Interfaces

Clients

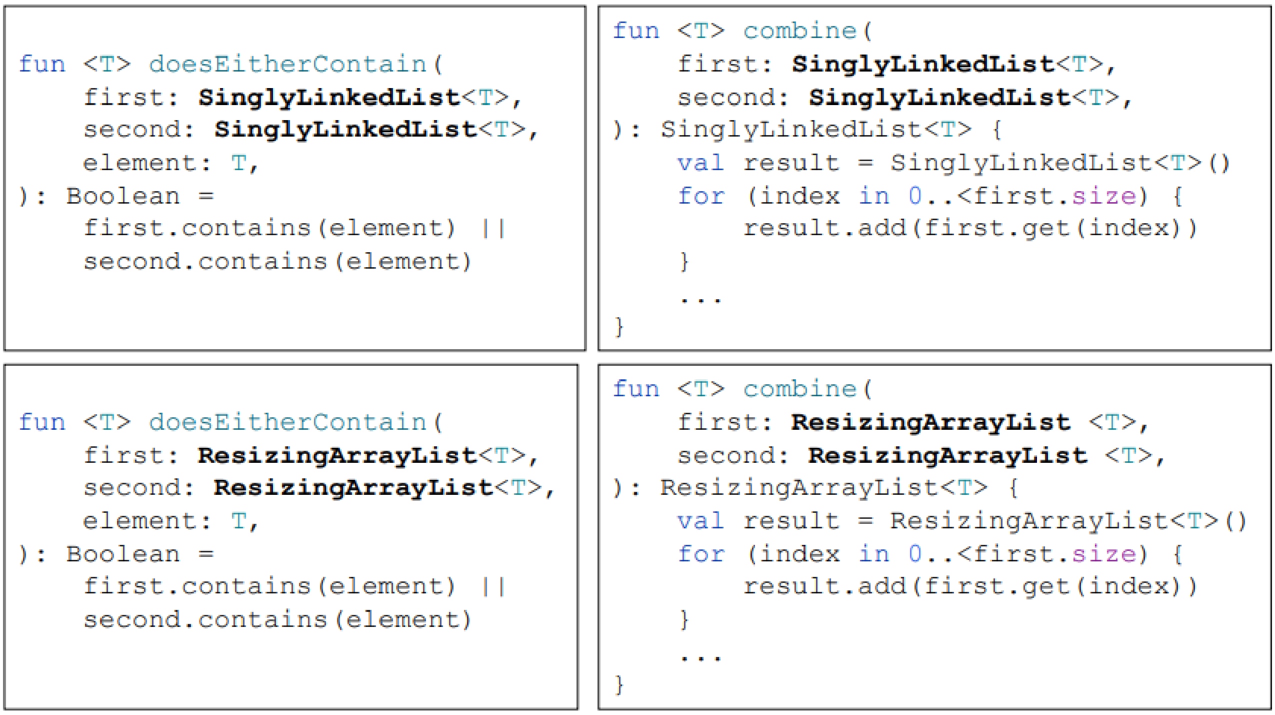
If class A uses class B, we say that A is a client of B. We also say that B provides a service to A.

When designing a class, what services does the class provide to its clients?

- The service is provided via the public stuff
- Everything else – all internal details – should be private

```
fun <T> doesEitherContain(
  first: SinglyLinkedList<T>,
  second: SinglyLinkedList<T>,
  element: T,
  ): Boolean = first.contains(element) || second.contains(element)
```

This is how to write a stand-alone function that is generic with respect to some type T



```
fun <T> doesEitherContain(
    first: SinglyLinkedList<T>,
    second: SinglyLinkedList<T>,
    element: T,
): Boolean =
    first.contains(element) ||
    second.contains(element)
```

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

```
fun <T> doesEitherContain(
    first: ResizingArrayList<T>,
    second: ResizingArrayList<T>,
    element: T,
): Boolean =
    first.contains(element) ||
    second.contains(element)
```

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

Bad: Lots of duplication.

Better Solution: A mutable list interface

```
interface ImperialMutableList<T> {
  val size: Int
  fun get(index: Int): T
  fun add(element: T)
  fun add(index: Int, element: T)
  fun clear()
  fun contains(element: T): Boolean
  fun removeAt(index: Int): T
  fun remove(element: T): Boolean
}
```

None of the methods have bodies. They simply describe the services that a mutable list promises should provide. These are called abstract methods.

