



Interfaces

Alastair F. Donaldson

Aims of this lecture

- Introduce the notion of an interface
- Use an interface to specify the functionality common to the two kinds of lists we have implemented
- See how this allows us to use these lists interchangeably
- Introduce default methods

Terminology: 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, think about the *service* the class is intended to provide to its *clients*

- The service is provided via the **public** properties and methods of the class
- Everything else – all internal details – are not part of the service and should be **private**

Imagine a world ...

... where Kotlin collections do not exist, and people are really going to use our list classes

In reality, you should use a language's standard collections unless there is a very good reason to "roll your own"

But: as computer scientists, you should know how they work!

Client code that uses `SinglyLinkedList`

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)
```

There is nothing special about the letter **T** – using any other letter, or a longer name – works fine

Client code that uses SinglyLinkedList

Exercise: why is this very inefficient?

We will see later how to avoid this inefficiency

```
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))  
    }  
    for (index in 0..<second.size>) {  
        result.add(second.get(index))  
    }  
    return result  
}
```

Nicer if we could write

`for (element in second)`

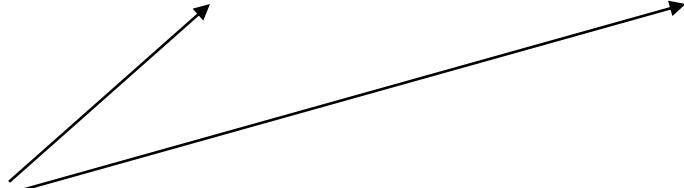
Nicer if we could write

`second[index]`

Later, we will achieve this via **iterators** and **operator overloading**

What if we want to apply these functions to array-based lists?

```
val someList = ResizingArrayList<String>();  
val someOtherList = ResizingArrayList<String>();  
...  
if (doesEitherContain(someList, someOtherList, "Cat")) {  
    ...  
}
```



Type mismatch.

Required: SinglyLinkedList<TypeVariable (T)>

Found: ResizingArrayList<String>

What if we want to apply these functions to array-based lists?

```
val someList = ResizingArrayList<String>();  
val someOtherList = ResizingArrayList<String>();  
...  
val bigList = combine(someList, someOtherList)
```

Type mismatch.



Required: SinglyLinkedList<TypeVariable (T)>

Found: ResizingArrayList<String>

Solution? Overload the client functions

Existing
functions:

```
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))  
    }  
    ...  
}
```

New
overloads:

```
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

What if we want to mix different kinds of list?

```
val someList = ResizingArrayList<String>();  
val someOtherList = SinglyLinkedList<String>();  
...  
if (doesEitherContain(someList, someOtherList, "Cat")) {  
    ...  
}
```

Type error: neither overload is applicable



Solution? More overloads ...

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

```
fun <T> combine(  
    first: ResizingArrayList<T>,  
    second: SinglyLinkedList<T>,  
) : ResizingArrayList<T> {  
    ...  
}
```

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

```
fun <T> combine(  
    first: SinglyLinkedList<T>,  
    second: ResizingArrayList<T>,  
) : ResizingArrayList<T> {  
    ...  
}
```

This is getting a bit silly!

The right 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  
}
```

I use this name to avoid confusion with Kotlin's `MutableList`

None of the methods have bodies

They simply describe the services that a mutable list promises should provide

These are called **abstract methods**

The 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  
}
```

This means: to be an **ImperialMutableList**, a class must provide read access to a **size** property

val means “at least read access must be provided”

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 solution: a mutable list interface

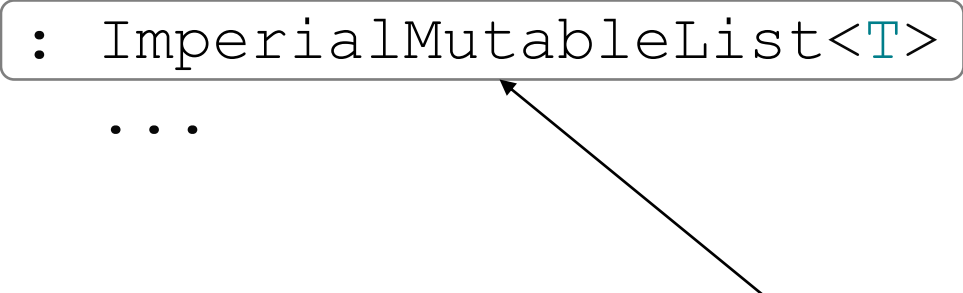
```
interface ImperialMutableList<T> {  
    val size: Int  
  
    fun get(index: Int): T  
  
    fun add(element: T)  
  
    fun add(index: Int, element: T)  
  
    fun clear()  
  
    fun contains(element: T): Boolean  
  
    fun removeAt(index: Int): T  
  
    fun remove(element: T): Boolean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

This 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

Implementing the interface

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



Read this as “implements `ImperialMutableList<T>`”

It is a **promise**: `ResizingArrayList<T>` promises to provide read access to a **size** property, and implementations of all the methods

Implementing the interface

Original class

