Iterators and inner classes

Alastair F. Donaldson

Aims of this lecture

- Introduce **iterators** for collection classes
- Provide iterators for the `ResizingArrayList` and `SinglyLinkedList` classes we made earlier
- Show how **inner classes** can be used to represent iterators concisely
- Introduce **anonymous objects**

Remember our `combine` method on `ImperialMutableLists`

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

Nicer if we could write
`first[index]`

Operator overloading enables this!

Marking `get` and `set` as operators

```
interface ImperialMutableList<T> {
    val size: Int
    operator fun get(index: Int): T
    ...
    operator fun set(index: Int, element: T): T
}
```

No changes required to implementing classes when we mark interface method as `operator`

```
class ResizingArrayList<T>(...) :
    ImperialMutableList<T>() {

    ...

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

    ...

    override fun set(index: Int, element: T): T {
        ...
    }
```

No need to reiterate that `get` and `set` overload operators – the interface documents this

Making `combine` a little nicer

```
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
}
```

Remaining problems

```
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

Remaining problems

```
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
}
```

Less urgent – nicer if we could write:

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

Introducing … **iterators**

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

An iterator provides the following **service**:

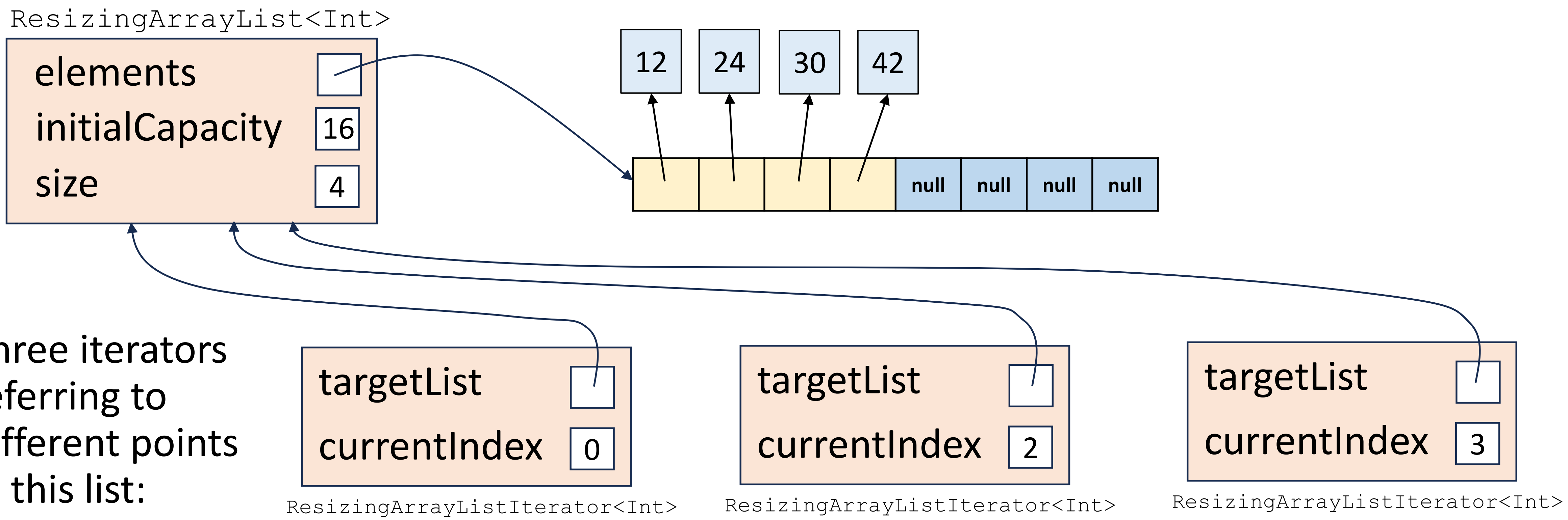
- `hasNext()`: Indicates whether it has reached the end of the collection, or whether there are more elements to be iterated over
- `next()`: Provides the current element to which it is referring, and moves on to the next element in the collection, if any

