Resizing Array Lists

Fixed-capacity list:

- not very useful
    - If capacity is too small, we run out of space
    - If capacity is too large, we waste memory

Let’s build a resizing array-based list

- Starts with an array of some initial capacity
- When array is full, switch to new array with double the capacity

A secondary constructor must use this to call some other constructor. Usually it calls the primary constructor – but could e.g. call another secondary constructor that in turn calls the primary constructor.

```
package collections

import java.lang.IllegalArgumentException

class ResizingArrayList<T>(private val initialCapacity: Int) {
    // A secondary constructor
    // A secondary constructor calls primary constructor, via this
    constructor() : this(DEFAULT_INITIAL_CAPACITY)

    // Initialisation block, executed right
    // after primary constructor
    init {
        if (initialCapacity < 0) {
            throw IllegalArgumentException()
        }
    }

    var size = 0
        private set

    private var elements: Array<T?> = clearedArray()
    private fun clearedArray(): Array<T?> =
        arrayOfNulls<Any?>(initialCapacity) as Array<T?>
    // Array initialization logic extracted into a helper method,
```

```
    // clearedArray, as we will use it again later    // Helper
Methods must be private    // they do not provide services to    //
other classes    // This is a helper method - captures common logic
used multiple times    // in the class
    fun get(index: Int): T = if (index !in 0..<size) {
        throw IndexOutOfBoundsException()
    } else {
        elements[index]!!
    }

    fun add(index: Int, element: T) {
        if (index !in 0..size) {
            throw IndexOutOfBoundsException()
        }
        for (i in size downTo index + 1) {
            // If size > elements.size we
            // have an out-of-bounds access
            elements[i] = elements[i - 1]
        }
        elements[index] = element
        size++
    }

    fun addBetter(index: Int, element: T) {
        if (index !in 0..size) {
            throw IndexOutOfBoundsException()
        }
        if (size + 1 > elements.size) {
            elements = elements.copyOf(2 * elements.size)
            // If we need more space,
            // allocate a new array            // twice as large
and copy            // the old array in        }
        for (i in size downTo index + 1) {
            elements[i] = elements[i - 1]
        }
        elements[index] = element
        size++
    }

    fun clear() {
        elements = clearedArray()
        size = 0
    }
}
```

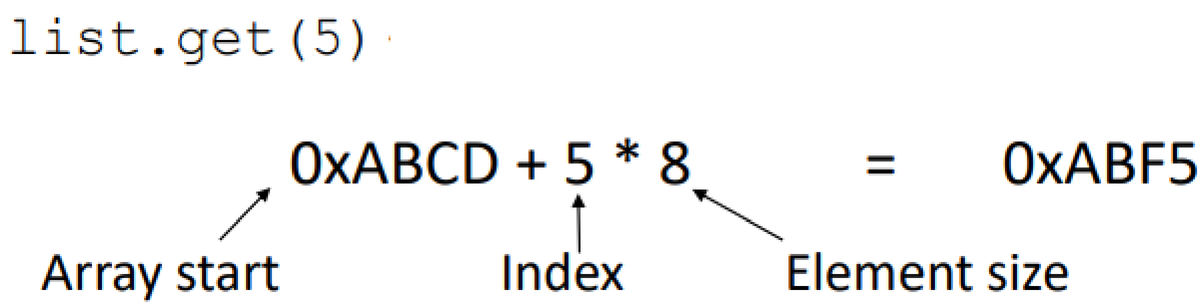
```kotlin
// A read-only property of this file,
// private means visible only within
// the file
private const val DEFAULT_INITIAL_CAPACITY = 16

fun main() {
    val oneList = ResizingArrayList<Int>(1024)
    // Creates a list with capacity 1024
    // using the primary constructor
    val anotherList = ResizingArrayList<Int>()
    // Creates a list with default capacity
    // using the secondary constructor
}
```

Efficiencies of resizing array-based list

Get the element at index i?

