Kotlin recap: fixed-capacity lists

Alastair F. Donaldson

Aims of this lecture

- Remind you of some Kotlin syntax and features
- Show how to implement a simple fixed-capacity list
- Use exceptions to deal with bad parameters and out-of-bounds access
- Discuss the **Any** type and the use of **Any** to represent a general list
- Briefly introduce casting
- Use generics to write a properly generic list

Fixed-capacity list of integers

To recap some Kotlin, let's write a class representing a list of integers, with a fixed capacity

Representation:

- Array **elements** of N integers – N is the *capacity* of the list
- Integer **size** indicating which elements are part of the list
- List contents: elements 0 .. **size** - 1

Fixed-capacity list of integers

Free slot: not yet part of list

Stores a list element

An empty list with capacity 10:

size 0

-1 is used here as a default value (any value would work)

After adding the number 5:

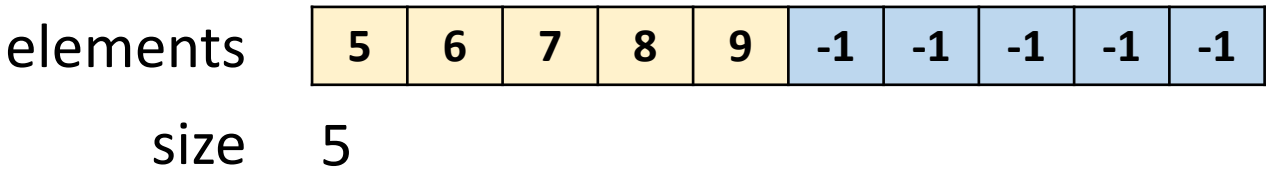
size 1

Fixed-capacity list of integers

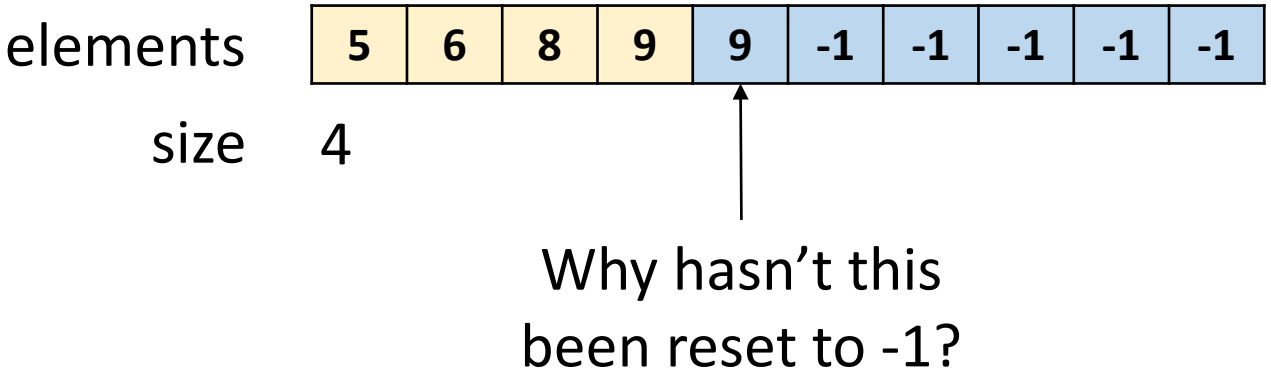
Free slot: not yet part of list

Stores a list element

After adding a few more numbers:



After removing the item at index 2:

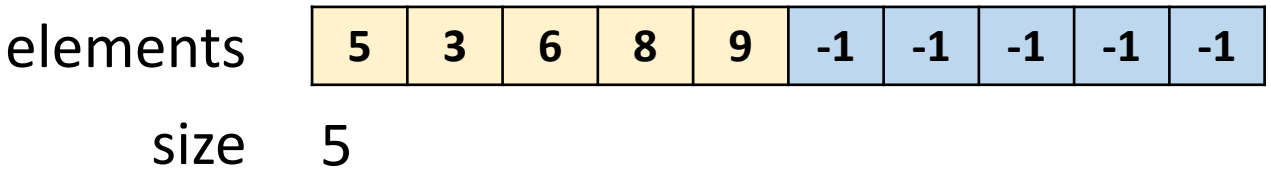


Fixed-capacity list of integers

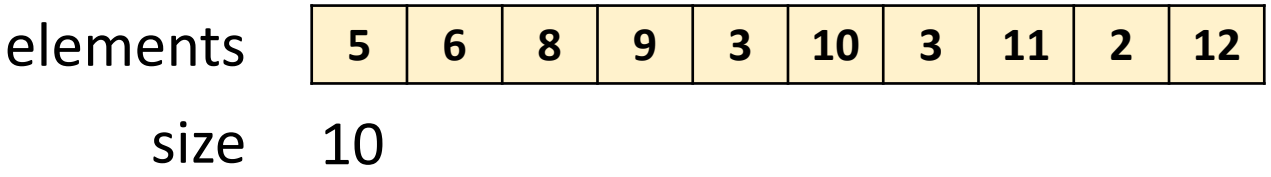
Free slot: not yet part of list

Stores a list element

After adding 3 at index 1:



After adding several more numbers:



Fixed-capacity list of integers

Free slot: not yet part of list

Stores a list element

The list is full – adding an element would lead to an exception being thrown.

Clearing the list just involves setting size to 0:

Let’s write a Kotlin class with code to …

- **Construct** an empty list with a given capacity
- **Add** an integer at a given list index (shifting remaining elements up)
- **Add** an integer to the end of the list
- **Get** the integer at a given index in the list

Constructor and properties

```
package collections

class FixedCapacityIntList(capacity: Int) {

    var size: Int = 0
        private set

    private val elements: Array<Int> = if (capacity < 0) {
        throw IllegalArgumentException()
    } else {
        Array(capacity) { -1 }
    }
```

Parameter to primary constructor, not a property (no **val** or **var**)

Property with **public** read access and **private** write access

The **elements** property is read-only: the contents of the array can change, but it will always be the same array

Understanding `Array(capacity) { -1 }`

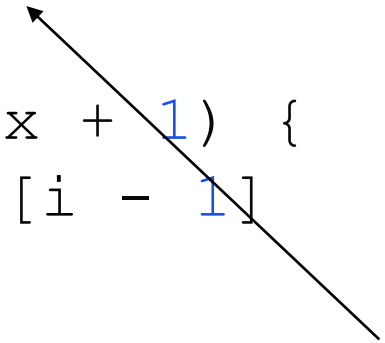
These are all equivalent:

```
Array(capacity) { -1 }
Array(capacity, { -1 })
Array(capacity, { index -> -1 })
Array(capacity, { index: Int -> -1 })
```

When a lambda is the final argument to a function, Kotlin style is to write it after the parameter list: `(…) { … }`

Adding at an index

```
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++
}
```



This is an example of **throwing** an exception

Adding at the end of the list

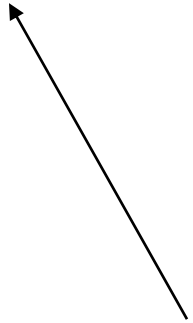
```
fun add(element: Int) = add(size, element)
```

The add method is **overloaded** – here are the two **overloads**:

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

fun add(element: Int)
```

Getting the element at an index

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



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

Exercise: implement these additional methods

- `clear`: resets the list to an initial state (think about how to do this efficiently).
- `contains`: takes an integer element argument. Returns true if and only if the element is present in the list.
- `removeAt`: takes an integer argument – an index. Throws an `IndexOutOfBoundsException` if the index is not in the range [0, `size`). Otherwise, removes the element at the given index by shifting all further elements of the list back one place, and decreasing the size of the list accordingly. Returns the element that was removed.
- `remove`: takes an integer argument – an element to be removed from the list. Removes the first occurrence of this element from the list, if any exists. Returns true if and only if something was removed.
- `set`: takes two argument – an index and an element. Throws an `IndexOutOfBoundsException` if the index is not in the range [0, `size`). Otherwise, updates the list at the given index to hold the given element.
- `toString`: returns a string representation of the list as comma-separated sequence of values, enclosed in square brackets, with a space after each comma. You will need to mark this method as override since every Kotlin class has a default `toString` method, whose behaviour you will override.

What about a fixed-capacity list of strings?

- The code will be very similar
- It would be a pain to have to write a separate list class for every different data type!

Poor person’s generic fixed-capacity list

- **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?

Fixed-capacity list of **Any**

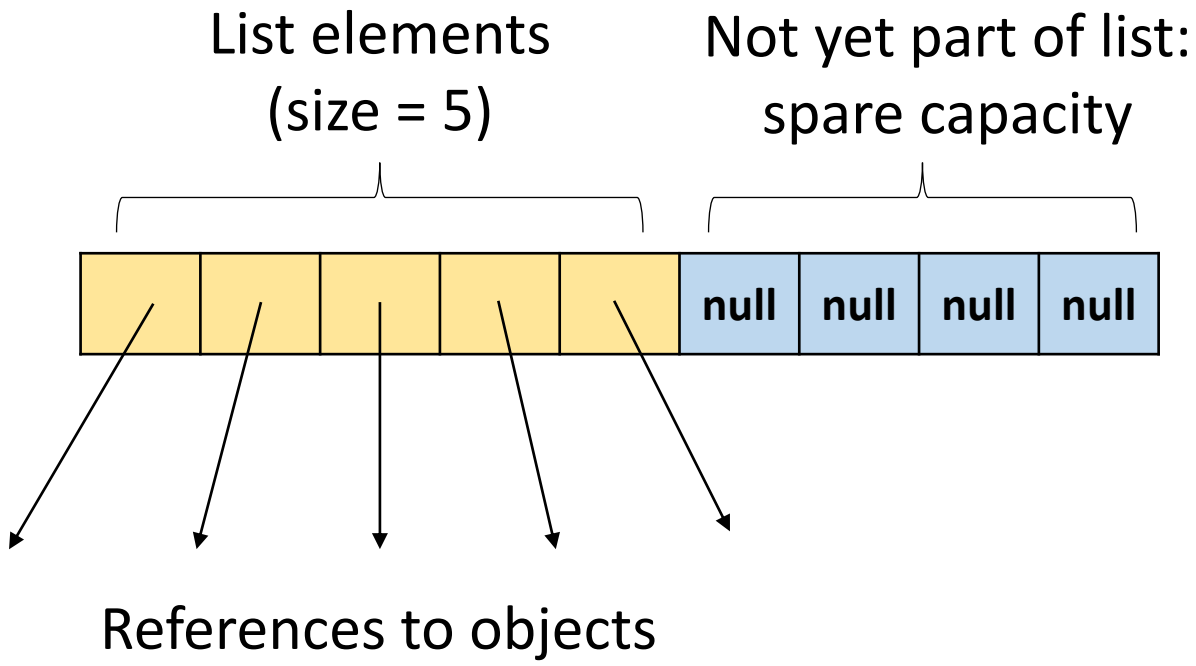
The type of array elements is “nullable **Any**” – the array can store references to any objects, as well as null values

```
package collections

class FixedCapacityAnyList(capacity: Int) {

    var size: Int = 0
        private set

    private val elements: Array<Any?> = if (capacity < 0) {
        throw IllegalArgumentException()
    } else {
        arrayOfNulls(capacity)
    }
```

Creates an array of size capacity, that is null everywhere

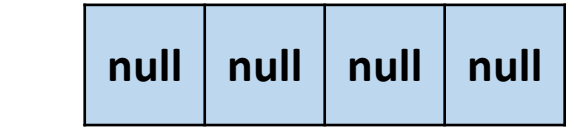
Invariant maintained by fixed-capacity list of **Any**

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

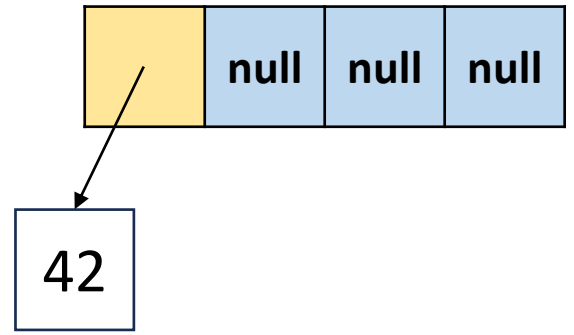


What does this look like in action?

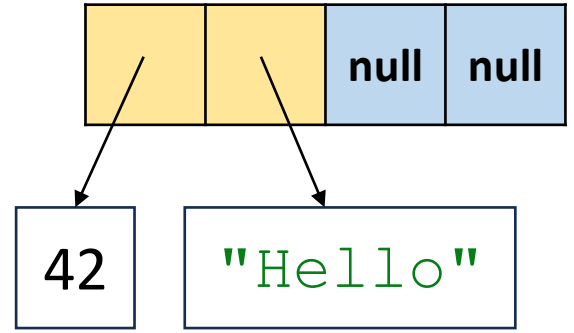
```
val stuff = FixedCapacityAnyList(4)
```