```
class ResizingArrayList<T>(  
    private val initialCapacity: Int  
) {  
    ...  
  
    var size: Int = 0  
        private set  
  
    private var elements: Array<T?>  
        = clearedArray()  
  
    fun get(index: Int): T = ...  
  
    fun add(element: T) = ...  
  
    fun add(index: Int, element: T) {  
        ...  
    }  
    ...  
}
```

Version of class that implements the interface

```
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) {  
        ...  
    }  
    ...  
}
```


Implementing the interface

Original class

```
class ResizingArrayList<T>(  
    private val initialCapacity: Int  
) {  
    ...  
  
    var size: Int = 0  
        private set  
  
    private var elements: Array<T?>  
        = clearedArray()  
  
    fun get(index: Int): T = ...  
  
    fun add(element: T) = ...  
  
    fun add(index: Int, element: T) {  
        ...  
    }  
    ...  
}
```

Version of class that implements the interface

```
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) {  
        ...  
    }  
    ...  
}
```

Understanding the extra syntax

Version of class that implements the interface

```
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) {  
        ...  
    }  
    ...  
}
```

Declares that the class intends to implement the interface

Asserts that this fulfils the promise of read access to a **size** property

Private write access is also provided – that's fine

Asserts that these methods intentionally implement the required methods of the interface

The `override` keyword

When you write a class to implement an interface, you **must** annotate each of your implementations of the interface methods with `override`

Strange use of the term “override” – the interface does not describe any actual behaviour, so what are we overriding?

We will soon see that interfaces can also provide **default** method implementations whose behaviour can be changed

What if we do not implement all interface methods?

```
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) {  
        ...  
    }  
    ...  
}
```

Rule: A class that implements an interface must provide implementations for all abstract methods

Missing methods lead to compilation errors

Implementation of `get` has been omitted

Error: Class 'ResizingArrayList' does not implement fun `get(index: Int): T`

Do we have to override properties?

```
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) {  
        ...  
    }  
    ...  
}
```

Rule: A class that implements an interface must override all abstract properties

Missing properties lead to compilation errors

size property has been omitted


Error: Class 'ResizingArrayList' does not implement size

Exercise: adapt your `SinglyLinkedList` class to that it implements our new interface

Client code can now use the *interface*

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

The function works with any
objects of classes that implement
ImperialMutableList<T>



```
val someList = ResizingArrayList<String>();  
val someOtherList = ResizingArrayList<String>();  
...  
if (doesEitherContain(someList, someOtherList, "Cat")) {  
    ...  
}
```


Fine: **someList** and **someOtherList** both have type **ImperialMutableList<T>**

Why? Because class **ResizingArrayList<T>** implements **ImperialMutableList<T>**
interface

Client code can now use the *interface*

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

The function works with any
objects of classes that implement
ImperialMutableList<T>



```
val someList = SinglyLinkedListList<String>();  
val someOtherList = SinglyLinkedListList<String>();  
...  
if (doesEitherContain(someList, someOtherList, "Cat")) {  
    ...  
}
```

Fine: **someList** and **someOtherList** both have type **ImperialMutableList<T>**

Why? Because class **SinglyLinkedList<T>** implements **ImperialMutableList<T>**
interface

Client code can now use the *interface*

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

The actual types of the two objects might be different when the function is invoked

```
val someList = ResizingArrayList<String>();  
val someOtherList = SinglyLinkedListList<String>();  
...  
if (doesEitherContain(someList, someOtherList, "Cat")) {  
    ...  
}
```

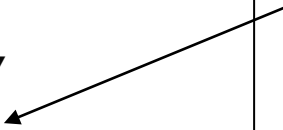
Fine: **someList** and **someOtherList** both have type **ImperialMutableList<T>**

Why? **SinglyLinkedList<T>** implements **ImperialMutableList<T>**

ResizingArrayList<T> implements **ImperialMutableList<T>**

Does this work?