Efficient: compute memory location i objects after start of array, return object at that location. For example, if each element is an object reference – 8 bytes on a 64-bit machine



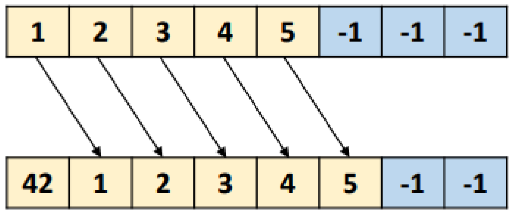
Add an element to the end of the list?

Usually, very efficient: insert element into next free slot. Occasionally, very slow: requires a resize operation. The cost of resizing is **amortized**: resizes are rare and become rarer as size of array is doubled on each resize.

Add an element earlier in the list?

Expensive: all remaining elements must be moved. Worst case: insert at the start of the list – entire contents must be moved.

```
add(index = 0, element = 42)
```

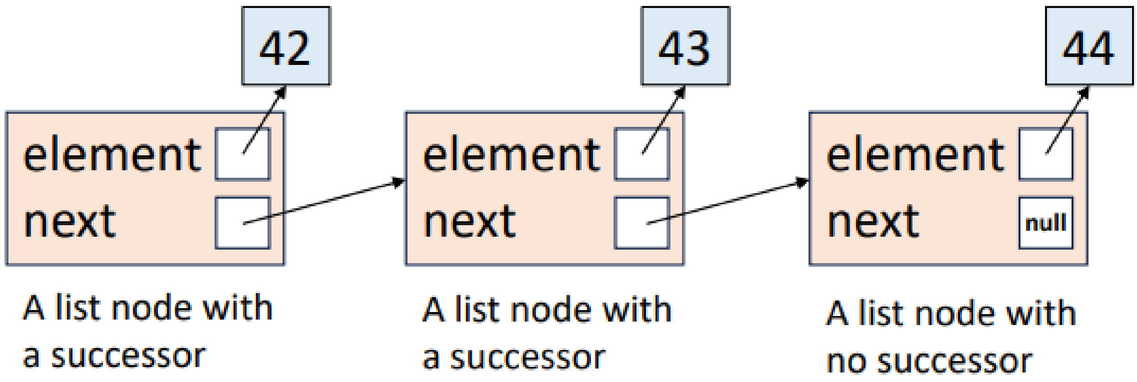


Singly-linked lists

A list is represented as a chain of nodes.

A node is a pair:

- A reference to an object: the element stored by the node
- A nullable reference to the next node; null if node has no successor



The Node Class

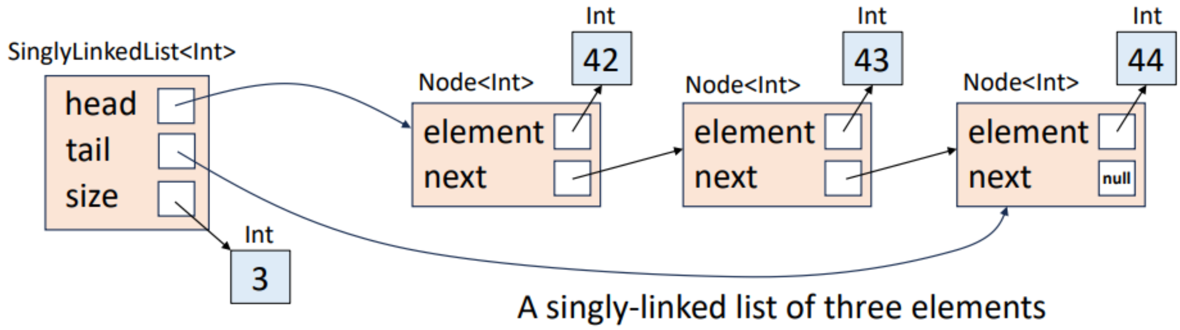
```
class Node<T>(var element: T, var next: Node? = null)
```