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



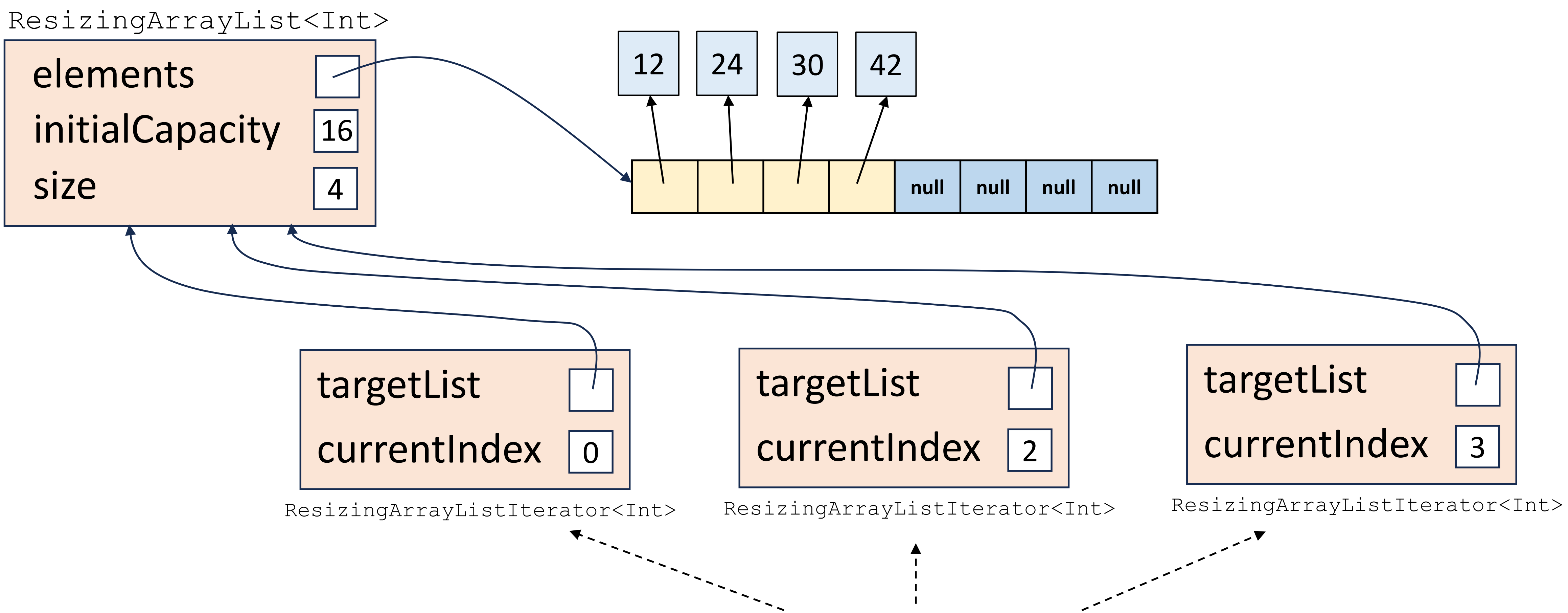
Three iterators referring to different points in this list:

Refers to list element 0 (value: 12)

Refers to list element 2 (value: 30)

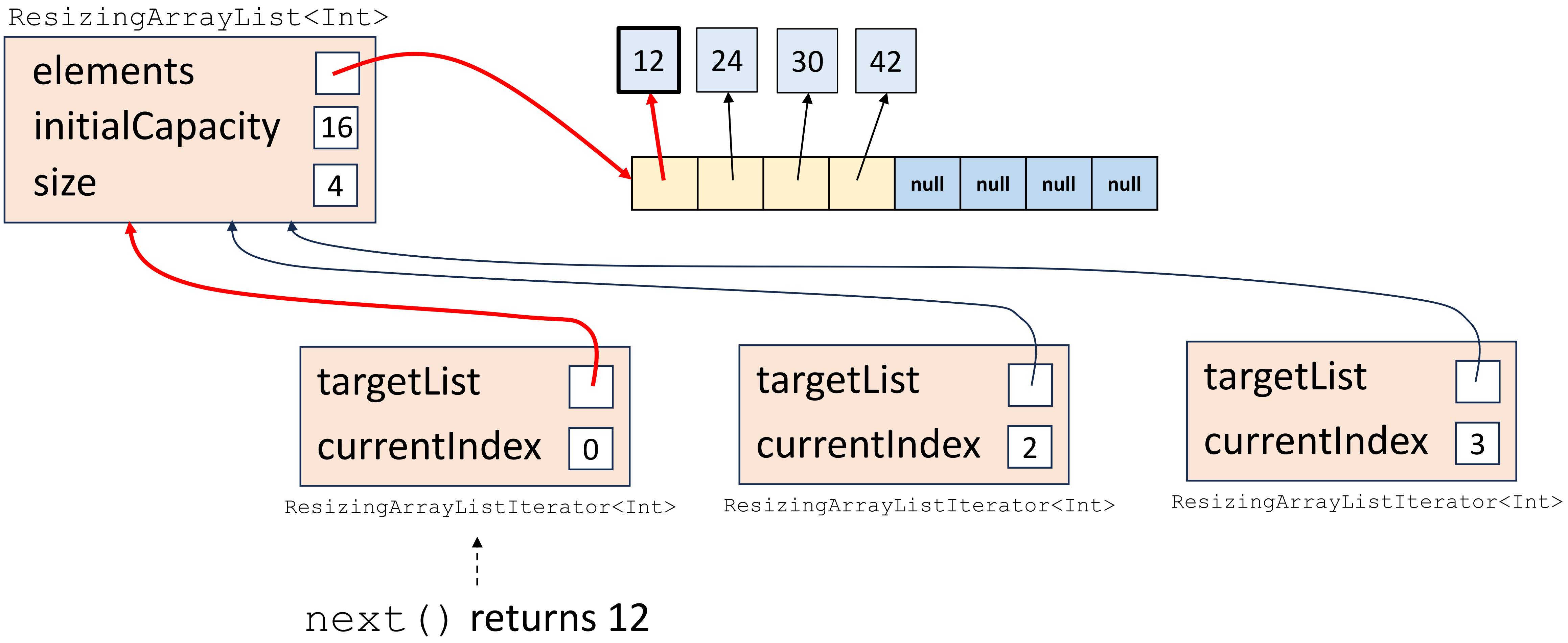
Refers to list element 3 (value: 42)

Iterating through a resizing array list

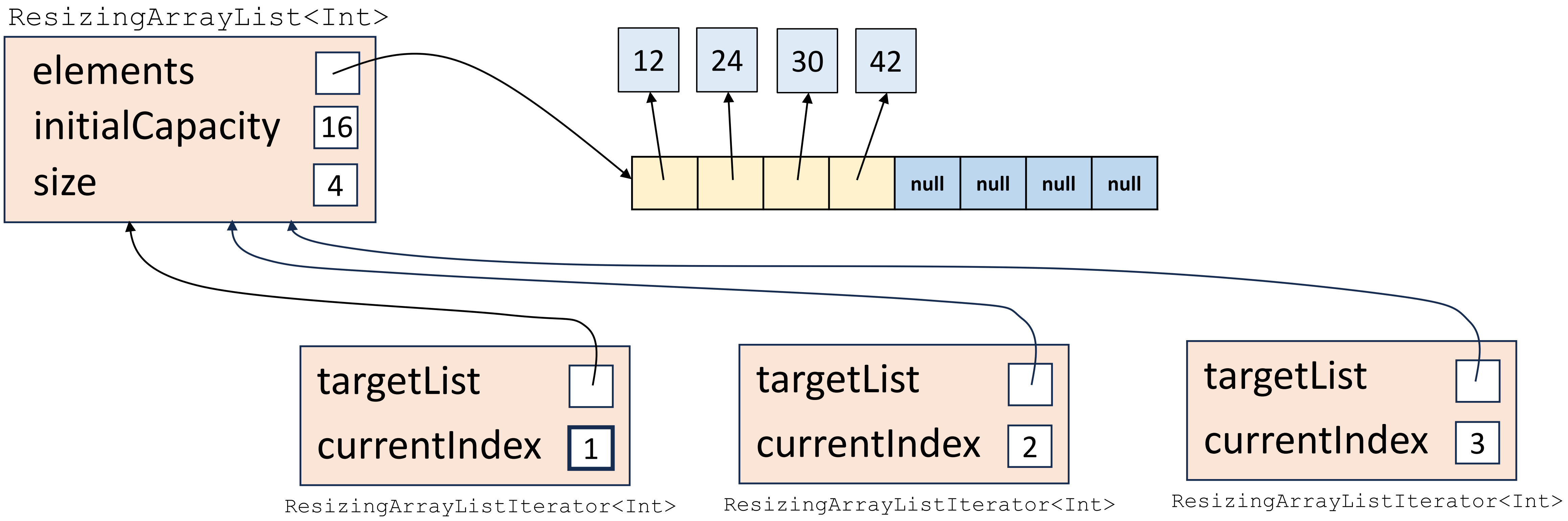


`hasNext()` is **true**: `currentIndex < targetList.size`

Iterating through a resizing array list

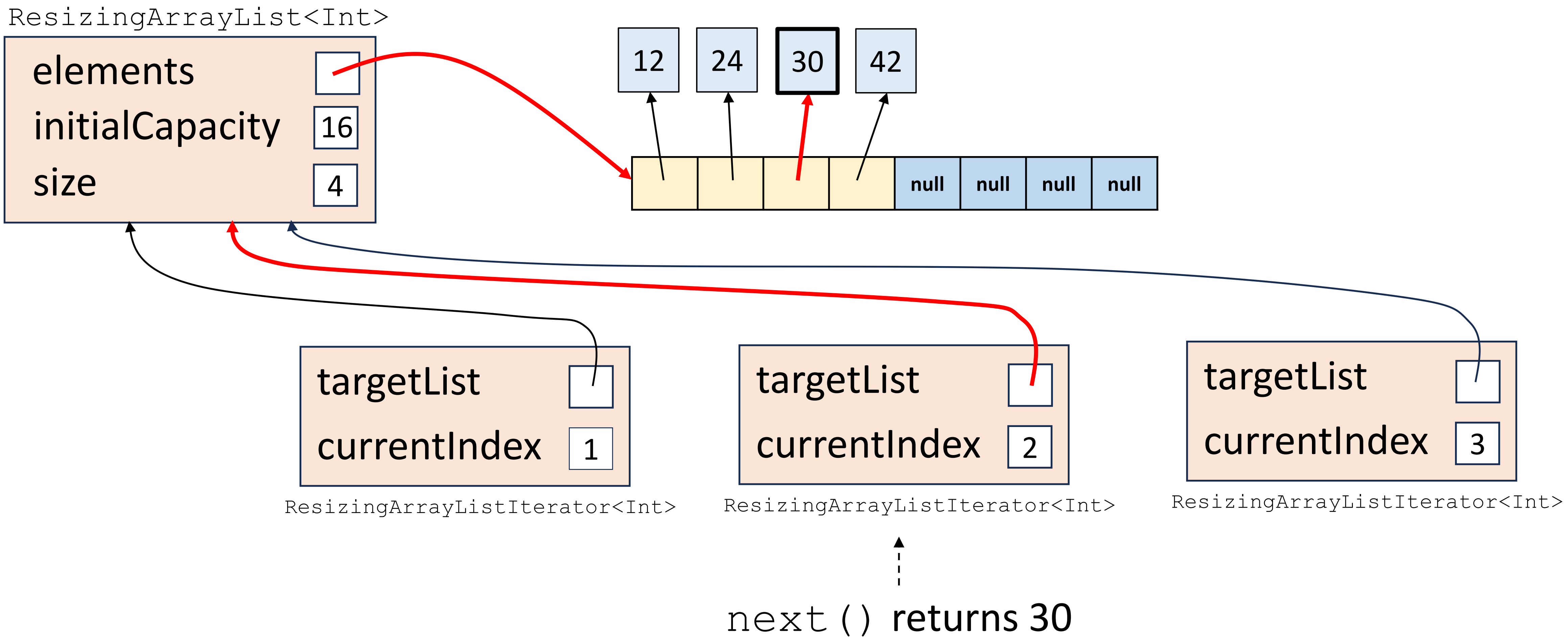


Iterating through a resizing array list