```
fun <T> combine(  
    first: ImperialMutableList<T>,  
    second: ImperialMutableList<T>,  
) : ImperialMutableList<T> {  
    val result = ImperialMutableList<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  
}
```



No: `ImperialMutableList<T>` is an *interface*

We cannot directly create an instance of an interface type

We must instead create an instance of some class that implements the interface type

Error: Interface `ImperialMutableList` does not have constructors

Does this work?

```
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  
}
```

Yes: **SinglyLinkedList**<T> is a *class*,
so we can construct an instance

The function needs to return
an **ImperialMutableList**<T>

result has type **SinglyLinkedList**<T>

This is fine, because a
SinglyLinkedList<T> is an
ImperialMutableList<T>

Interfaces: another motivating example

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

Let's start simple:

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

TextBox class

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

Image class

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

DocumentManager has text boxes and images

```
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)  
}
```

Here comes trouble...

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

Identical computation for text boxes and images

Duplication is **bad**:

- Makes software difficult to **maintain**

```
class DocumentManager {  
  
    // Declarations as before, plus:  
  
    fun maxHeight(): Int =  
        max(  
            textBoxes.map { it.height }.max(),  
            images.map { it.height }.max(),  
        )  
}
```


More trouble: let's have Menu page elements

```
class Menu(  
    val width: Int,  
    val height: Int,  
) {  
    private val options: MutableList<String> =  
        mutableListOf()  
  
    fun addOption(option: String) {  
        options.add(option)  
    }  
  
    fun hasOption(candidateOption: String) =  
        options.contains(candidateOption)  
}
```

DocumentManager with text boxes, images and menus

```
class DocumentManager {  
    private val textBoxes: MutableSet<TextBox> = mutableSetOf()  
    private val images: MutableSet<Image> = mutableSetOf()  
    private val menus: MutableSet<Menu> = mutableSetOf()  
  
    fun addTextBox(textBox: TextBox) = textBoxes.add(textBox)  
    fun addImage(image: Image) = images.add(image)  
    fun addMenu(menu: Menu) = menus.add(menu)  
  
    // Continued on next slide
```

DocumentManager with text boxes, images and menus

// Continued from previous slide

```
fun maxHeight(): Int =
```

```
    listOf(
```

```
        textBoxes.map { it.height }.max(),
```

```
        images.map { it.height }.max(),
```

```
        menus.map { it.height }.max(),
```

```
    ).max()
```

```
}
```

} Lots of duplication!

Problems with this?

- A lot of **duplicate code** in `DocumentManager`
- `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

Even worse...

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!
```

Terrible! 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

What do we really want?

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**: they all have widths and heights

We would like 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: <code>PageElement</code> interface