- Generic: we can have a node storing elements of any type T
- The default value of next is *null* – this is a default parameter
- The next property is nullable: a node might not have a successor
- We use `var` properties so that the node can be updated – necessary for a mutable list

The singly-linked list Class

Attributes:

- The head of the list: (a reference to) a list node
- The tail of the list: (a reference to) a list node
- The size of the list: (a reference to) an integer
  - Could traverse the list to find size, but this is inefficient
  - Storing the size is much better
  - But the programmer must remember to change the size attribute in methods



A singly-linked list of three elements

A first attempt

```
class Node(var element: T, var next: Node? = null)

class SinglyLinkedList() {
  // Empty primary constructor - better style to omit it
  private var head: Node? = null
  private var tail: Node? = null
  var size: Int = 0
    private set
```

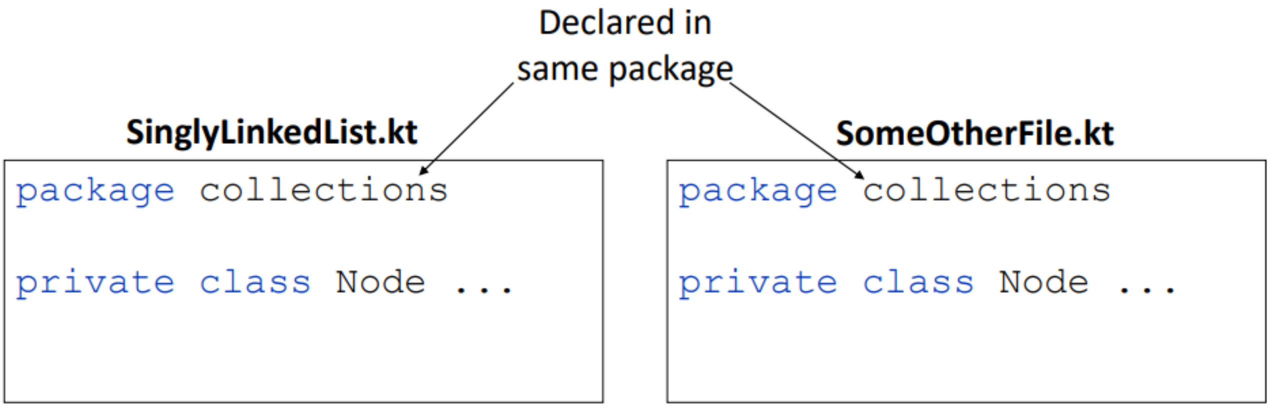
Anyone can create instances of this Node class – that's bad; we only introduced it to support `SinglyLinkedList`.

General rule: Hide internal details of your classes

```
private class Node(var element: T, var next: Node? = null)
```

Now Node is only visible in this Kotlin file.