```
stuff.add(42)
```

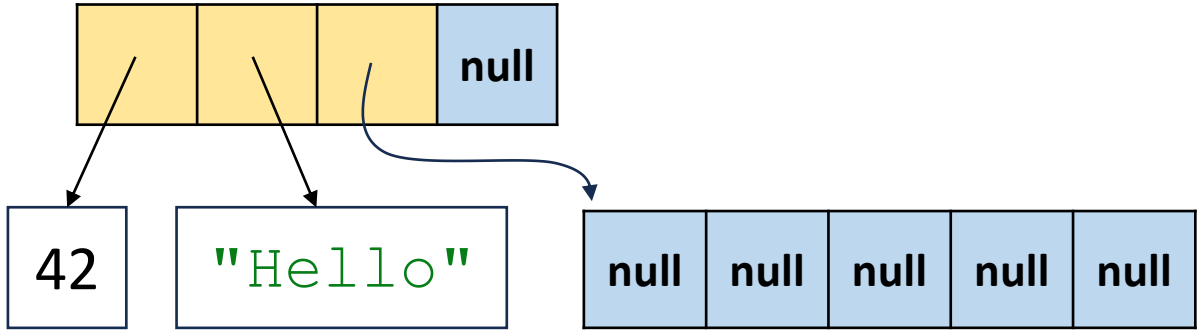



```
stuff.add("Hello")
```



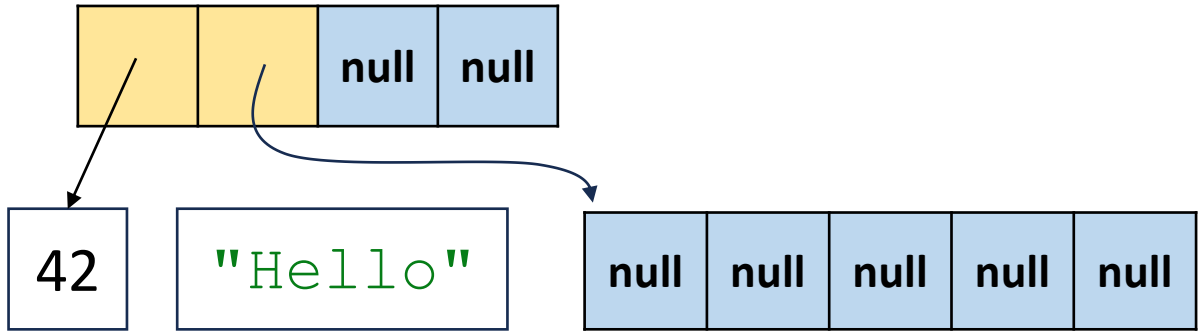
What does this look like in action?

```
stuff.add(
    FixedCapacityAnyList(5)
)
```



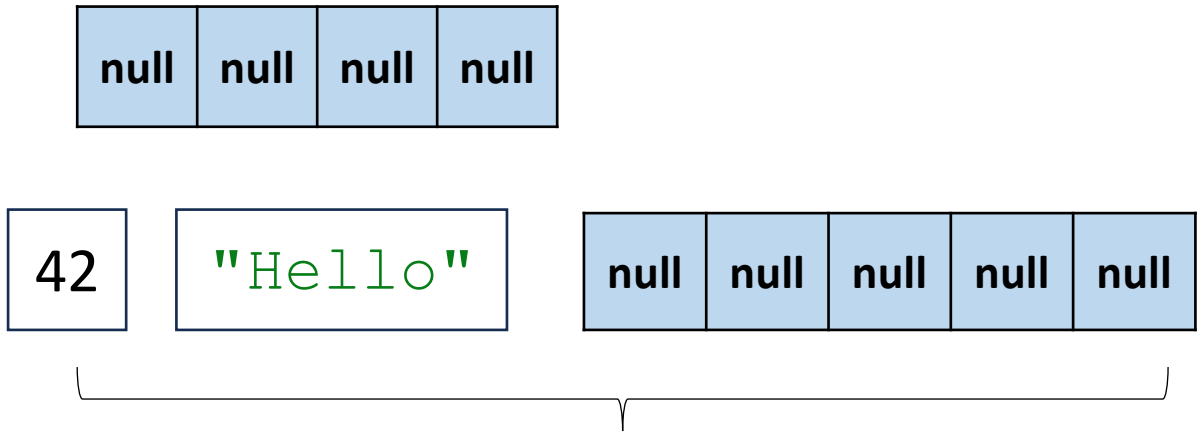
Our list of **Any** now contains an integer, a string, and another list

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



What does this look like in action?

