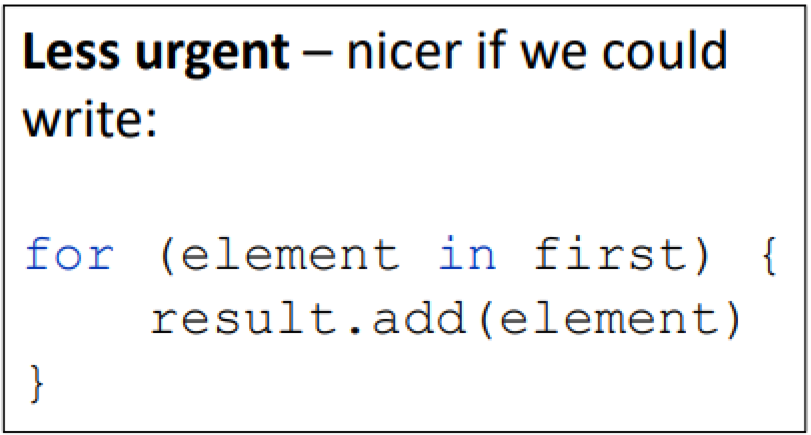
Iterators

```
fun <T> combine(
    first: ImperialMutableList<T>,
    second: ImperialMutableList<T>,
): ImperialMutableList<T> {
    val result = SinglyLinkedList<T>()
    for (index in 0..<first.size) {
        result.add(first[index])
    }
    for (index in 0..<second.size) {
        result.add(second[index])
    }
    return result
}
```

These loops have high computational complexity. Each lookup may take linear time (if linked lists are used). We do a linear number of linear time lookups: one for each array element.

Overall: quadratic time complexity = Bad



Less urgent – nicer if we could write:

```
for (element in first) {
    result.add(element)
}
```

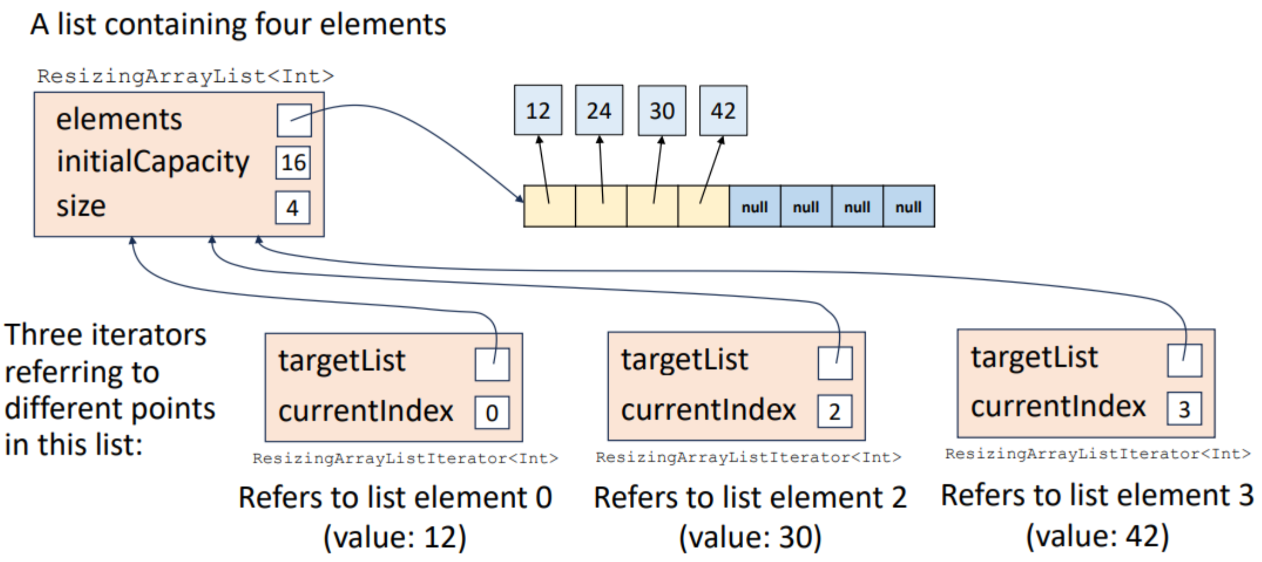
An iterator is an object that can be used to iterate through all elements in a collection.

An iterator provides the following service:

- `hasNext()` : Indicates whether it has reached the end of the collection, or whether there are more elements to be iterated over

- `next()`: Provides the current element to which it is referring, and moves on to the next element in the collection, if any
- An exception is thrown by `next()` if `hasNext()` does not hold:
  - `hasNext()` is a **precondition** of `next()`

Iterating through a resizing array list



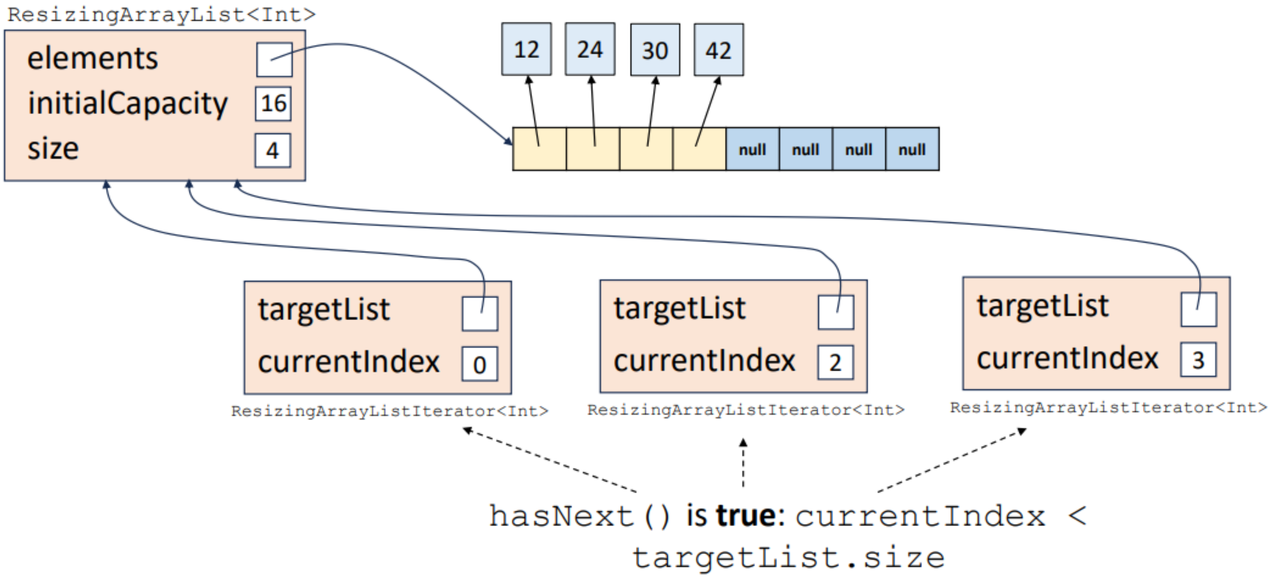
A list containing four elements

Three iterators referring to different points in this list:

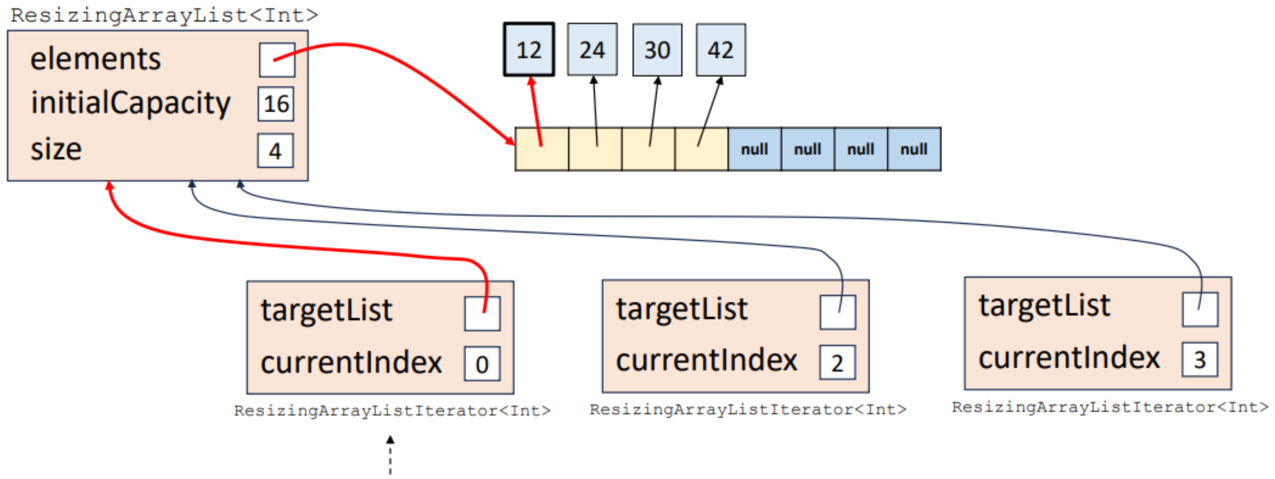
Refers to list element 0 (value: 12)

Refers to list element 2 (value: 30)

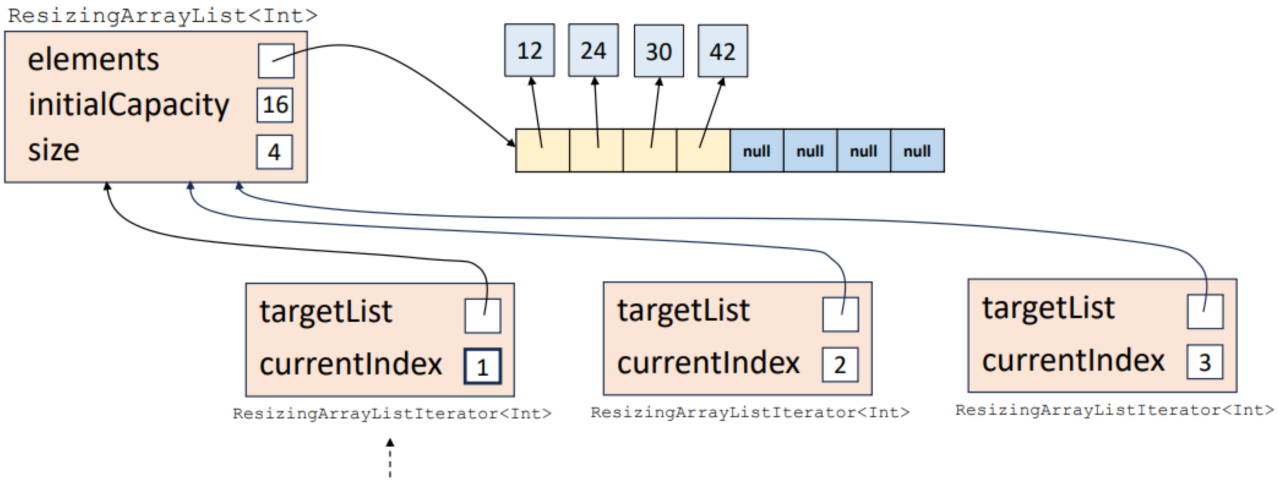
Refers to list element 3 (value: 42)



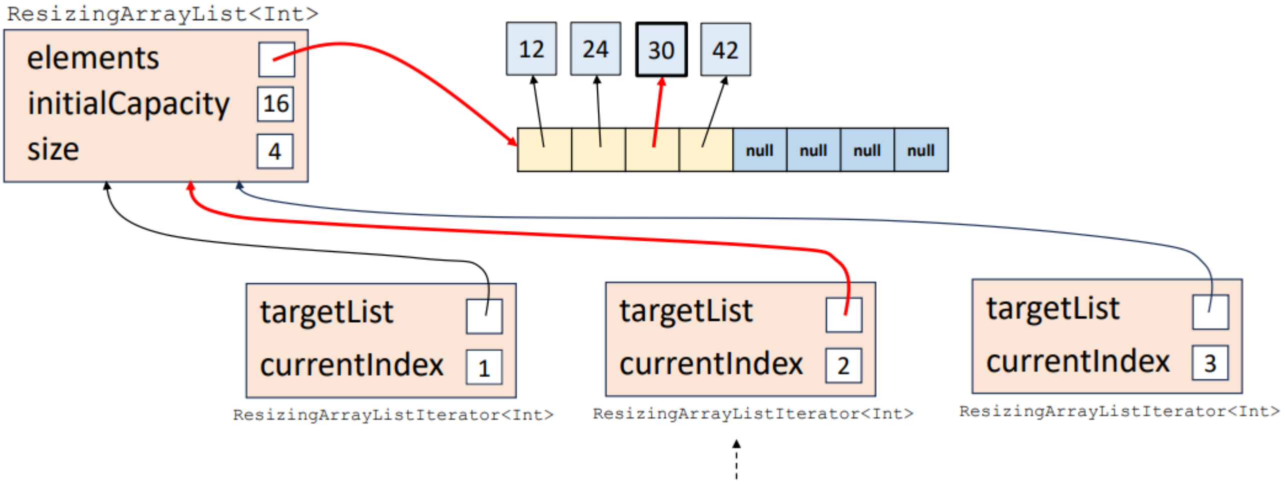
`hasNext()` is **true**: `currentIndex < targetList.size`