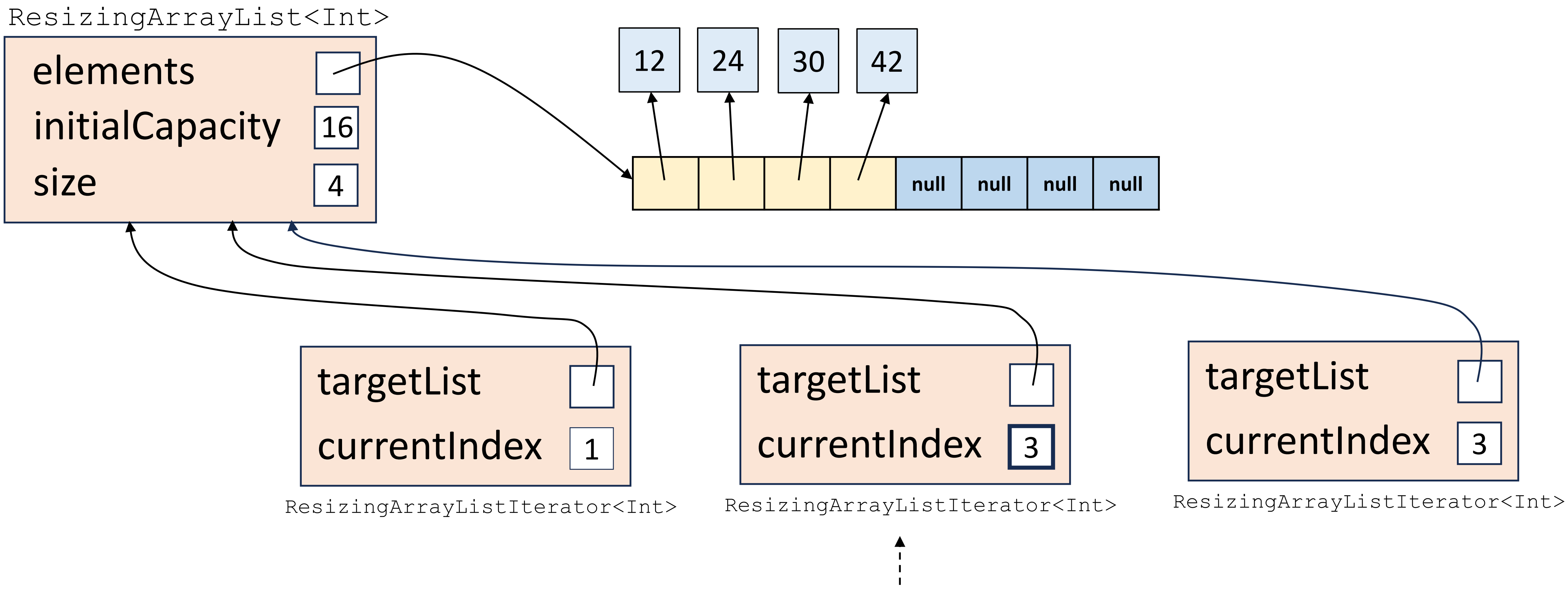


`next()` causes iterator to move on to the next element

Iterating through a resizing array list

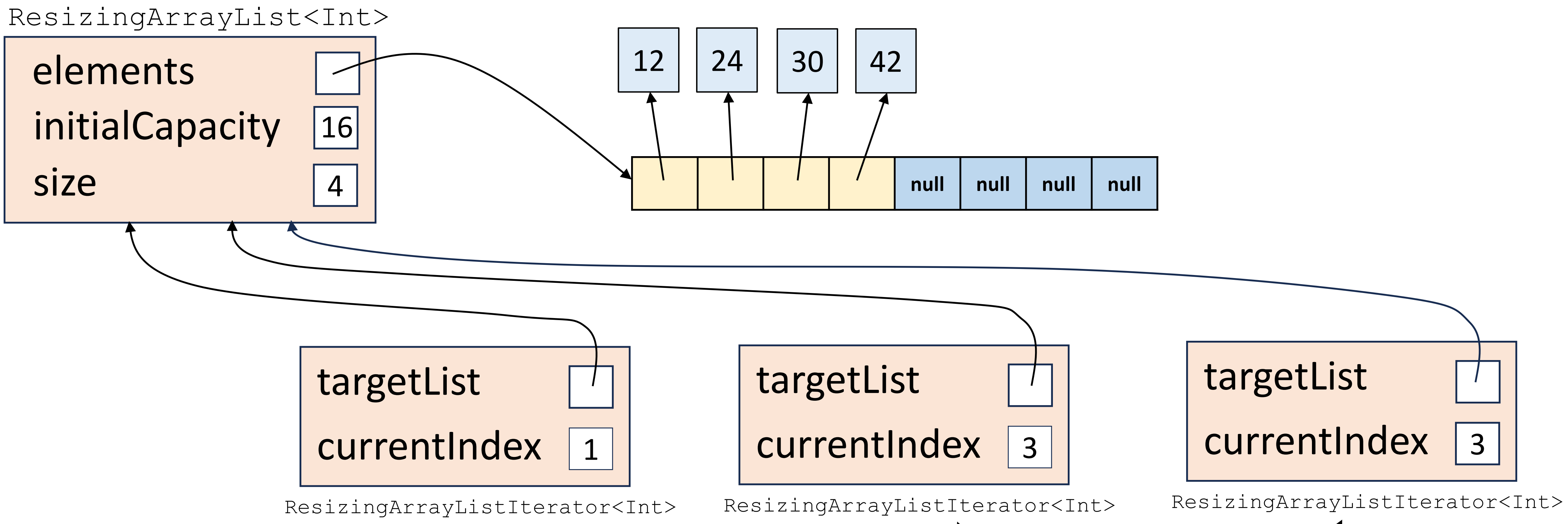


Iterating through a resizing array list



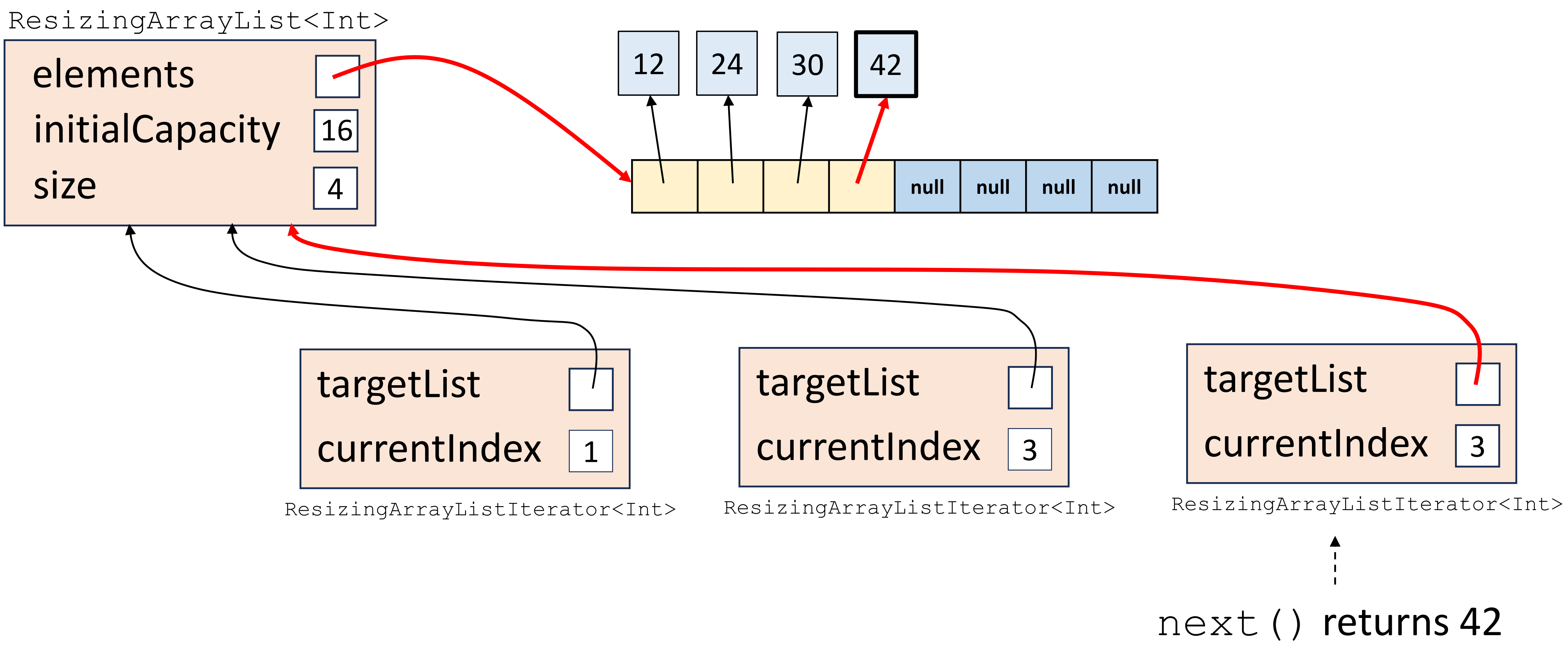
`next()` causes iterator to move on to the next element

Iterating through a resizing array list

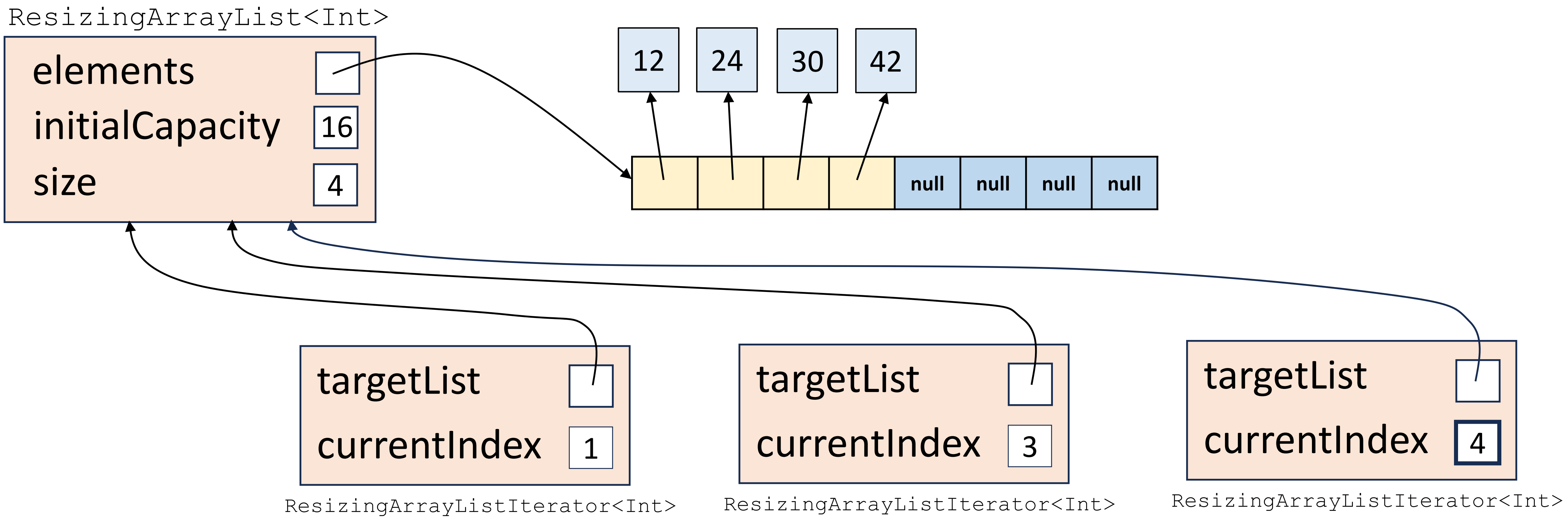


These iterators both refer to the last element in the list

Iterating through a resizing array list

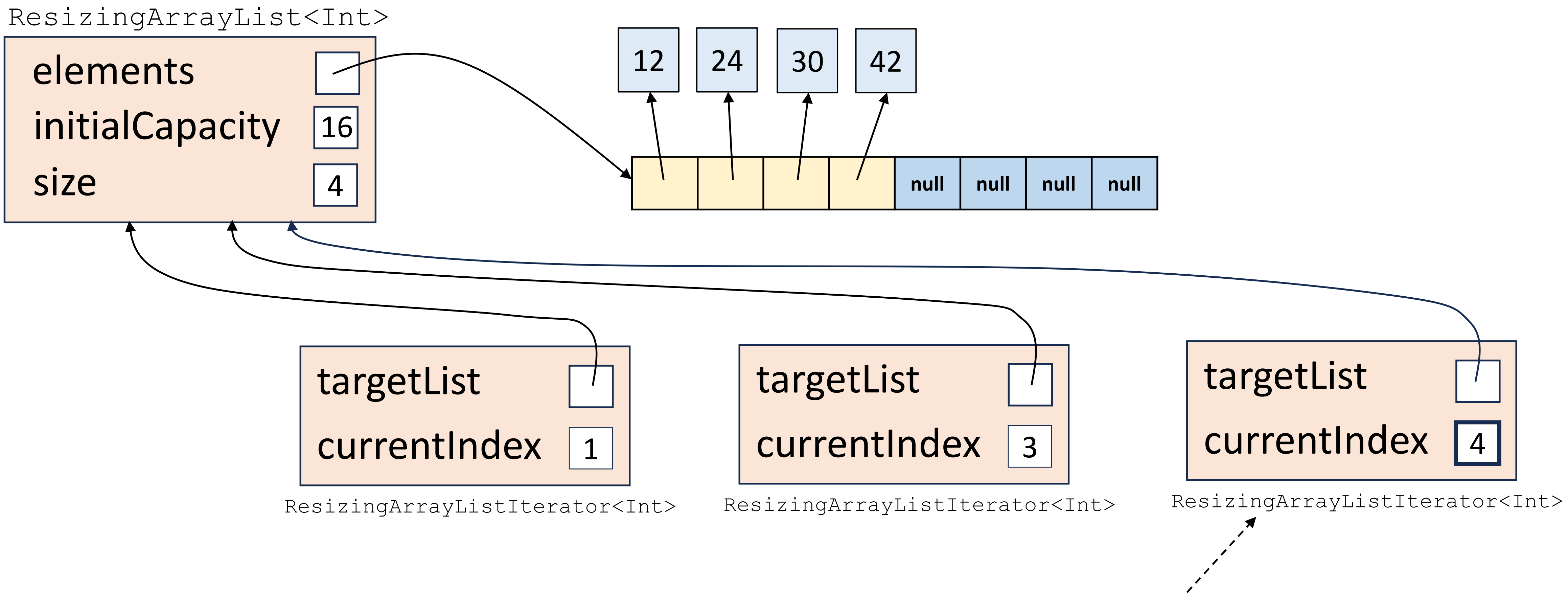


Iterating through a resizing array list



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