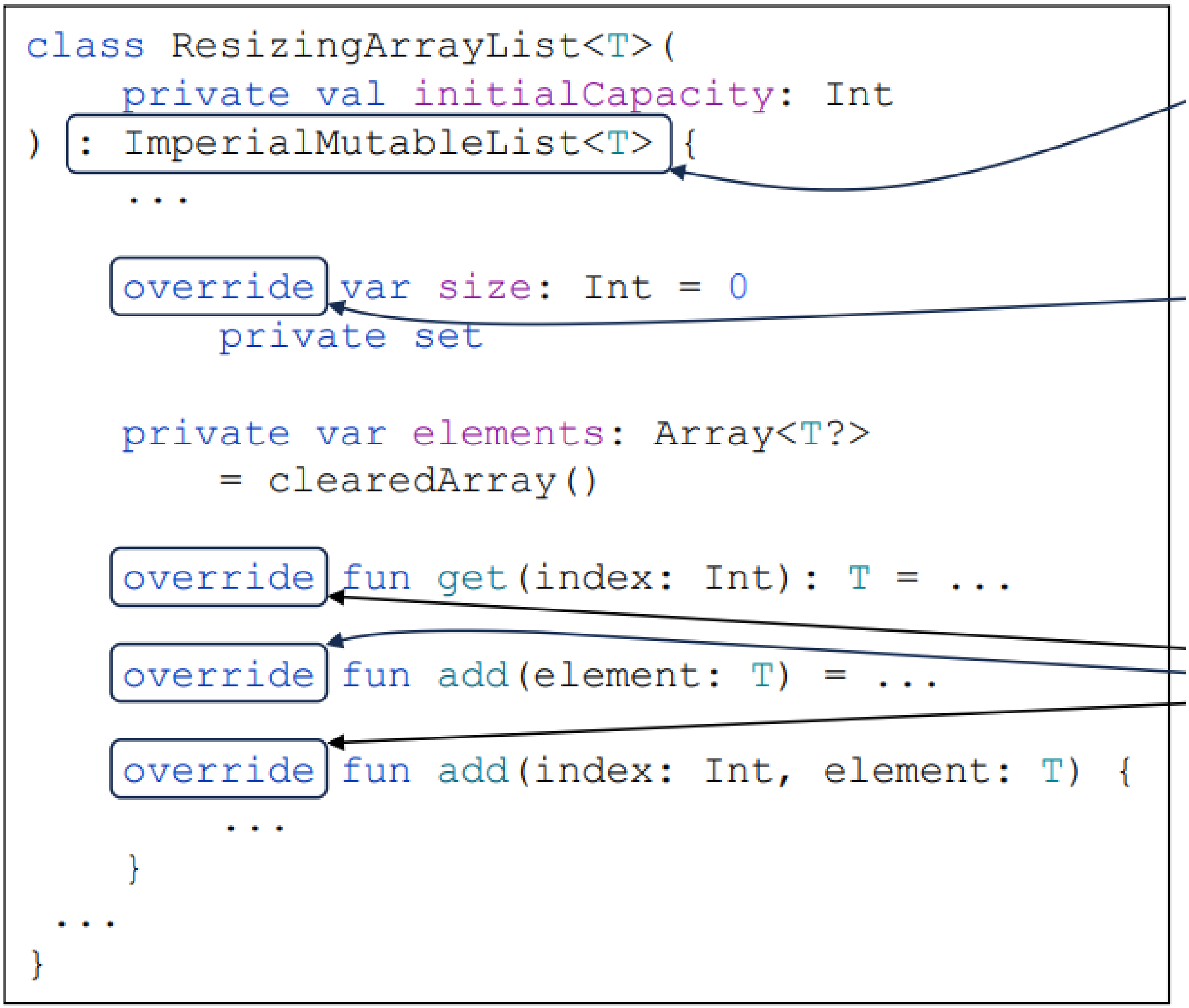
`val size: Int`

This means: to be an `ImperialMutableList`, a class must provide read access to a size property (given by the `val`). Clients of a mutable list should be able to read its size The size may change (due to add and remove calls) But a client should not be able to change the size property directly

The rest means: to be an `ImperialMutableList`, a class must provide implementations of all of these methods. If a client has a reference to an `ImperialMutableList` object, it can depend on these operations being available.

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

`:` = 'intends to implement this interface' = a promise to provide everything `ImperialMutableList` demands.



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

    override var size: Int = 0
        private set

    private var elements: Array<T?>
        = clearedArray()

    override fun get(index: Int): T = ...

    override fun add(element: T) = ...