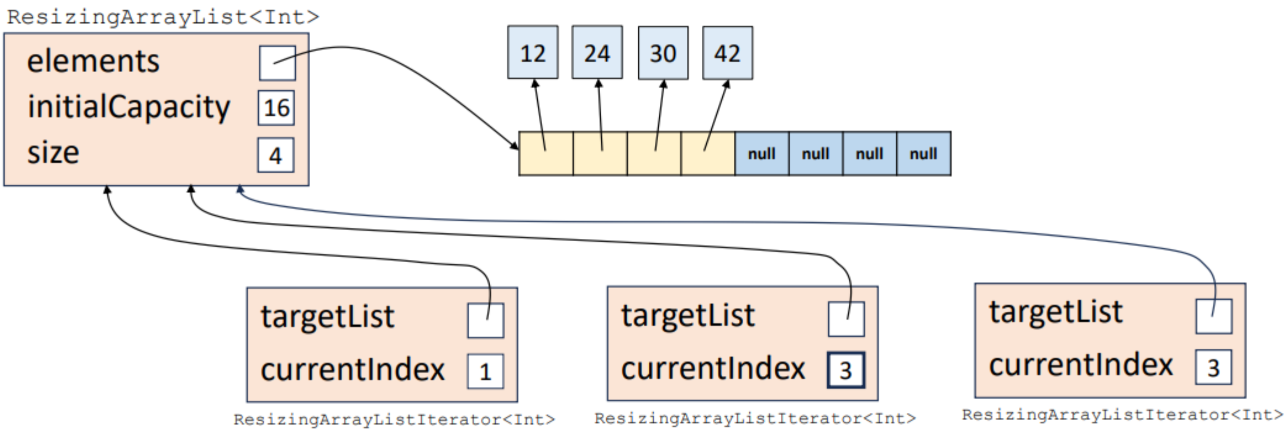
`next()` returns 12



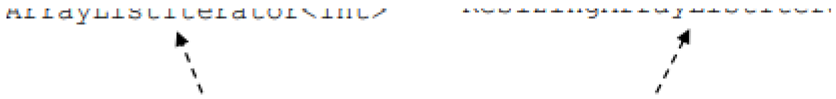
`next()` causes iterator to move
on to the next element



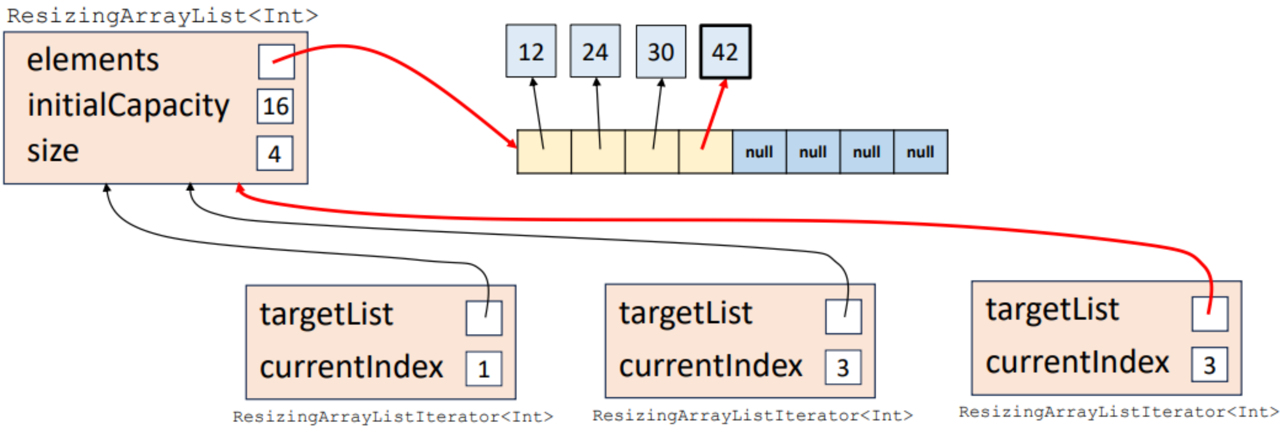
`next()` returns 30



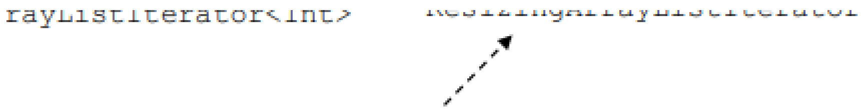
`next()` causes iterator to move
on to the next element



These iterators both refer to the
last element in the list



`next()` returns 42



`next()` causes iterator to move
beyond the last element

`hasNext()` is **false**: `currentIndex == targetList.size`

`next()` should not be called: it will throw an **exception**

Recap of iterator for `ResizingArrayList`

The iterator needs to track:

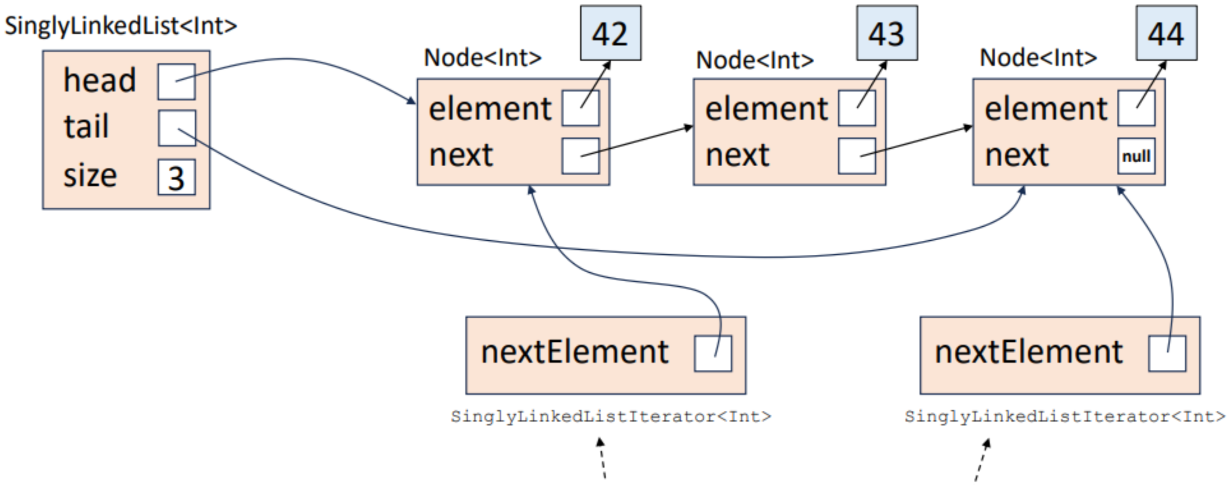
- The list being traversed
- The index associated with the iterator's next element

The methods work as follows

- `hasNext()`: checks whether iterator's index has reached list size
- `next()`: retrieves element at iterator's index; increments the index

