st.push(3)
println(it.next())



2
```

3. A collection and an iterator share mutable state.

4. Assumption -- a collection does not get modified while it is enumerated by an iterator.

[Q] Fix this.

```
class Stack {
  private[this] var i: Int = -1
  private[this] var s: Array[Int] = new Array(100)
  def push(x: Int) { i += 1; s(i) = x }
  def pop(): Int = { i -= 1; s(i+1) }
  def iterator : Iterator[Int] = new Iterator[Int] {
    var j: Int = i
    def hasNext = (j >= 0)
    def next() = { j -= 1; s(j+1) }
  }
}
```

```
val st = new Stack; st.push(1)
val it = st.iterator
st.pop(); st.push(2); st.push(3)
println(it.next())



2
```

3. A collection and an iterator share mutable state.

4. Assumption -- a collection does not get modified while it is enumerated by an iterator.

[Q] Fix this.

```
class Stack {
  private[this] var i: Int = -1
  private[this] var s: Array[Int] = new Array(100)
  def push(x: Int) { i += 1; s(i) = x }
  def pop(): Int = { i -= 1; s(i+1) }
  def iterator : Iterator[Int] = new Iterator[Int] {
    var j: Int = i
    def hasNext = (j >= 0)
    def next() = { j -= 1; s(j+1) }
  }
}
```

[A] Version number.

```
val st = new Stack; st.push(1)
val it = st.iterator
st.pop(); st.push(2); st.push(3)
println(it.next())



2
```

3. A collection and an iterator share mutable state.

4. Assumption -- a collection does not get modified while it is enumerated by an iterator.

[Q] Fix this.

```
class Stack {
  private[this] var i: Int = -1; private[this] var v: Int = 0
  private[this] var s: Array[Int] = new Array(100)
  def push(x: Int) { v += 1; i += 1; s(i) = x }
  def pop(): Int = { v += 1; i -= 1; s(i+1) }
  def iterator : Iterator[Int] = new Iterator[Int] {
    var j: Int = i
    def hasNext = j >= 0
    def next() = { j -= 1; s(j+1) }
  }
}
```

[A] Version number.

```
val st = new Stack; st.push(1)
val it = st.iterator
st.pop(); st.push(2); st.push(3)
println(it.next())



2
```

3. A collection and an iterator share mutable state.

4. Assumption -- a collection does not get modified while it is enumerated by an iterator.

[Q] Fix this.

```
class Stack {
  private[this] var i: Int = -1; private[this] var v: Int = 0
  private[this] var s: Array[Int] = new Array(100)
  def push(x: Int) { v += 1; i += 1; s(i) = x }
  def pop(): Int = { v += 1; i -= 1; s(i+1) }
  def iterator : Iterator[Int] = new Iterator[Int] {
    var j: Int = i; val v1 = v
    def hasNext = if (v==v1) { j >= 0 } else { ... }
    def next() = if (v==v1) { j -= 1; s(j+1) } else { ... }
  }
}
```

[A] Version number.

# Iterator

- An abstraction of enumeration. It represents a sequence of objects with the current position.
- A collection usually comes with a method for creating an iterator.
- A collection and its iterators share state, and they together form a data abstraction.

# Using iterator abstraction 1: Implement a buffered iterator

- A buffered iterator has a head method.
- This method returns the current value without changing the current position.
- Define a function that transforms an iterator to a buffered iterator.

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def toBuffered[T](x: Iter[T]): BIter[T] = ...
```

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def toBuffered[T](x: Iter[T]): BIter[T] = new BIter[T] {

  def head: T = ...



  def hasNext: Boolean = ...
  def next(): T = ...



}
```

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def toBuffered[T](x: Iter[T]): BIter[T] = new BIter[T] {
  private var hd: Option[T] = None
  def head: T = hd match {
    case None => val v = next(); hd = Some(v); v
    case Some(v) => v
  }
  def hasNext: Boolean = ...
  def next(): T = ...



}
```

One lookahead when forced.

# [Q] Complete the definition of toBuffered.

One lookahead when forced.

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def toBuffered[T](x: Iter[T]): BIter[T] = new BIter[T] {
  private var hd: Option[T] = None
  def head: T = hd match {
    case None => val v = next(); hd = Some(v); v
    case Some(v) => v
  }
  def hasNext: Boolean = ...
  def next(): T = ...



