Fixed Capacity Lists

Parameter to primary constructor is not a property (no `var` or `val`), no need, since it is only used in the constructor.

```
package collections

class FixedCapacityList(capacity: Int) {

    private val elements = if (capacity < 0) {
        throw IllegalArgumentException("List capacity cannot be
negative")
    }
    else {
        Array(capacity) { -1 }
    }

    var size = 0
        private set

    override fun toString(): String = elements.slice(0..<size)
        .joinToString(
        prefix = '[',
        postfix = ']'
    )

    fun add(index: Int, element: Int) {

        if (size >= elements.size || index !in 0..size) {
            throw IndexOutOfBoundsException()
        }

        for (i in size downTo index + 1) {
            elements[i] = elements[i - 1]
        }
        elements[index] = element
        size++
    }
    // Method Overloading: Same name, different parameters
    fun add(element: Int) = add(size, element)
}

fun main() {
    val data = FixedCapacityList(10)
    println(data.size)
}
```

```
fun get(index: Int): Int = if (index !in 0..<size) {
    throw IndexOutOfBoundsException()
} else {
    elements[index]
}
```

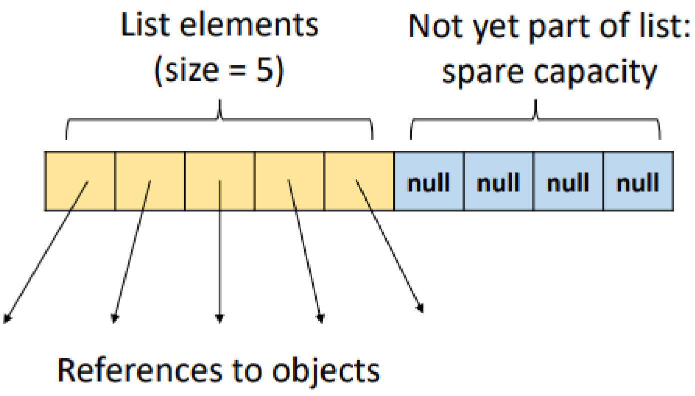
This is neat way to write `0..size - 1`

Generic Fixed Capacity Lists

Any is a Kotlin type that can store a reference to any object – a string, an integer, a person, a point – anything • If we write a fixed-capacity list of Any, wouldn't this work for every type?

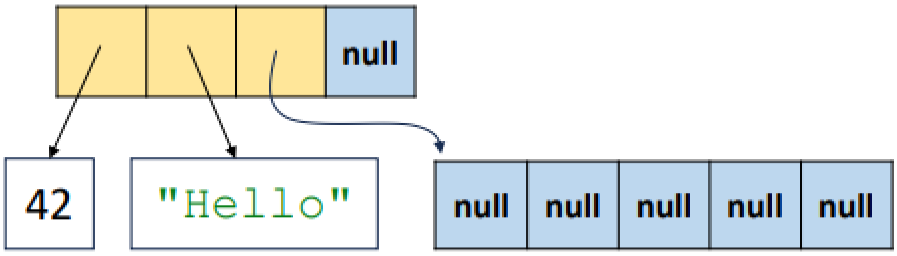
Invariant maintained by fixed-capacity list of Any

For all $0 \leq i <$ **capacity**, **elements**[i] = null iff $i \geq$ **size**



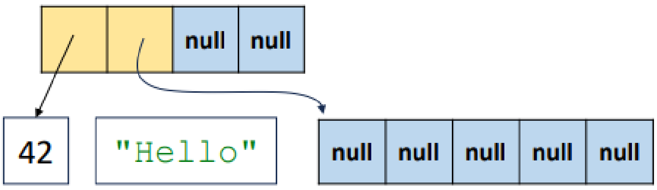
Can make this:

)



Our list of **Any** now contains an integer, a string, and another list

```
stuff.removeAt(1)
```



"Hello" = garbage = removed from list, but still exists in memory = inaccessible = Cleaned up via garbage collection

```
class FixedCapacityAnyList(capacity: Int) {
    var size: Int = 0
        private set

    // The type of array elements is “nullable
    // Any” – the array can store references     // to any objects,
as well as null values
    private val elements: Array<Any?> = if (capacity < 0) {
        throw IllegalArgumentException()
    } else {
        // Creates an array of size capacity,
        // that is null everywhere
        arrayOfNulls(capacity)
    }
    fun get(index: Int): Any = if (index !in 0..<size) {
      throw IndexOutOfBoundsException()
    } else {
      elements[index]!!
      // !! Asserts expression is non-nullable
      // If you are wrong and the expression is Null, a
NullPointerException
      // is thrown
      // Because of our invariant, this will work
    }
}
```

Type Safety

```
fun main() {
    val myStrings = FixedCapacityAnyList(10)
    myStrings.add("Minty")
    myStrings.add("Jekyll")
    myStrings.add(42)
    myStrings.add(Pair("Cat", "Dog"))
    // No Type safety! Can add anything!
    val upperCaseMinty = myStrings.get(0).uppercase()
    // Can't do this, since Any does not have an uppercase() method
    val upperCaseMinty = (myStrings.get(0) as String)
        .uppercase()
    // Can do this
    // But if you are wrong, program compiles, but crashes at
runtime}
```

Generic Fixed Capacity Lists with Type Safety

```
class FixedCapacityList<T>(capacity: Int) {
    var size: Int = 0
        private set
    private val elements: Array<T?> = if (capacity < 0) {
        throw IllegalArgumentException()
    } else {
        arrayOfNulls<Any?>(capacity) as Array<T?>
        // Need this due to reasons related to Kotlin / Java
interoperability
    }
}

fun main() {
    val myStrings = FixedCapacityList<String>(10)
    myStrings.add("Minty")
    myStrings.add("Jekyll")
    // This works!
    myStrings.add(42)
```

```kotlin
    myStrings.add(Pair("Cat", "Dog"))
    // This does not
    // The Kotlin compiler gives type errors    // Detecting problems at compile time is good - avoids debugging runtime failures
    val upperCaseMinty = myStrings.get(0).uppercase()
    // Works, no casting required
}
```