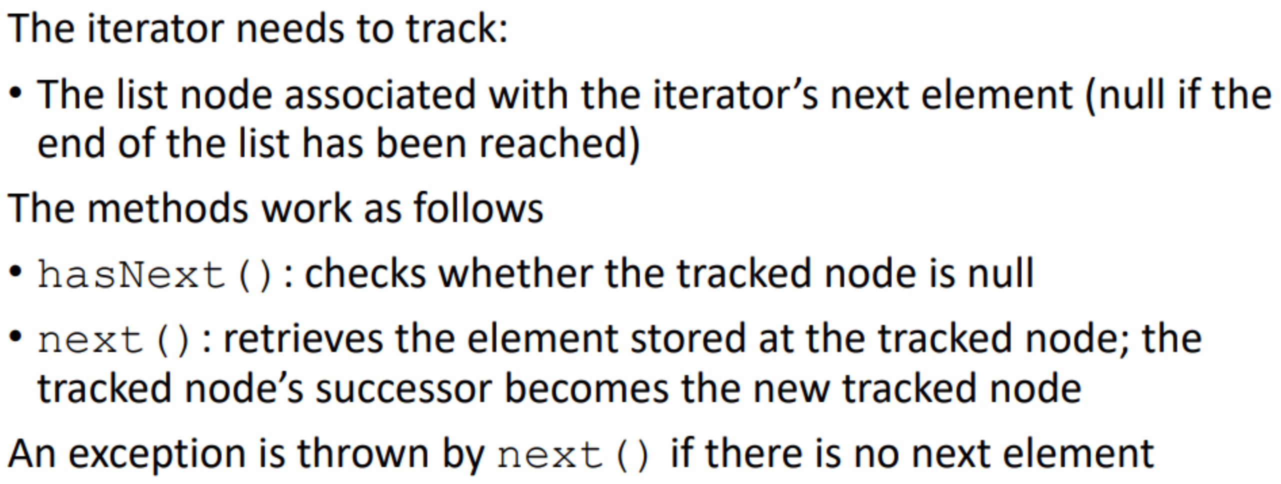
An exception is thrown by `next()` if there is no next element

Iterating through a singly-linked list



`hasNext()` is **true**: `nextElement != null`

Recap of iterator for `SinglyLinkedList`



The iterator needs to track:

- The list node associated with the iterator's next element (null if the end of the list has been reached)

The methods work as follows

- `hasNext()`: checks whether the tracked node is null
- `next()`: retrieves the element stored at the tracked node; the tracked node's successor becomes the new tracked node

An exception is thrown by `next()` if there is no next element

The `Iterator<T>` interface

```
interface Iterator<T> {
    fun hasNext(): Boolean
    fun next(): T
}
```

An iterator for `ResizingArrayList`

```
class ResizingArrayListIterator<T>(
    private val targetList: ResizingArrayList<T>,
) : Iterator<T> {
    private var currentIndex: Int = 0
    // Determines the list element
    // the iterator will return next
    override fun hasNext(): Boolean = currentIndex <
targetList.size
    override fun next(): T = if (!hasNext()) {
        throw NoSuchElementException()
    } else {
        targetList[currentIndex++]
        // The index is post-incremented: the increment happens
        // after an element from targetList has been retrieved
        }
}
```

Implementing the `iterator()` method

```
class ResizingArrayList<T>(
    private val initialCapacity: Int,
) : ImperialMutableList<T>() {

    override fun iterator(): Iterator<T> =
        ResizingArrayListIterator(this)

}
```

Use a private nested class instead

```
class ResizingArrayList<T>(
    private val initialCapacity: Int,
) : ImperialMutableList<T>() {
    private class ResizingArrayListIterator<T>(
        private val targetList: ResizingArrayList<T>,
    ) : Iterator<T> {
        private var currentIndex: Int = 0
        override fun hasNext(): Boolean = currentIndex <
targetList.size
        override fun next(): T = ...
    }
    ...
    override fun iterator(): Iterator<T> =
ResizingArrayListIterator(this)
}
```

Observation

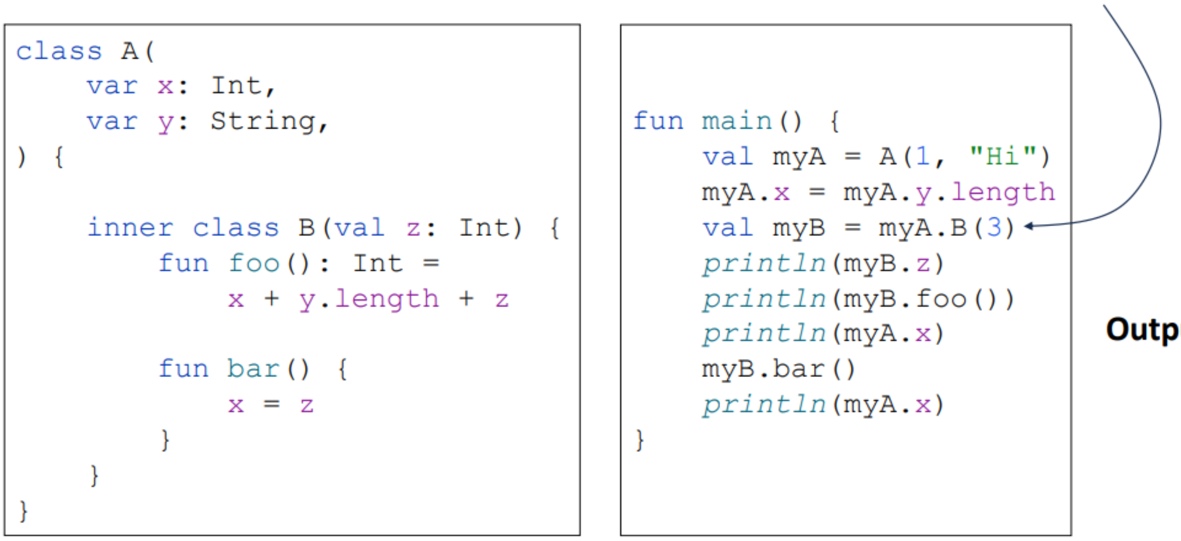
- Our `ResizingArrayListIterator` requires access to a `ResizingArrayList`
- Further, it should always have access to exactly the `ResizingArrayList` on which `iterator()` was called
- That's why we pass this to `ResizingArrayListIterator`: no other `ResizingArrayList` would be appropriate
- This use case is better served by an inner class than a nested class

Inner classes

If A is a class, then an inner class of A is a regular class B defined inside A, with two key differences:

- An instance of inner class B can only be created via an instance of A
- The resulting instance of B has access to the properties and methods of the instance of A that created it
  Use inner to declare an inner class