Iterating through a resizing array list



`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

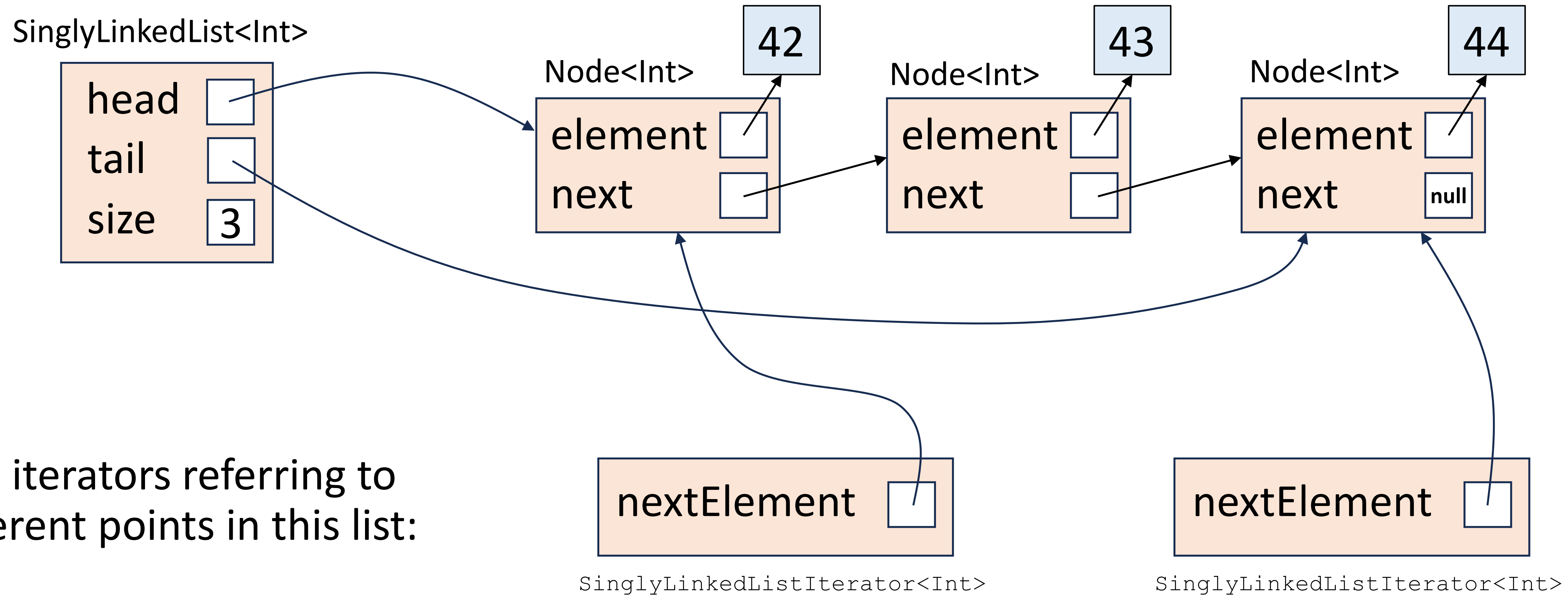
The methods work as follows

- `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

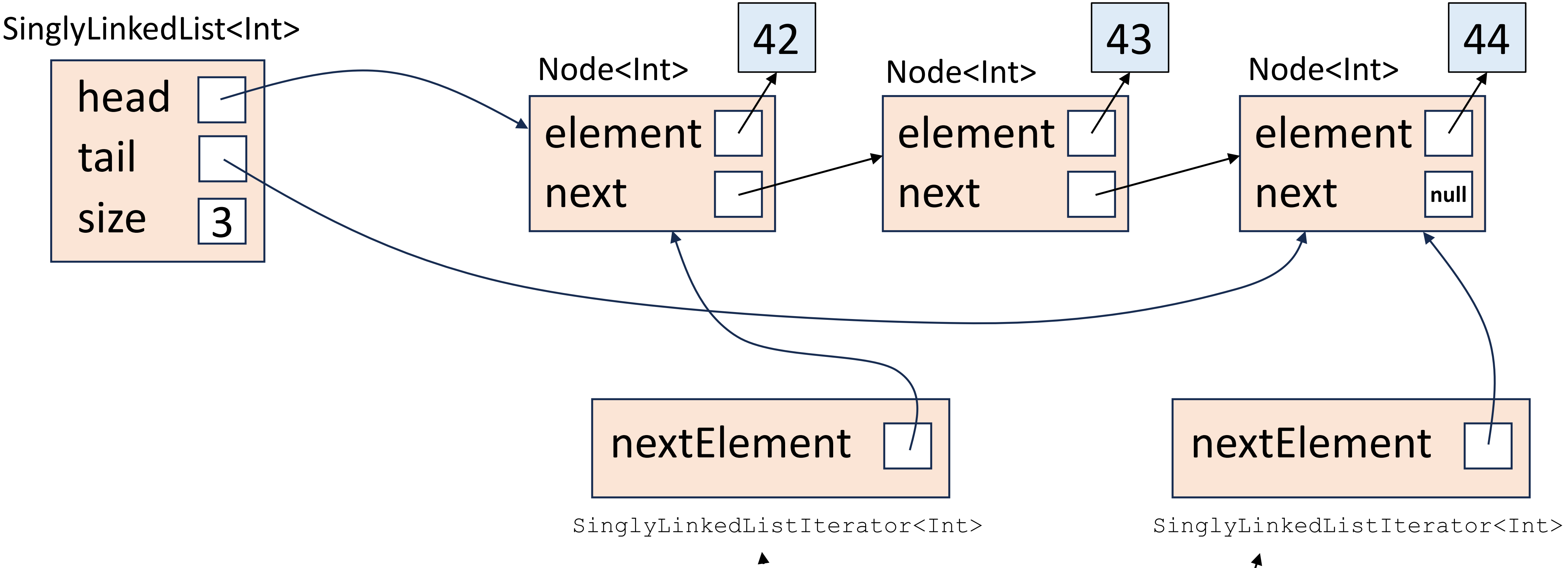
Iterating through a singly-linked list

A list containing three elements:



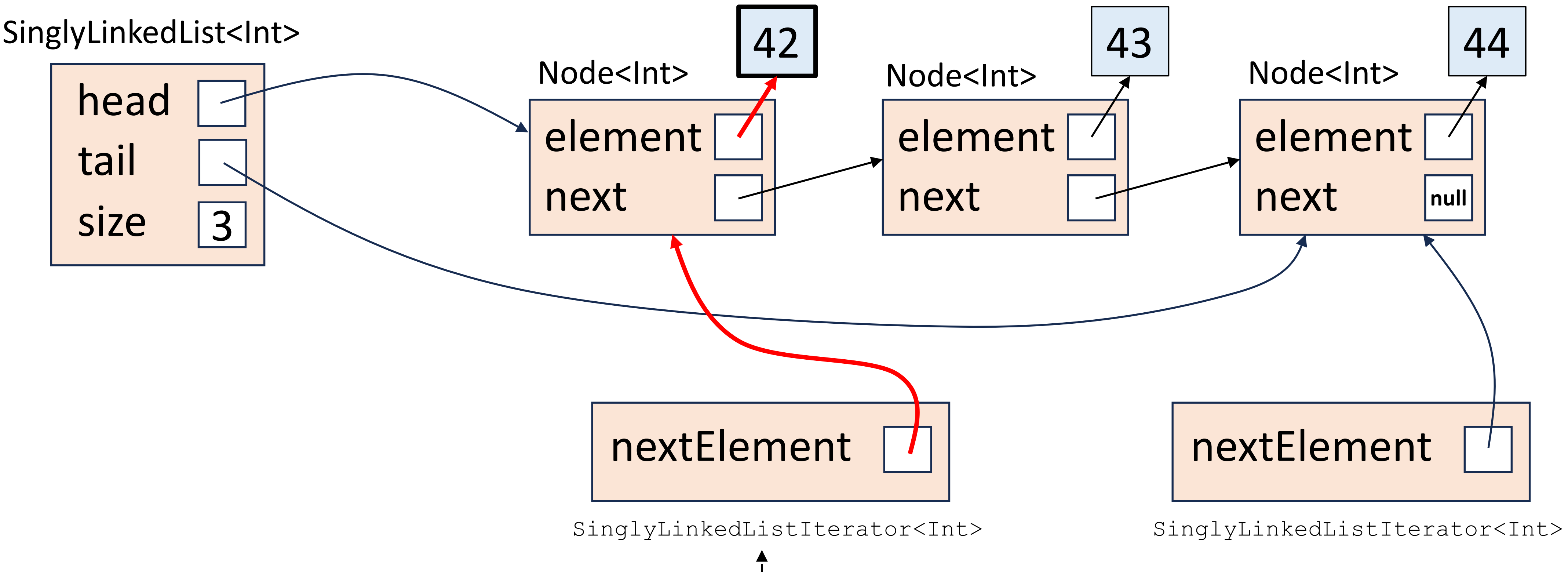
Two iterators referring to different points in this list:

Iterating through a singly-linked list



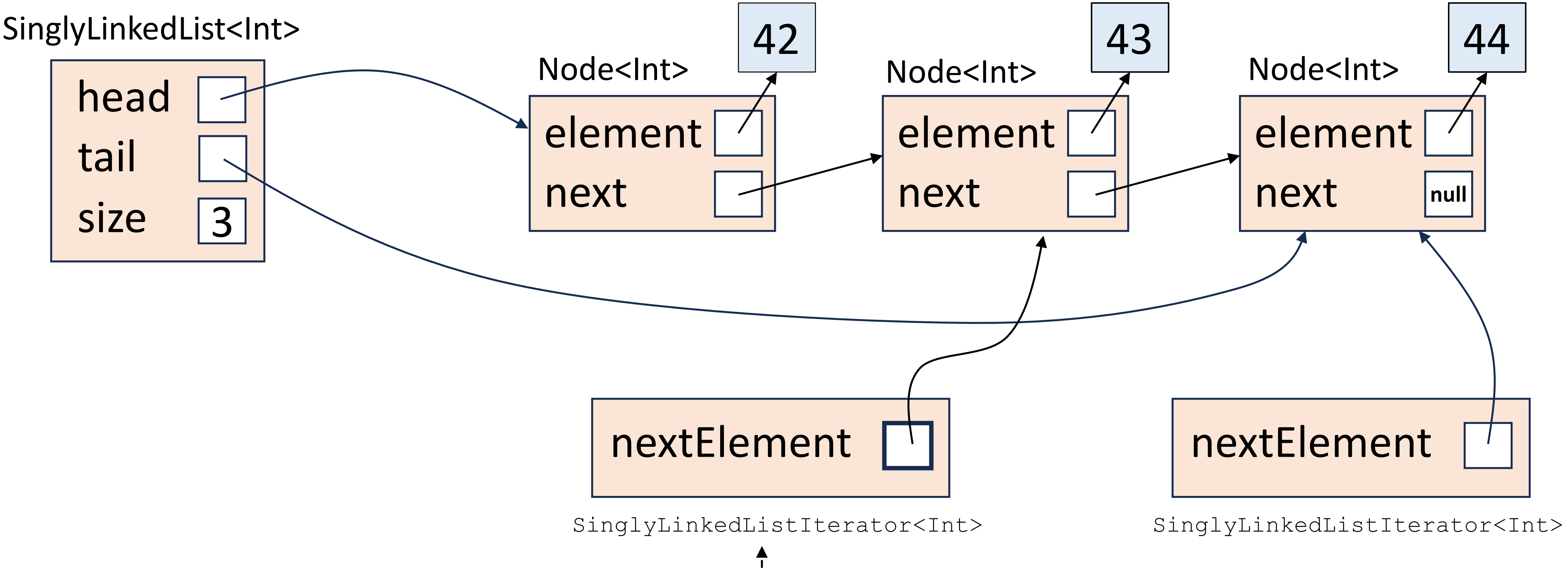
`hasNext()` is **true**: `nextElement != null`

Iterating through a singly-linked list