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

This fulfils the promise of read access to a size property. Private write access is also provided – that's fine.

These methods implement the required methods of the interface.

Override Keyword

When you write an interface, you must annotate each implementation of the interface methods and attributes with `override`. This is because interfaces can also provide default method implementations.

All interface methods and attributes must be implemented.

Otherwise, Error at compile time.

Interface Instantiation

Cannot directly create an instance of an interface type. Instead, we must create an instance of a class that implements the interface type.

Otherwise, error at compile time.

Interfaces Example

Suppose a document management application manages various kinds of page elements:

- Text box – has a width, height, and maximum number of characters
- Image – has a width, height, and filename

```
package kotlin

class TextBox(
    val width: Int,
    val height: Int,
    val maxChars: Int, // Good Style to have extra comma
)

class Image(
    val width: Int,
    val height: Int,
    val filename: String,
)

class DocumentManager {
    private val textBoxes: MutableSet<TextBox> = mutableSetOf()
    private val images: MutableSet<Image> = mutableSetOf()
    fun addTextBox(textBox: TextBox) = textBoxes.add(textBox)
    fun addImage(image: Image) = images.add(image)
}
```

How can we find the height of the tallest page element?

Identical computation for text boxes and images.

```
fun maxHeight(): Int =
    max(
        textBoxes.map { it.height }.max(),
        images.map { it.height }.max(),
    )
```

Duplication is bad:

- Makes software difficult to maintain

Adding Menus

```
fun maxHeight(): Int =
    listOf(
        textBoxes.map { it.height }.max(),
        images.map { it.height }.max(),
        menus.map { it.height }.max(),
    ).max() // Lots of Duplication
```

A lot of duplicate code

- `DocumentManager` needs to be explicitly aware of all the different sorts of page elements that exist
- If we introduce a new page element, we need to change `DocumentManager`
- Makes it difficult for third parties to contribute page elements

Suppose we want to determine whether one page element is taller than another, mixing page element types:

```
fun tallerThan(first: TextBox, second: TextBox) =
    first.height > second.height
fun tallerThan(first: TextBox, second: Image) =
    first.height > second.height
fun tallerThan(first: TextBox, second: Menu) =
    first.height > second.height
fun tallerThan(first: Image, second: TextBox) =
    first.height > second.height
// and so on - 9 methods total!
```

The methods are all the same. We have to overload `tallerThan` for each pair of types.

N kinds of page element → N^2 `tallerThan` methods

Implementation 2

A `TextBox` is a page element
An `Image` is a page element
A `Menu` is a page element

`TextBoxes`, `Images` and `Menus` are not the same, but are similar:

- All have widths and heights
- Want to be able to talk about a page element, and look at its width and height without caring which specific kind of page element it is

Solution: `PageElement` interface

```
interface PageElement {
    val width: Int
    val height: Int
}
```

```
class TextBox(
    override val width: Int,
    override val height: Int,
    val maxChars: Int
) : PageElement
```

The promised properties (`width` and `height`) are provided.

`val maxChars: Int` is specific to `TextBox`.

New `DocumentManager`

```
class DocumentManager {
    private val pageElements: MutableSet<PageElement> =
        mutableSetOf()
    fun addPageElement(pageElement: PageElement) =
        pageElements.add(pageElement)
    fun maxHeight(): Int =
        pageElements.map { it.height }.max()
}
```

One set of `PageElements`. One method for adding `PageElements`. `maxHeight` function is much nicer. We can map once to get the heights of all page elements.

The page elements may have a variety of different types (`TextBox`, `Image`, `Menu`, other page elements). The `PageElement` interface allows us to treat them all uniformly (polymorphism).

```
fun tallerThan(first: PageElement, second: PageElement) =
first.height      > second.height
```

Now, this single method suffices, no matter how many kinds of `PageElement`s we have.

Add new Page Elements

If we add another page element, no changes needed to `DocumentManager`.