Name Clashes Between files

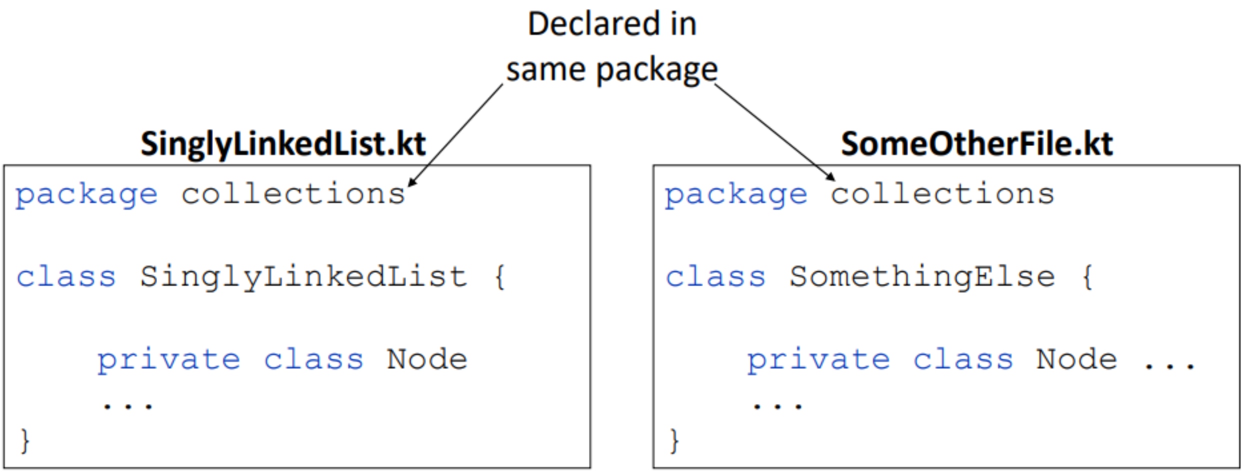


Error: Redeclaration of class Node

Solution: Make Node a private nested class

```
class SinglyLinkedList {
  private class Node(var element: T, var next: Node? = null)
  private var head: Node? = null
  private var tail: Node? = null
  var size: Int = 0
    private set
```

Because Node is nested inside `SinglyLinkedList`, its full name is `SinglyLinkedList.Node`.



No errors.

Add to Singly-Linked List

```
fun add(element: T) {
  size++
  val newNode = Node(element)
  if (head == null) {
    head = newNode
    tail = newNode
    return
  }
  tail!!.next = newNode
  tail = newNode
}
```

Removing from Singly-Linked List

```

class SinglyLinkedList<T> {
    private class Node<T>(var element: T, var next: Node<T>? =
null)

    private var head: Node<T>? = null
    private var tail: Node<T>? = null
    var size: Int = 0
        private set

    fun removeAt(index: Int): T {
        if (index !in 0..<size) {
            throw IndexOutOfBoundsException()
        }
        val (previous: Node<T>?, current: Node<T>?) =
traverseTo(index)
        val result = current!!.element
        unlink(previous, current)
        return result
    }

    private fun traverseTo(index: Int):
        Pair<Node<T>?, Node<T>?> {
        var previous: Node<T>? = null
        var current: Node<T>? = head
```

```
        for (i in 0..<index) {
            previous = current
            current = current!!.next
        }
        return Pair(previous, current)
    }

    private fun unlink(
        previous: Node<T>?,
        current: Node<T>,
    ) {
        if (previous == null) {
            head = current.next
        } else {
            previous.next = current.next
        }
        if (current == tail) {
            tail = previous
        }
        size--
    }
}

fun main() {
    println()
}
```

Efficiencies with singly-linked lists

Get the element at index i?

Inefficient: we need to follow i links

Add an element to the end of the list?

Efficient: link new node to previous tail, update tail property

Add an element earlier in the list?

Efficient if we have a reference to predecessor of insertion point:

- just link in the new node

  Inefficient if all we know is the index i of insertion – need to chase i links.

```
class SinglyLinkedList<T> {
    private class Node<T>(var element: T, var next: Node<T>? =
null)

    private var head: Node<T>? = null
    private var tail: Node<T>? = null
    var size: Int = 0
        private set

    fun removeAt(index: Int): T {
        if (index !in 0..<size) {
            throw IndexOutOfBoundsException()
        }
        val (previous: Node<T>?, current: Node<T>?) =
traverseTo(index)
        val result = current!!.element
        unlink(previous, current)
        return result
    }

    private fun traverseTo(index: Int):
        Pair<Node<T>?, Node<T>?> {
        var previous: Node<T>? = null
        var current: Node<T>? = head
        for (i in 0..<index) {
            previous = current
            current = current!!.next
        }
        return Pair(previous, current)
    }

    private fun unlink(
        previous: Node<T>?,
        current: Node<T>,
    ) {
        if (previous == null) {
            head = current.next
        } else {
            previous.next = current.next
```

```
        }
        if (current == tail) {
            tail = previous
        }
        size--
    }
}

fun main() {
    println()
}
```