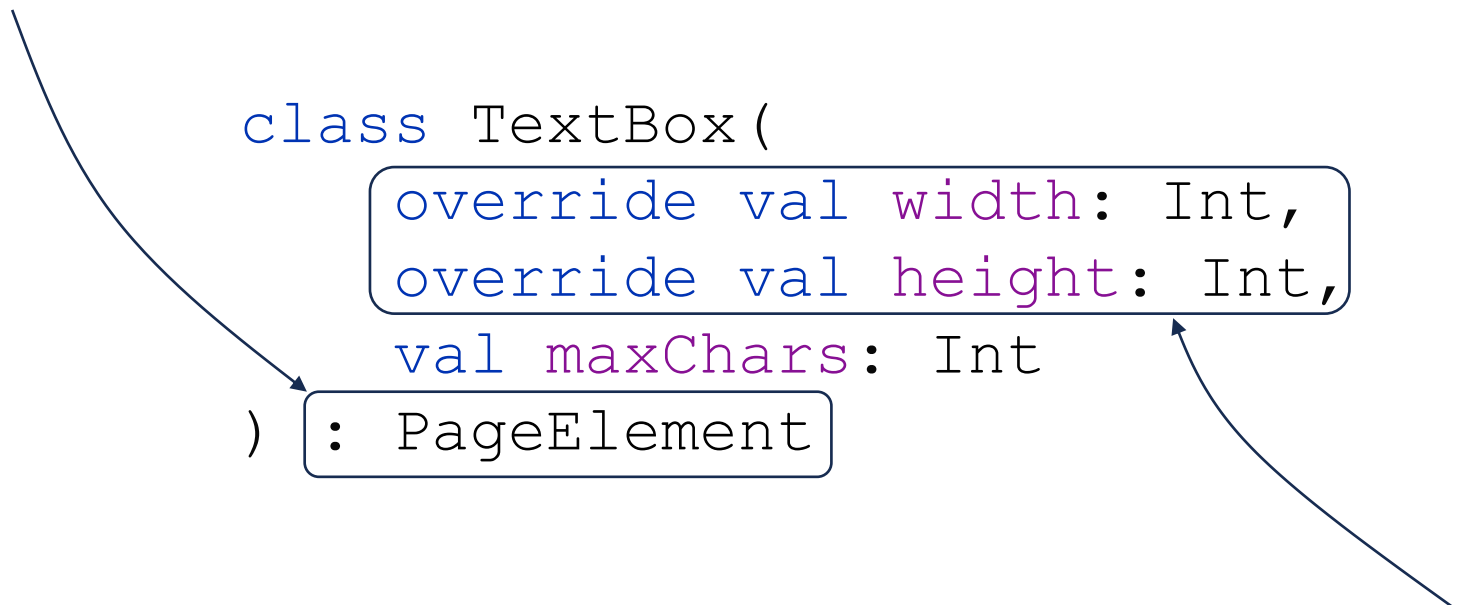
PageElement interface

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

TextBox class implements PageElement interface

TextBox implements PageElement: it promises to provide width and height properties

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

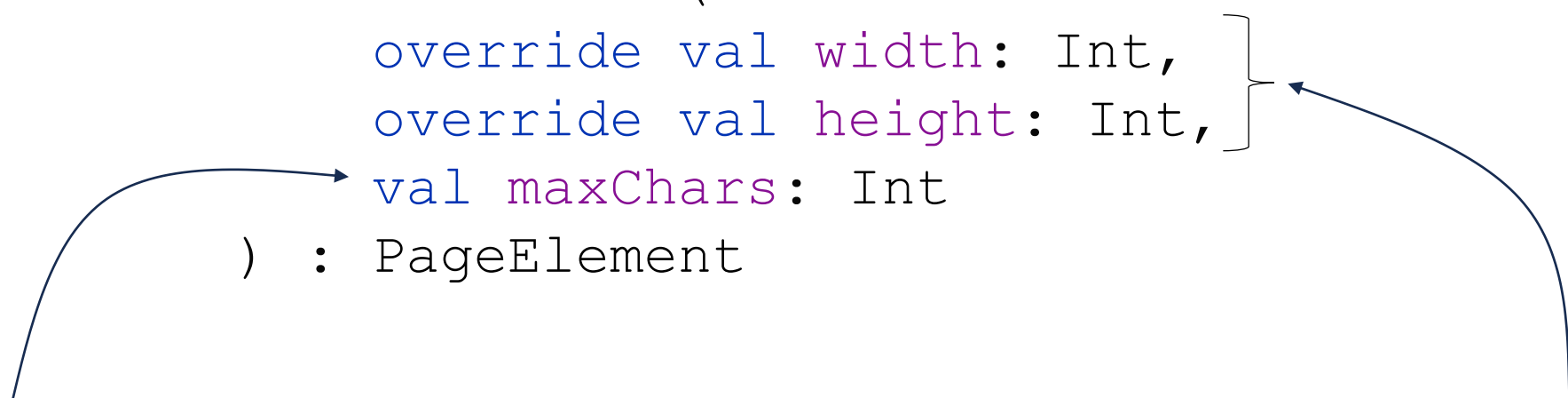


The promised properties are provided

TextBox class implements PageElement interface

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

Specific to TextBox



Required by PageElement

Image class implements PageElement interface

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

Specific to Image



The diagram consists of two curved arrows. One arrow starts from the text 'Specific to Image' and points to the 'val filename: String,' line in the code. The other arrow starts from the text 'Required by PageElement' and points to a bracket that groups the 'override val width: Int,' and 'override val height: Int,' lines in the code.

Required by PageElement

Menu class implements PageElement interface

```
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)  
}
```

} Required by PageElement

} Specific to Menu

DocumentManager with PageElements

Much simpler!

One set of PageElements (before: separate sets for TextBoxes, Images, Menus)

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

One method for adding PageElements
(before: addTextBox, addImage, addMenu)

DocumentManager with PageElements

Much simpler!