An Example



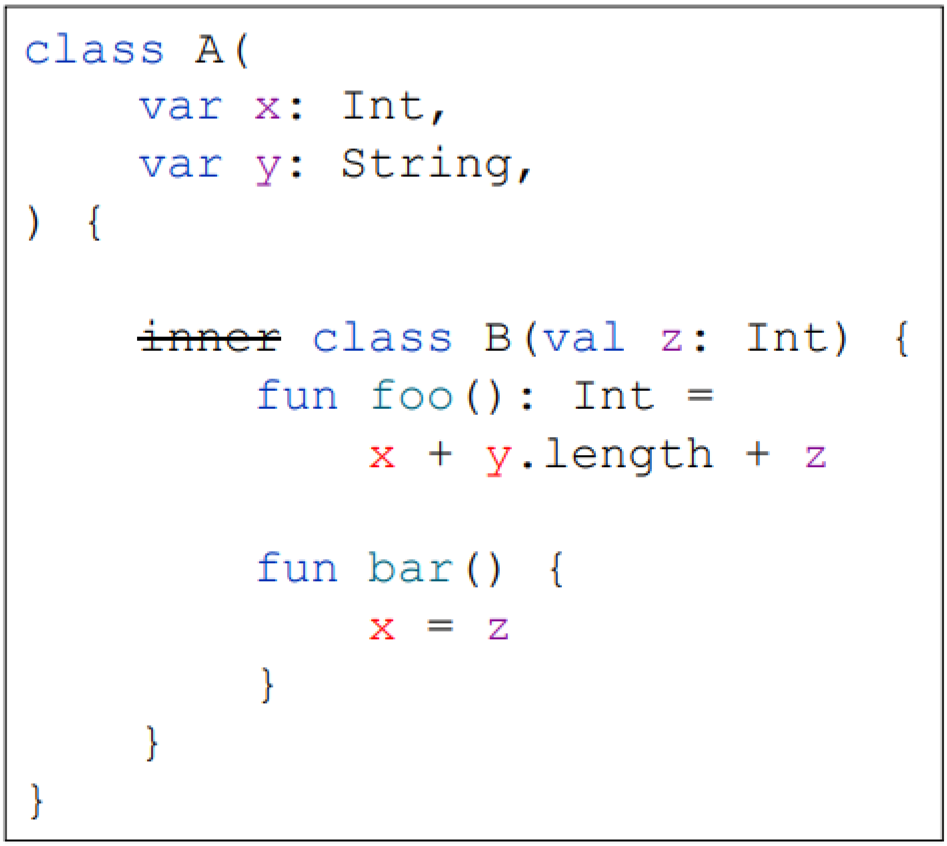
```
class A(
    var x: Int,
    var y: String,
) {

    inner class B(val z: Int) {
        fun foo(): Int =
            x + y.length + z

        fun bar() {
            x = z
        }
    }
}
```

```
fun main() {
    val myA = A(1, "Hi")
    myA.x = myA.y.length
    val myB = myA.B(3)
    println(myB.z)
    println(myB.foo())
    println(myA.x)
    myB.bar()
    println(myA.x)
}
```

Output:

```
3
7
2
3
```

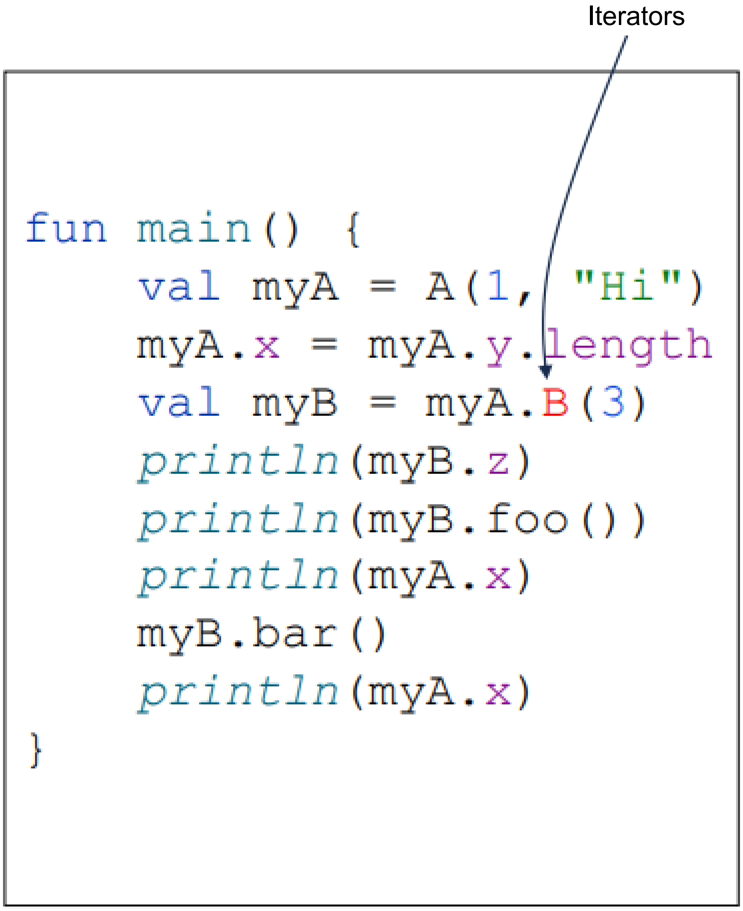
Inner class vs. nested class



```
class A(
    var x: Int,
    var y: String,
) {

    ~~inner~~ class B(val z: Int) {
        fun foo(): Int =
            x + y.length + z

        fun bar() {
            x = z
        }
    }
}
```

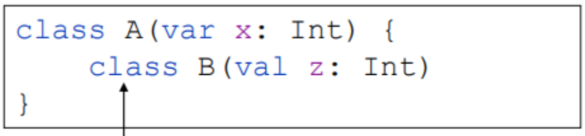
This makes B a nested class. It does not compile (unresolved references to x and y).

A nested class does not have an associated instance of the enclosing class. Here, a B instance can exist even though no A instances exist. Referring to x and y from code in B is therefore meaningless.

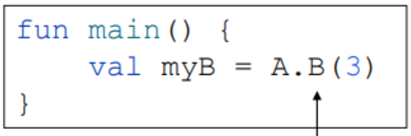


```
fun main() {
    val myA = A(1, "Hi")
    myA.x = myA.y.length
    val myB = myA.B(3)
    println(myB.z)
    println(myB.foo())
    println(myA.x)
    myB.bar()
    println(myA.x)
}
```

This attempt to construct a B instance is also illegal. Looks to compiler like we are trying to call a method named B on `myA`.



```
class A(var x: Int) {
    class B(val z: Int)
}
```

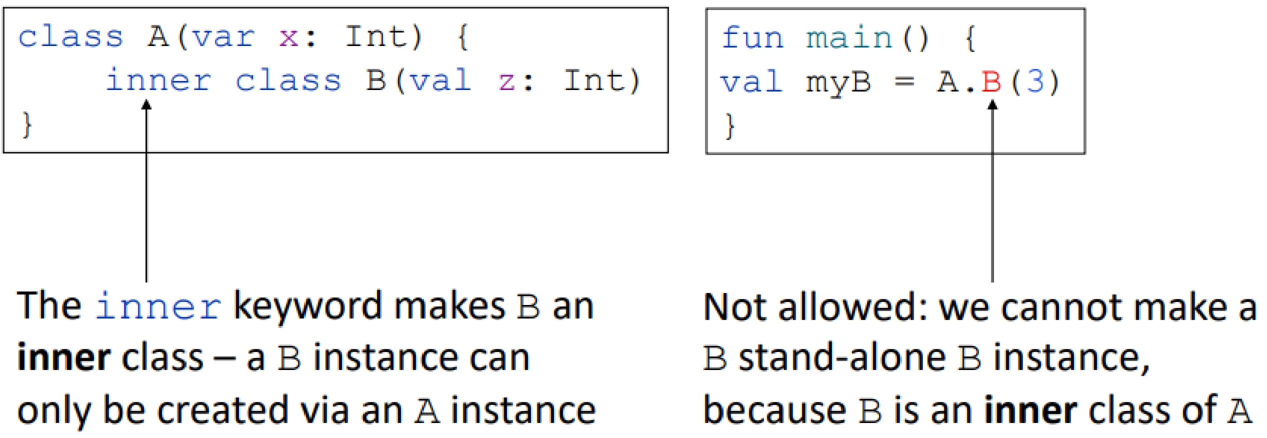


```
fun main() {
    val myB = A.B(3)
}
```