```
stuff.clear()
```

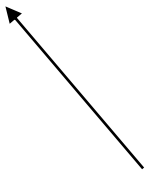


These inaccessible objects are known as **garbage**

They will be cleaned up via garbage collection – more on that later

Fixed-capacity list of **Any**

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



!! asserts that an expression is non-nullable

Non-nullable assertions

Suppose an expression **e** has type **T?**

This means either:

- **e** refers to an object of type **T**, or
- **e** is null

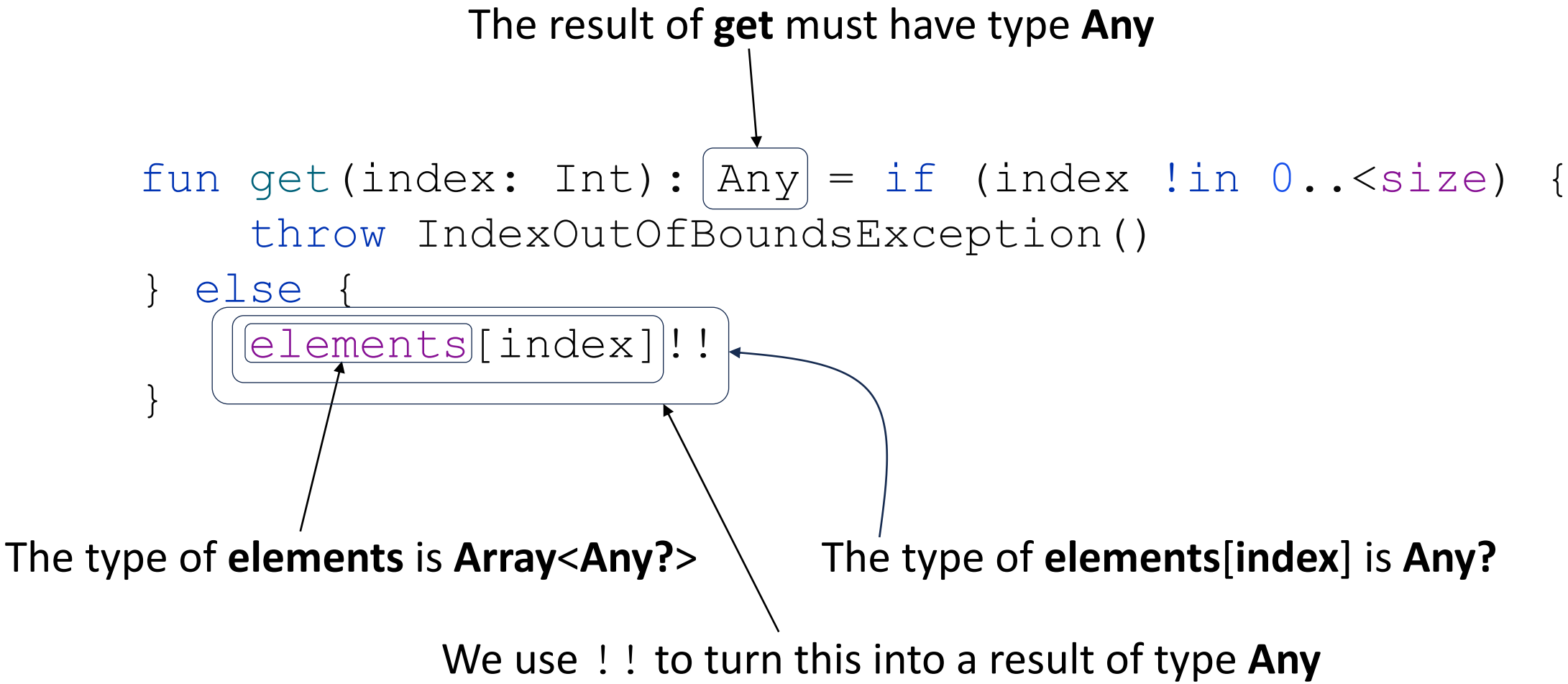
If you – the programmer – are *sure* **e** will not be null, you can write:

e!!

If you are wrong and **e** is null, a **NullPointerException** is thrown

Otherwise, the result of **e**!! has type **T**: it is the object reference

Revisiting this use of non-nullable assertion



Could this fail?

Remember our **invariant** for the class:

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

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


Control only reaches here if $0 \leq$ **index** $<$ **size**

The invariant ensures that **elements**[**index**] will *not* be null

It is up to you – the programmer – to get this reasoning right

A list of **Any** provides no type safety

We need a list of strings? Best we can do is indicate that using a name:

```
val myStrings = FixedCapacityAnyList(10)
```

We can certainly put strings into the list:

```
myStrings.add("Minty")
myStrings.add("Jekyll")
```

But nothing stops us from putting other stuff in by accident:

```
myStrings.add(42)
myStrings.add(Pair("Cat", "Dog"))
```

A list of **Any** provides no type safety

Unresolved reference: **Any** does not provide an **uppercase** method

We cannot easily work with the strings in our list:

```
val upperCaseMinty = myStrings.get(0).uppercase()
```

We need to use explicit casts:

```
val upperCaseMinty = (myStrings.get(0) as String)
                        .uppercase()
```

If you get casting wrong: program compiles, but crashes at runtime!

```
val upperCase42 = (myStrings.get(2) as String)
                      .uppercase()
```

ClassCastException: **Int** cannot be cast to **String**

A truly generic fixed-capacity list

This class is *generic* with respect to *type parameter* **T** – we can have a list of any type we want

```
class FixedCapacityList<T>(capacity: Int) {

    var size: Int = 0
        private set

    private val elements: Array<T?> = if (capacity < 0) {
        throw IllegalArgumentException()
    } else {

    }
```

What should go here?

This backing array can contain nulls, or references to objects of type **T**

A truly generic fixed-capacity list

```
class FixedCapacityList<T>(capacity: Int) {

    var size: Int = 0
        private set

    private val elements: Array<T?> = if (capacity < 0) {
        throw IllegalArgumentException()
    } else {
        arrayOfNulls(capacity)
    }
```

This is what we would like! Unfortunately, it does not work for reasons related to Kotlin / Java interoperability – more on that later

A truly generic fixed-capacity list

```
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?>
    }
```

As a workaround we have to make an **Array<Any?>** and *cast* it to **Array<T?>**

Exercise: complete the `FixedCapacityList<T>` class

- Use your **FixedCapacityIntList** class as a starting point
- Change all uses of **Int** that refer to a list element to use **T** instead
- The changes required beyond that should be fairly minimal – mainly to do with nullable types

No more type safety problems!

We need a list of strings? No problem:

```
val myStrings = FixedCapacityList<String>(10)
```

We can put strings into the list:

```
myStrings.add("Minty")
myStrings.add("Jekyll")
```

Signature of **add** in **FixedCapacityList<T>**: `add(element: T)`

Because **myStrings** has type **FixedCapacityList<String>**, the signature for **add** when applied to **myStrings** is `add(element: String)`

No more type safety problems!

We cannot put other stuff in by accident:

```
myStrings.add(42)
myStrings.add(Pair("Cat", "Dog"))
```

Again, signature of **add** in **FixedCapacityList<T>**: `add(element: T)`

Signature for **add** when applied to **myStrings** is `add(element: String)`

The Kotlin compiler gives type errors: **Int** and **Pair** passed when String was expected

Detecting problems at compile time is good – avoids debugging runtime failures

No more type safety problems!

We can easily work with the strings in our list – no casting required:

```
val upperCaseMinty = myStrings.get(0).uppercase()
```

Signature of **get** in **FixedCapacityList<T>**: `get(index: Int): T`

Signature for **get** when applied to **myStrings** is `get(index: Int): String`

The compiler knows that the returned value is a string, so the **uppercase** method exists