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

We can map **once** to get the heights of all page elements

it will refer to a mixture of TextBoxes, Images and Menus

They are guaranteed to have heights because they implement PageElement

DocumentManager with PageElements

A more explicit way to write `maxHeight`:

```
fun maxHeight(): Int =  
    pageElements.map {  
        item: PageElement -> item.height  
    }.max()
```

DocumentManager with PageElements

A more explicit way to write `maxHeight`:

```
pageElements has type MutableSet<PageElement>

fun maxHeight(): Int =
    pageElements.map {
        item: PageElement -> item.height
    }.max()
```

We can map a `PageElement` \rightarrow `Int` function over `pageElements`

This lambda is our mapper function

`map` yields a `Set<Int>` and we use `max` to compute its maximum

DocumentManager with PageElements

```
fun maxHeight(): Int =  
    pageElements.map {  
        item: PageElement -> item.height  
    }.max()
```

The page elements the lambda will process may have a variety of different types (TextBox, Image, Menu, other page elements)

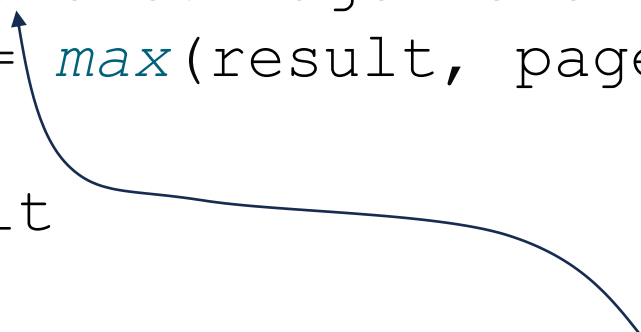
The PageElement interface allows us to treat them all **uniformly**

This is called **polymorphism**

DocumentManager with PageElements

An imperative implementation of `maxHeight`:

```
fun maxHeight(): Int {  
    var result = 0  
    for (pageElement: PageElement in pageElements) {  
        result = max(result, pageElement.height)  
    }  
    return result  
}
```



On each loop iteration, `pageElement` may refer to a `TextBox`, `Image` or `Menu`, depending on the contents of `pageElements`

Again, being able to handle all these page elements uniformly is an example of **polymorphism**

DocumentManager with PageElements

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

Huge win:

- **Before:** we had 9 overloaded versions of `tallerThan`
- **Before:** N different page elements led to N^2 versions of `tallerThan`, one for each pair of types
- **Now:** this **single** method suffices, **no matter how many** kinds of `PageElements` we have

Is it easy to add further page elements?

If we add another page element, say `RadioButton`, what changes do we have to make to `DocumentManager`?

NONE!

Advantages of interfaces so far

Helps us **manage complexity** by treating objects of various classes **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
- Obvious way to implement this for any list: check `size > 0`
- We can add this as a **default method**

isEmpty() as a default method

```
interface ImperialMutableList<T> {  
    val size: Int  
    fun get(index: Int): T  
    fun add(element: T)  
    // Other methods as before  
    fun remove(element: T): Boolean  
    fun isEmpty(): Boolean = size <= 0  
}
```

This is a **default method** because it has an implementation



We can now ask whether any ImperialMutableList is empty

No changes needed to ResizingArrayList or SinglyLinkedList

A default method for adding one list to another

```
interface ImperialMutableList<T> {  
    val size: Int  
    fun get(index: Int): T  
    fun add(element: T)  
    // Other methods as before  
    fun isEmpty(): Boolean = size <= 0  
  
    fun addAll(other: ImperialMutableList<T>) {  
        for (index in 0..<other.size) {  
            add(other.get(index))  
        }  
    }  
}
```

This is a straightforward way to
add one list to another

It works, but for a specific list
there might be a better way

Will the default `addAll()` be efficient when invoked on a `ResizingArrayList`?

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

Two problems:

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

Efficient addAll () for ResizingArrayList

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

```
    // Properties and methods as before
```

```
    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
```

```
    }
```

```
}
```

Do a **single** resize
if necessary

We **override** the
default method to
give a specialised
implementation

Add the new elements, without the need for resize checks

Exercise: add elements of another list at a given index

- Write a default method for `ImperialMutableList<T>` with the following signature:

```
fun addAll(index: Int, other: ImperialMutableList<T>)
```

- The method should add all the elements of other right after the given index
- Is your implementation likely to be efficient for `ResizingArrayLists`? For `SinglyLinkedLists`?
- If not, can you override the method in these classes to provide a more efficient implementation?

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  
}
```

Default property



Consider a scaled page element: represents an existing page element in a larger form

```
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  
}
```

What does the default `area` property compute?

```
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  
}
```

`area` is `width * height`, which expands to:

`(target.width * scaleFactor) * (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  
  
}
```

Overriding a default property

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  
    override val someQuantity: Int  
    get () {  
        if (precomputedQuantity == null) {  
            precomputedQuantity = super.someQuantity  
        }  
        return precomputedQuantity!!  
    }  
}
```

null if we have not yet computed the quantity,
otherwise stores the value of the quantity

Overriding the default property someQuantity

A new property
to store the
pre-computed
quantity

super.someQuantity accesses the default
implementation of get () for this property