`B` is a **nested** class, not an inner class: no use of `inner` keyword

We do not need an `A` instance to construct a `B` instance – here, `A` refers to the **class** `A`, not any particular instance of `A`

The full name of `B` is `A.B`, so really we are creating an instance of the `A.B` class



```
class A(var x: Int) {
    inner class B(val z: Int)
}
```

```
fun main() {
val myB = A.B(3)
}
```

The `inner` keyword makes `B` an **inner** class – a `B` instance can only be created via an `A` instance

Not allowed: we cannot make a `B` stand-alone `B` instance, because `B` is an **inner** class of `A`

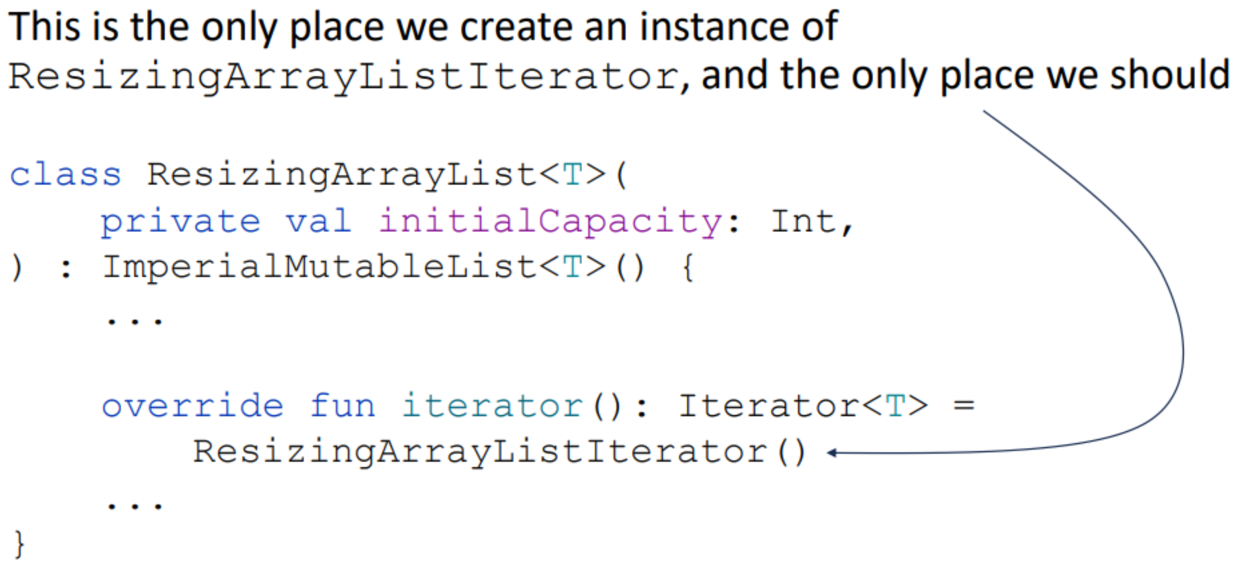
`ResizingArrayListIterator` as an inner class

```
class ResizingArrayList<T>(
    private val initialCapacity: Int,
) : ImperialMutableList<T>() {
    private inner class ResizingArrayListIterator : Iterator<T> {
        private var currentIndex: Int = 0
        override fun hasNext(): Boolean = currentIndex < size
        override fun next(): T = if (!hasNext()) {
            throw NoSuchElementException()
        } else {
            this@ResizingArrayList[currentIndex++]
            // Refers to enclosing class instance
        }
    }
}
```

```
class ResizingArrayList<T>(
    private val initialCapacity: Int,
) : ImperialMutableList<T>() {
    ...
    override fun iterator(): Iterator<T> =
        ResizingArrayListIterator()
    ...
}
```

Before: we had to pass `this` to the constructor: No longer required.

Observations



This is the only place we create an instance of `ResizingArrayListIterator`, and the only place we should

```
class ResizingArrayList<T>(
    private val initialCapacity: Int,
) : ImperialMutableList<T>() {
    ...

    override fun iterator(): Iterator<T> =
        ResizingArrayListIterator()
    ...
}
```

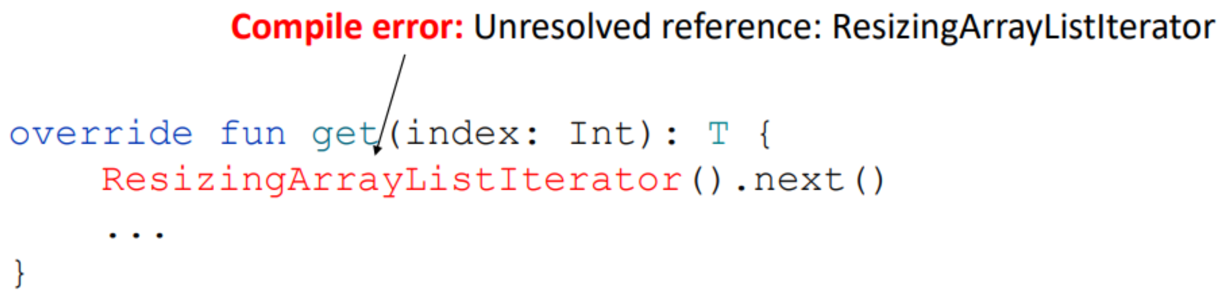
It would be wrong to create a `ResizingArrayListIterator` in any other method of `ResizingArrayList`, but it is possible. Better if it were impossible to make this mistake.

Implementing `iterator()` via an anonymous object

```
class ResizingArrayList<T>(
    private val initialCapacity: Int,
) : ImperialMutableList<T>() {
    ...
    override fun iterator(): Iterator<T> = object : Iterator<T> {
        private var currentIndex = 0
        override fun hasNext(): Boolean = currentIndex < size
        override fun next(): T = elements[currentIndex++]!!
    }
    ...
}
```

Instead of declaring an inner class and then returning an instance of it, this directly returns an object that meets the `Iterator` interface requirements. The object that gets created is an instance of a nameless inner class, so it has access to the `ResizingArrayList` that created it.

We cannot mistakenly create another instance of an anonymous object.