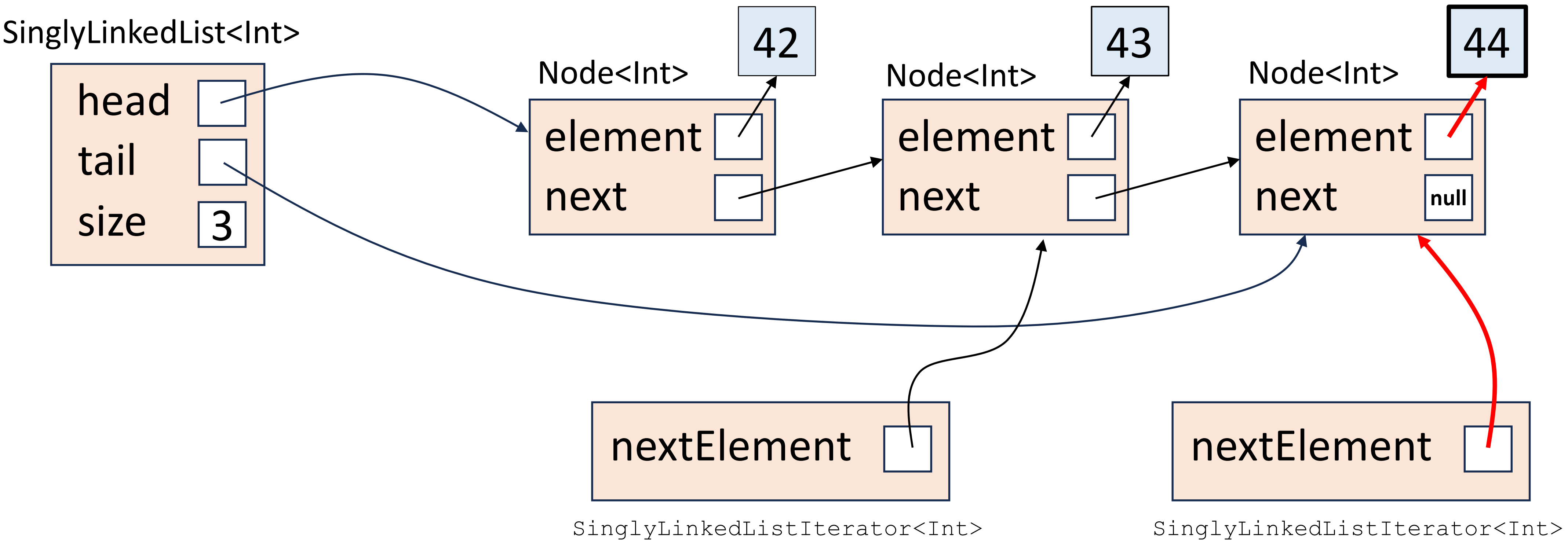
`next()` returns 42

Iterating through a singly-linked list



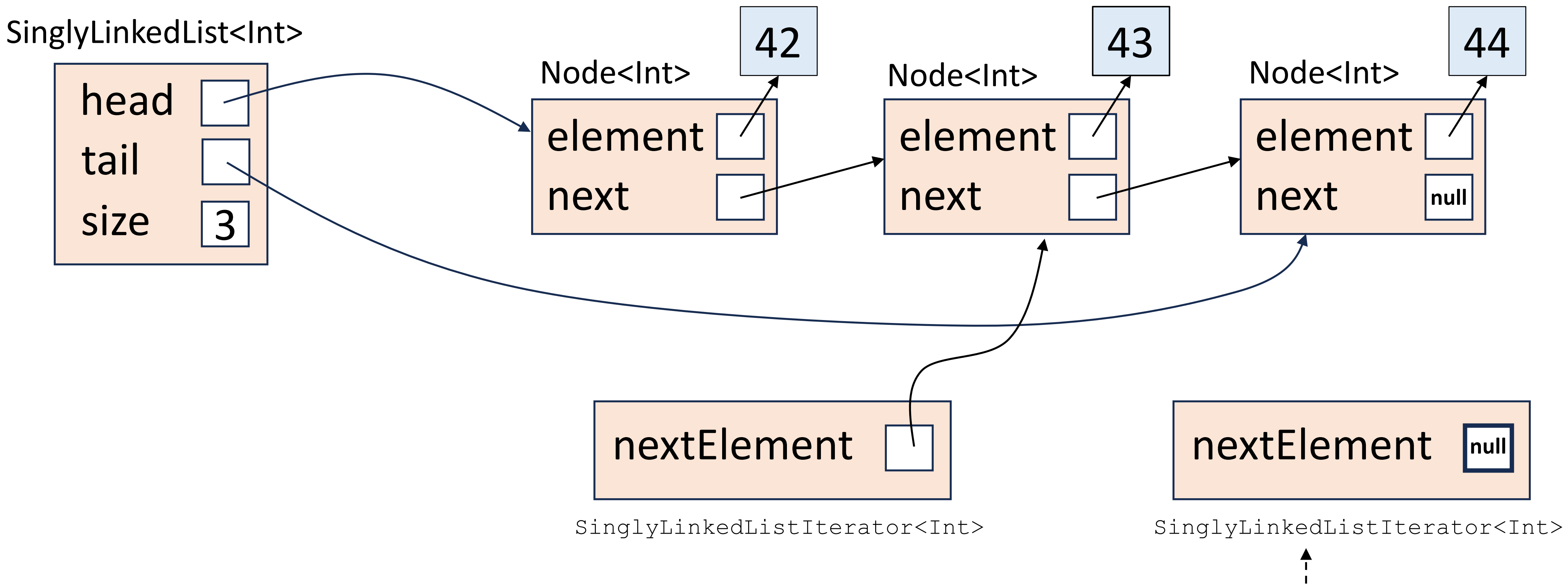
`next()` causes iterator to move on to the next element

Iterating through a singly-linked list



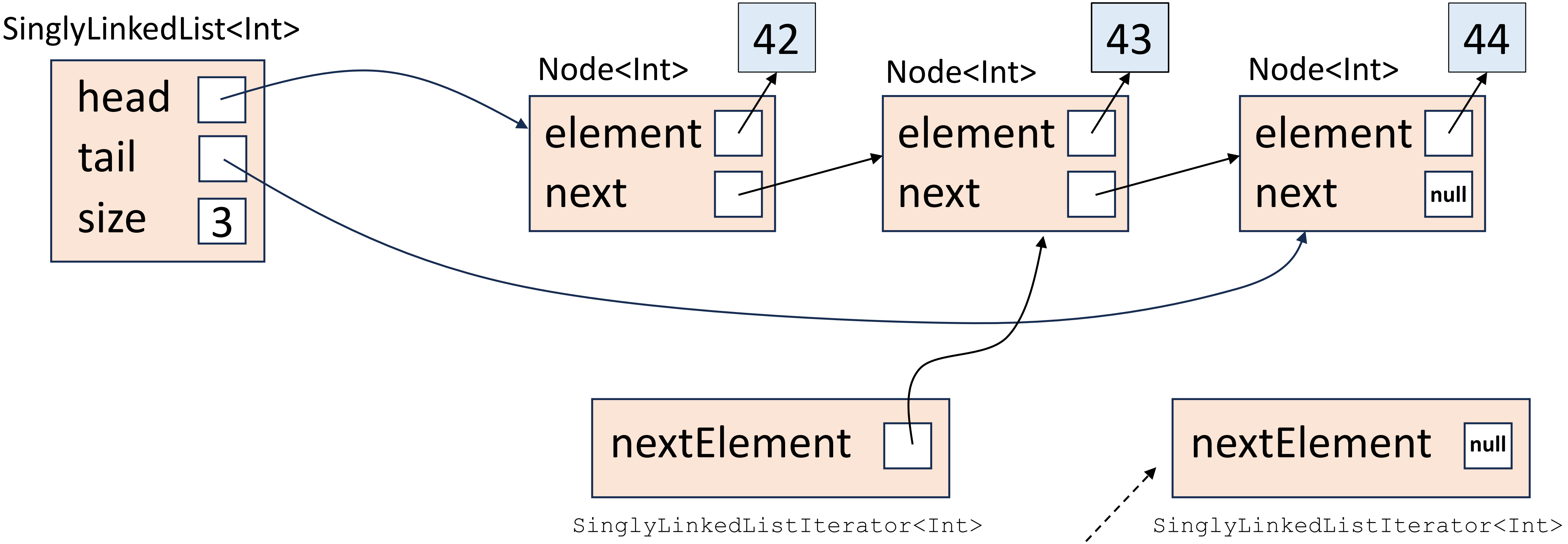
`next()` returns 44

Iterating through a singly-linked list



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

Iterating through a singly-linked list



`hasNext()` is **false**: `nextElement == null`

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

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
}
```