Advantages of Interfaces

- Helps us manage complexity by treating all objects uniformly
- Methods and properties common to all the classes are specified in an interface
- Each class implements the interface
- Client code (e.g. `DocumentManager`) can refer solely to the interface without knowing or caring about details of the implementing classes
- Which methods and properties get invoked at runtime depends on details of implementing classes
- This is a form of polymorphism

Default Methods in Interfaces

The `ImperialMutableList` interface lacks an `isEmpty()` method. Implement this: check `size <= 0` . We can add this as a default method.

```
interface ImperialMutableList<T> {
    val size: Int
    ...
    fun isEmpty(): Boolean = size <= 0
}
```

`isEmpty()` is a default method because **it has an implementation**.

```
fun addAll(other: ImperialMutableList<T>) {
    for (index in 0..<other.size) {
        add(other[index])
    }
}
```

This is one way to add one list to another. It works, but for a specific list there might be a better way.

```
interface PageElement {
    val width: Int
    val height: Int
}

class TextBox(
    override val width: Int,
    override val height: Int,
    val maxChars: Int,
) : PageElement

class Image(
    override val width: Int,
    override val height: Int,
    val filename: String,
) : PageElement

class Menu(
    override val width: Int,
    override val height: Int,
) : PageElement {

    private val options: MutableList<String> =
        mutableListOf()
    fun addOption(option: String) {
        options.add(option)
    }
    fun hasOption(candidateOption: String) =
        options.contains(candidateOption)
}

class DocumentManager {
    private val pageElements: MutableSet<PageElement> =
        mutableSetOf()
    fun addPageElement(pageElement: PageElement) =
        pageElements.add(pageElement)
    fun maxHeight(): Int =
        pageElements.map { it.height }.max()
    fun tallerThan(first: PageElement, second: PageElement) =
        first.height > second.height
}

interface ImperialMutableList<T> {
    val size: Int
    fun isEmpty(): Boolean = size <= 0
    fun add(element: T)
    operator fun get(index: Int): T
    fun addAll(other: ImperialMutableList<T>) {
        for (index in 0..<other.size) {
            add(other[index])
        }
    }
}
```

`addAll()` invoked on a `ResizingArrayList` Efficiency

Two problems:

- Every call to `add` will check to see whether a resize is needed
- Multiple resizes could occur if `other` is large

Better `addAll()` for `ResizingArrayList`

```
override fun addAll(other: ImperialMutableList<T>) {
        val newSize = size + other.size
        if (newSize > elements.size) {
                val newCapacity = max(newSize, 2 * elements.size)
                elements = elements.copyOf(newCapacity)
        }
        for (i in 0..<other.size) {
                elements[size + i] = other.get(i)
        }
        size = newSize
}
```

We override the default method to give a specialised implementation.
Do a single resize if necessary.
Add the new elements, without the need for resize checks.

Can't stop classes implementing an interface from not providing an implementation for certain functionalities.

Default properties in interfaces

The `PageElement` interface specifies properties `width` and `height`
We can add an `area` property that defaults to `width * height`

```
interface PageElement {
    val width: Int
    val height: Int
    val area: Int
        get() = width * height
}
```

```
class ScaledPageElement(
    val target: PageElement,
    val scaleFactor: Int,
) : PageElement {
    override val width: Int
        get() = target.width * scaleFactor
    override val height: Int
        get() = target.height * scaleFactor
}
```

Overriding a default property

What if a default property involves an expensive computation each time it is accessed?

```
interface SomeInterface {
    // Other properties and methods omitted
    val someQuantity: Int
        get() = ... // Complex calculation
}
```

In some implementing classes, it could be beneficial to compute the property once and reuse that result:

```
class SomeClass : SomeInterface {
    private var precomputedQuantity: Int? = null
    // A new property to store the pre-computed quantity
    // null if we have not yet computed the quantity, otherwise
stores the value of the quantity
    override val someQuantity: Int
        get() {
            if (precomputedQuantity == null) {
                precomputedQuantity = super.someQuantity
                // super.someQuantity accesses the default
implementation of get() for this property
            }
            return precomputedQuantity!!
        }
}
```

```


interface Widgit {
    val width: Int
}

class BasicWidgit(override val width: Int) : Widgit

class WidgitWithWidth20 : Widgit {
    override val width: Int = 20
}

class SideBySideWidgits(
    val first: Widgit,
    val second: Widgit,
) : Widgit {
    override val width
        get() = first.width + second.width
    // Need this so that if first.width and second.width change,
then width will also change
}

class MutableWidgit(override var width: Int) : Widgit

fun main() {
    val basic: Widgit = BasicWidgit(10)
    val stupid: Widgit = WidgitWithWidth20()
    val composite: Widgit = SideBySideWidgits(basic, stupid)
    val mutable: Widgit = MutableWidgit(20)
    val anotherComposite: Widgit = SideBySideWidgits(composite,
mutable)
    println(composite.width)
    println(anotherComposite.width)
    mutable.width = 256
    println(anotherComposite.width)
}
```