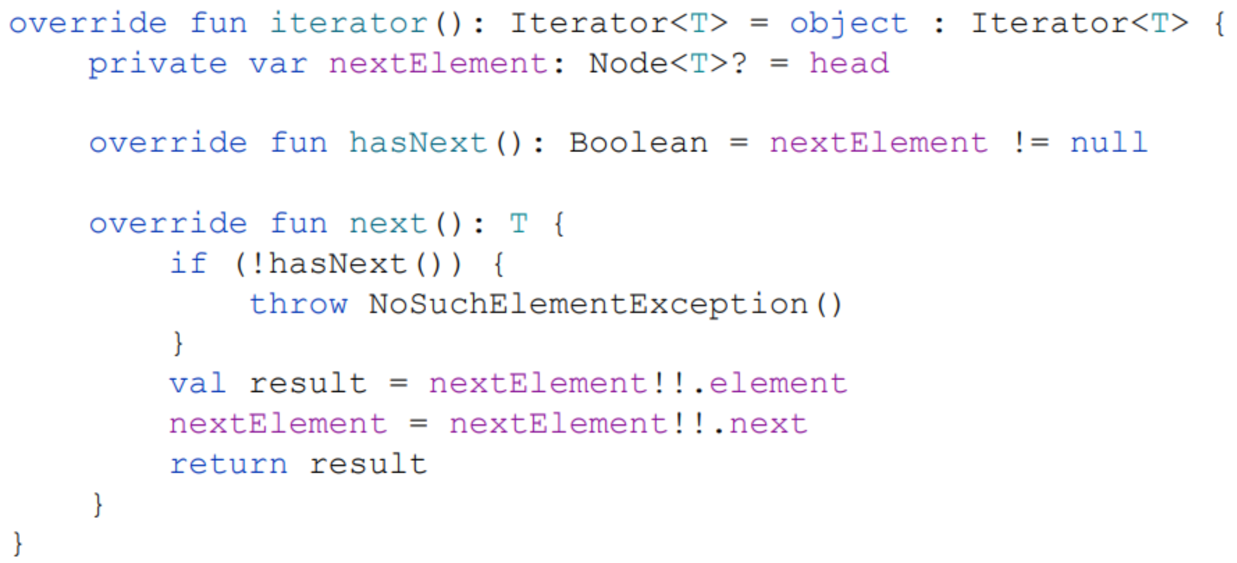
Compile error: Unresolved reference: ResizingArrayListIterator

```
override fun get(index: Int): T {
    ResizingArrayListIterator().next()
    ...
}
```

Impossible to make this mistake: we no longer have an named inner class declaration.

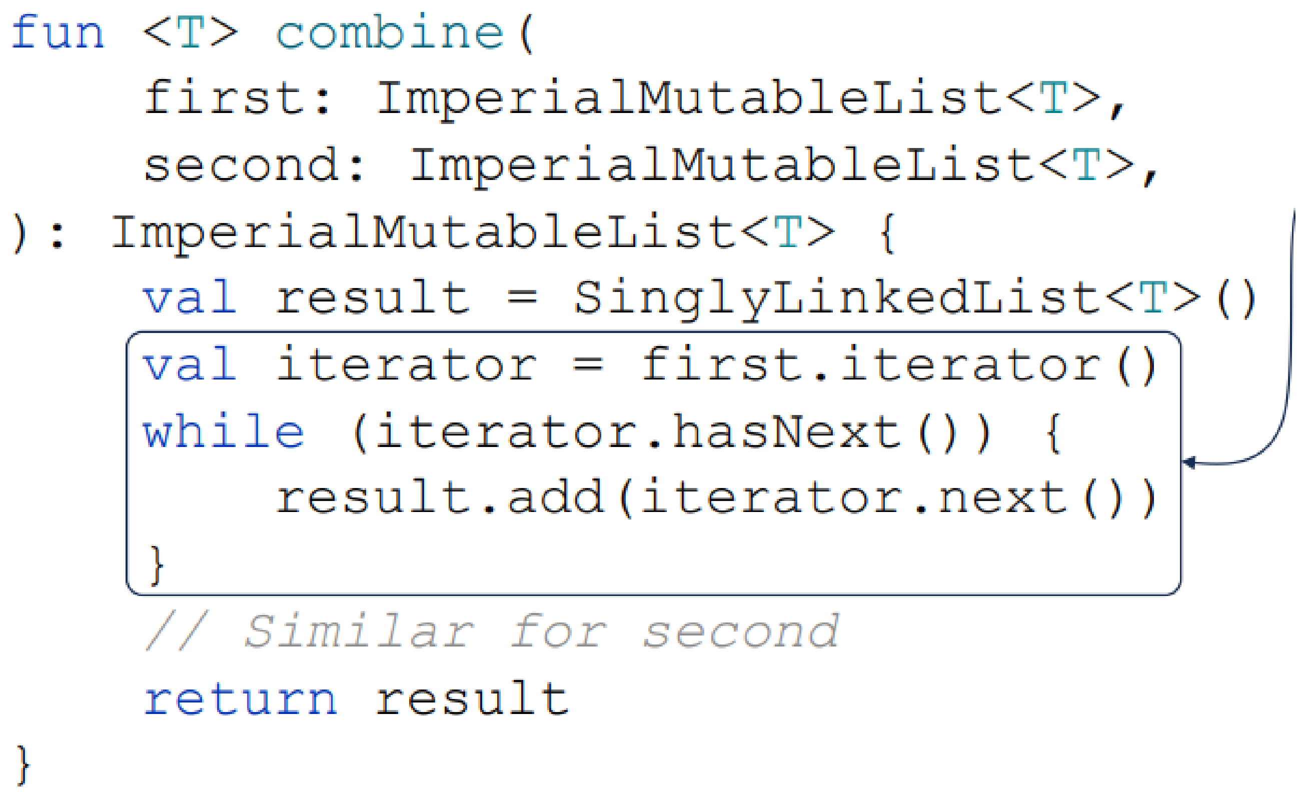
The iterator class defined in `iterator()` has no name – we cannot accidentally refer to it elsewhere.

Iterator for `SinglyLinkedList`



```
override fun iterator(): Iterator<T> = object : Iterator<T> {
    private var nextElement: Node<T>? = head

    override fun hasNext(): Boolean = nextElement != null

    override fun next(): T {
        if (!hasNext()) {
            throw NoSuchElementException()
        }
        val result = nextElement!!.element
        nextElement = nextElement!!.next
        return result
    }
}
```

Iterators avoid quadratic complexity in `combine`



```
fun <T> combine(
    first: ImperialMutableList<T>,
    second: ImperialMutableList<T>,
): ImperialMutableList<T> {
    val result = SinglyLinkedList<T>()
    val iterator = first.iterator()
    while (iterator.hasNext()) {
        result.add(iterator.next())
    }
    // Similar for second
    return result
}
```

The iterator keeps track of where we are in the list (no need to traverse from start to get each element).

If `next()` and `add()` have constant time complexity, this loop has linear time complexity (in the size of `first`). This syntax is painful.