}
```

# [Q] Complete the definition of toBuffered.

One lookahead when forced.

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def toBuffered[T](x: Iter[T]): BIter[T] = new BIter[T] {
  private var hd: Option[T] = None
  def head: T = hd match {
    case None => val v = next(); hd = Some(v); v
    case Some(v) => v
  }
  def hasNext: Boolean = hd.isDefined || x.hasNext
  def next(): T = ...



}
```

# [Q] Complete the definition of toBuffered.

One lookahead when forced.

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def toBuffered[T](x: Iter[T]): BIter[T] = new BIter[T] {
  private var hd: Option[T] = None
  def head: T = hd match {
    case None => val v = next(); hd = Some(v); v
    case Some(v) => v
  }
  def hasNext: Boolean = hd.isDefined || x.hasNext
  def next(): T =  hd match {
    case None => x.next()
    case Some(v) => hd = None; v
  }
}
```

# Using iterator abstraction 2: Partitioning an iterator

- Generate two iterators from a given buffered iterator and a predicate p.
- The first iterator enumerates elements satisfying p.
- The second iterator enumerates elements that do not satisfy p.

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def partition[T](it:BIter[T], p:T=>Boolean):(Iter[T],Iter[T]) =
  ...
```

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def partition[T](it:BIter[T], p:T=>Boolean):(Iter[T],Iter[T]) = {
  class PIter(q: T => Boolean) extends Iter[T] ...




  val l = new PIter(p); val r = new PIter(!p(_))
  ...
  (l, r)
}
```

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def partition[T](it:BIter[T], p:T=>Boolean):(Iter[T],Iter[T]) = {
  class PIter(q: T => Boolean) extends Iter[T] {



    def hasNext = ...
    def next() = ...

  }
  val l = new PIter(p); val r = new PIter(!p(_))
  ...
  (l, r)
}
```

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def partition[T](it:BIter[T], p:T=>Boolean):(Iter[T],Iter[T]) = {
  class PIter(q: T => Boolean) extends Iter[T] {
    var other: PIter = null
    val lookahead = new collection.mutable.Queue[T]



    def hasNext = ...
    def next() = ...

  }
  val l = new PIter(p); val r = new PIter(!p(_))
  l.other = r; r.other = l
  (l, r)
}
```

Give it to other if it is not mine.

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def partition[T](it:BIter[T], p:T=>Boolean):(Iter[T],Iter[T]) = {
  class PIter(q: T => Boolean) extends Iter[T] {
    var other: PIter = null
    val lookahead = new collection.mutable.Queue[T]
    def skip() =
      while (it.hasNext && !q(it.head))
        { other.lookahead += it.next() }
    def hasNext = !lookahead.isEmpty || { skip(); it.hasNext }
    def next() = ...

  }
  val l = new PIter(p); val r = new PIter(!p(_))
  l.other = r; r.other = l
  (l, r)
}
```

Give it to other if it is not mine.

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def partition[T](it:BIter[T], p:T=>Boolean):(Iter[T],Iter[T]) = {
  class PIter(q: T => Boolean) extends Iter[T] {
    var other: PIter = null
    val lookahead = new collection.mutable.Queue[T]
    def skip() =
      while (it.hasNext && !q(it.head))
        { other.lookahead += it.next() }
    def hasNext = !lookahead.isEmpty || { skip(); it.hasNext }
    def next() = ...

  }
  val l = new PIter(p); val r = new PIter(!p(_))
  l.other = r; r.other = l
  (l, r)
}
```

[Q] Define `next()`

Give it to other if it is not mine.

```
trait Iter[T] {
  def next(): T
  def hasNext: Boolean
}
trait BIter[T] extends Iter[T] {
  def head: T
}

def partition[T](it:BIter[T], p:T=>Boolean):(Iter[T],Iter[T]) = {
  class PIter(q: T => Boolean) extends Iter[T] {
    var other: PIter = null
    val lookahead = new collection.mutable.Queue[T]
    def skip() =
      while (it.hasNext && !q(it.head))
        { other.lookahead += it.next() }
    def hasNext = !lookahead.isEmpty || { skip(); it.hasNext }
    def next() = if (!lookahead.isEmpty) lookahead.dequeue()
                 else { skip(); it.next() }
  }
  val l = new PIter(p); val r = new PIter(!p(_))
  l.other = r; r.other = l
  (l, r)
}
```

[Q] Define next()

# Using iterator abstraction

- We can build interesting iterators from a given one, without any further assumption.
- Many examples are included in the implementation of Scala's `Iterator[T]`.

# Summary

- Iterator -- abstraction of enumeration.
- A collection class and its iterators.
    - Iterators generated by a collection.
    - They share the internal state of the coll.
- Read Chapter 24.1-4 and 24.16.