Adding an `iterator()` method to `ImperialMutableList`

```
interface ImperialMutableList<T> {
    val size: Int
    operator fun get(index: Int): T
    fun add(element: T)
    ...
    operator fun set(index: Int, element: T): T
    fun iterator(): Iterator<T>
}
```

This says: "To be an `ImperialMutableList`, you must provide an object that can be used to iterate over you"

`ResizingArrayList` no longer compiles

The Kotlin compiler complains that no implementation of abstract method `iterator` is provided

This is good: clients of `ImperialMutableList` can now assume that every implementing class provides an iterator implementation

This compiler error forces us to add one!

An iterator for `ResizingArrayList`

```
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 = if (!hasNext()) {
        throw NoSuchElementException()
    } else {
        targetList[currentIndex++]
    }
}
```

Provides access to the list being iterated over

Determines the list element the iterator will return next

Oops: `next()` should not have been called!

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)
    ...
}
```

Reminder: `this` refers to the **receiving object**. The `ResizingArrayList` whose `iterator()` method has been called passes a reference to **itself** to the `ResizingArrayListIterator`

Better: use a **private nested class**

Exercise: why is this better?

```
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

Inner class: contrived 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
        }
    }
}
```

Every B instance has an associated A instance – the B responsible for creating it

Each B instance has access to the properties (and methods) of its associated A instance

Inner class: contrived example

Constructing a `B` from an `A`: `myB` has `myA` as associated `A` instance

```
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

Exercise

- Work through the previous `main()`, drawing diagrammatically what goes on in memory: which objects are created and how do they reference one another?
- Confirm that the claimed output is accurate

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
        }
    }
}
```

If we get rid of `inner` and make `B` a nested class, it does not compile

Compile error: Unresolved reference: x

Compile error: Unresolved reference: 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**

Inner class vs. nested class

This attempt to construct a `B` instance is also illegal when `B` is not an inner class of `A`

```
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)
}
```

Again, in the nested case a `B` instance has no associated `A` instance

Looks to compiler like we are trying to call a method named `B` on `myA`

Compile error: Unresolved reference: B

Inner class vs. nested class

```
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

Inner class vs. nested 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 property of the inner class

Refers to property of the enclosing class

Refers to enclosing class instance

The `iterator()` method returning an inner 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: the inner class instance automatically has access to the instance of the enclosing class that created it

Spot the differences!

```
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 = if (!hasNext()) {
        throw NoSuchElementException()
    } else {
        targetList[currentIndex++]
    }
}
```

```
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++]
    }
}
```

Exercise: which version of `next()` is more efficient? Are they equivalent?

```
override fun next(): T = if (!hasNext()) {
    throw NoSuchElementException()
} else {
    this@ResizingArrayList[currentIndex++]
}
```

```
override fun next(): T = if (!hasNext()) {
    throw NoSuchElementException()
} else {
    elements[currentIndex++]!!
}
```

Observation

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()
    ...
}
```

Observation

It would be **wrong** to create a `ResizingArrayListIterator` in any other method of `ResizingArrayList`, but it is **possible**

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

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

Bad! Better if it were **impossible** to make this mistake

Implementing `iterator()` via an **anonymous object**

Instead of declaring an inner class and then returning an instance of it, this directly returns an object that meets the `Iterator<T>` interface requirements

```
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++]!!
    }
    ...
}
```

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 a named inner class declaration!

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

We could still make a different mistake

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

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
    }
}
```

Exercise: try to rewrite this iterator using:

- An inner class (but not an anonymous object)
- A nested class (but not an inner class)
- A separate class in the same file as `SinglyLinkedList`, but not inside the `SinglyLinkedList` class itself

Which of these options are **possible**?

What are the key differences between the approaches that do turn out to be possible?

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 both clunky and error prone

```
val iterator = first.iterator()
while (iterator.hasNext()) {
    result.add(iterator.next())
}
```

What stops us from making a mistake like this?

```
val iterator = first.iterator()
while (iterator.hasNext()) {
    result.add(iterator.next())
    iterator.next()
}
```

Accidental extra call to `next()` – skips every other element of `first`

Making `iterator()` an operator function

A Kotlin convention

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
}
```

Problems with `combine` are now solved!

```
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 enables neat for loop syntax – less error prone

Iterators avoid repeated list traversals: brings complexity down from **quadratic** to **linear**

Let's improve our `addAll` default method

```
interface ImperialMutableList<T> {
    // Other methods and properties as before
    fun addAll(other: ImperialMutableList<T>) {
        for (index in 0..<other.size) {
            add(other.get(index))
        }
    }
}
```

This suffers from problem of repeated calls to `get`, each of which may take linear time (if other is a `SinglyLinkedList`)

Improved `addAll` default method

```
interface ImperialMutableList<T> {
    // Other methods and properties as before
    fun addAll(other: ImperialMutableList<T>) {
        for (element in other) {
            add(element)
        }
    }
}
```

Using an iterator (behind the scenes, thanks to the iterator operator) avoids repeated `get` calls

Exercise: making `addAll` methods efficient

Adapt the following methods to make effective use of iterators:

- Your overridden versions of `addAll` in `ResizingArrayList` and `SinglyLinkedList`
- The default implementation of the `addAll` overload that adds at a given index to make effective use of iterators
- Your overridden versions of these methods in the two implementing classes

Exercise: `map` extension method

In a new file, `ImperialListUtilities.kt`, write the following extension method for `ImperialMutableList<T>`:

- `map`: generic with respect to an additional type `U`; takes a function of type `(T) -> U`; returns an `ImperialMutableList<U>` where each element of the receiving list has been mapped by the function

Use the fact that `ImperialMutableList<T>` has an iterator method to make your implementation efficient

Exercise: `filter` extension method

In `ImperialListUtilities.kt`, write the following extension method for `ImperialMutableList<T>`:

- `filter`: takes a predicate function of type `(T) -> Boolean`; returns an `ImperialMutableList<T>` containing only those items that satisfy the predicate

Use the fact that `ImperialMutableList<T>` has an iterator method to make your implementation efficient

Exercise: `zip` extension method

In `ImperialListUtilities.kt`, write the following extension method for `ImperialMutableList<T>`:

- `zip`: generic with respect to an additional type parameter `U`; takes an `ImperialMutableList<S>`; returns an `ImperialMutableList<Pair<T, U>>` containing pairs of elements from the receiving list and the parameter list, from indices 0 up to the shorter of the two lists

Use the fact that `ImperialMutableList<T>` has an iterator method to make your implementation efficient

Exercise: `reduce` extension method

In `ImperialListUtilities.kt`, write the following extension methods for `ImperialMutableList<T>`:

- `reduce`: takes an accumulator function of type `(T, T) -> T`; returns the value of type `T` obtained by performing a left fold of the accumulator across the elements of the list (i.e., accumulating the first two elements, then accumulating the next element with this result, etc.); throws an exception if the receiving list is empty
- An overload of `reduce` that takes an initial value of type `T` (which would normally be an identity element for the accumulator), and performs a left fold across the list starting with this initial value – this version of reduce can be applied to an empty list, in which case the initial value is returned

Use the fact that `ImperialMutableList<T>` has an iterator method to make your implementations efficient