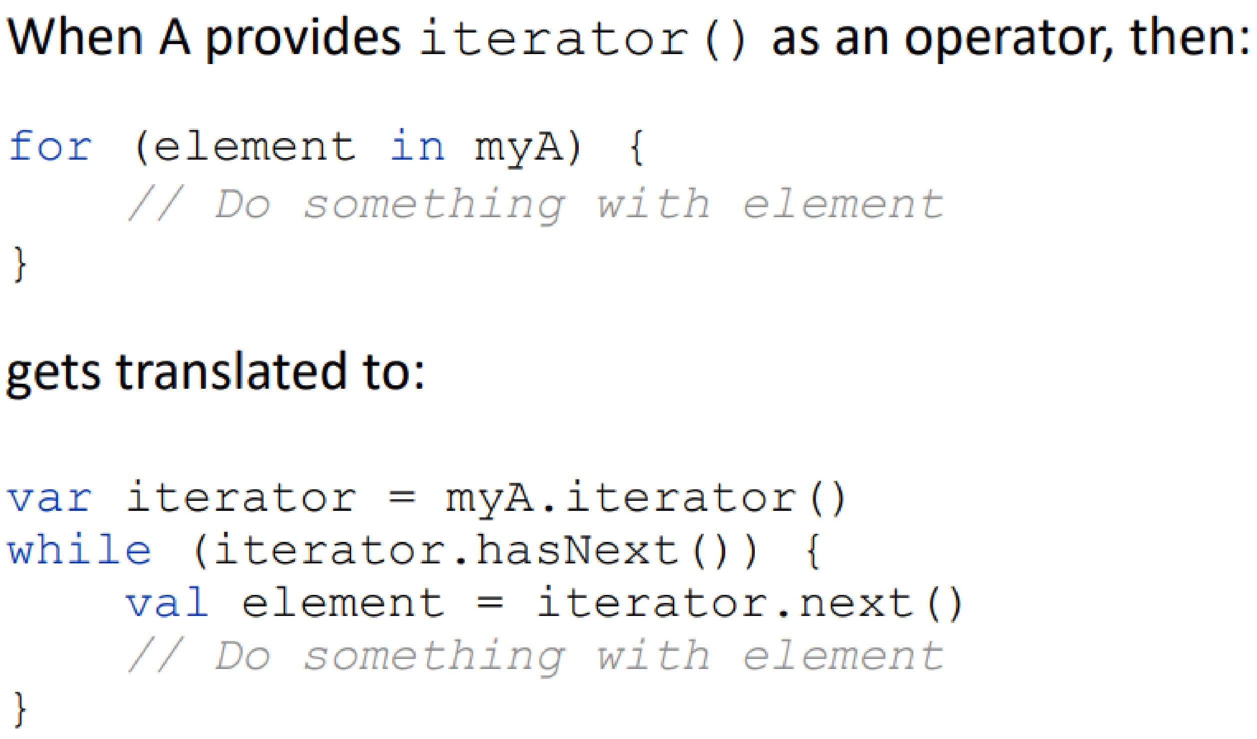
Better Syntax

Instead, do this:

- Suppose a class or interface A has a method that:
    - has name `iterator()`
    - Is declared as operator
    - has return type `Iterator<T>`

Then the syntax `for(element in myA)` can be used to iterate over an instance `myA` of A

Making iterator() an operator function



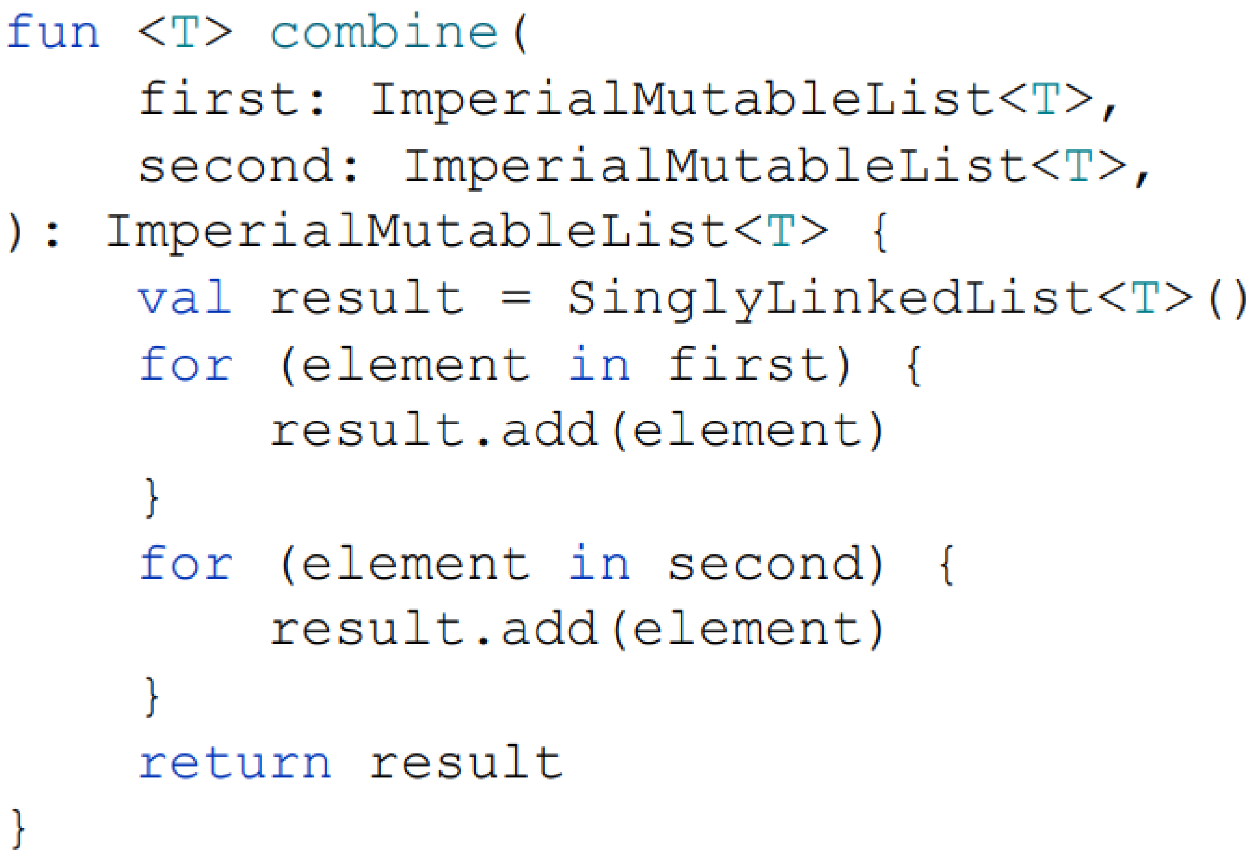
When A provides `iterator()` as an operator, then:

```
for (element in myA) {
    // Do something with element
}
```

gets translated to:

```
var iterator = myA.iterator()
while (iterator.hasNext()) {
    val element = iterator.next()
    // Do something with element
}
```

No more Problems...



```
fun <T> combine(
    first: ImperialMutableList<T>,
    second: ImperialMutableList<T>,
): ImperialMutableList<T> {
    val result = SinglyLinkedList<T>()
    for (element in first) {
        result.add(element)
    }
    for (element in second) {
        result.add(element)
    }
    return result
}
```

Overloading iterator operator gives neat for loop syntax:

- less error prone

Iterators avoid repeated list traversals:

- brings complexity down from **quadratic** to **linear**

...Almost

`add` may have linear time complexity. But at least we avoid repeated inefficient `get` calls.