



Inheritance and abstract classes – part 2

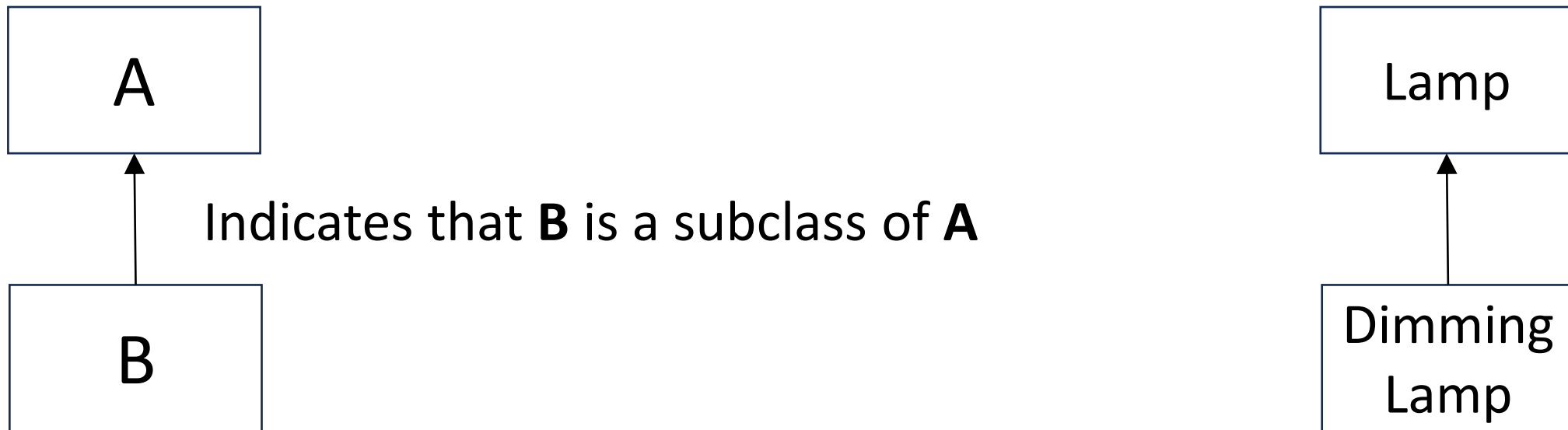
Alastair F. Donaldson

Aims of this lecture

- Introduce class diagrams
- Study subtyping for classes and interfaces
- Introduce the concept of apparent vs. actual types
- Study late binding, also known as dynamic dispatch
- Discuss benefits of programming against an interface
- Study dynamic type testing, smart casts and downcasts
- Look at overriding properties
- Study the use of strategies as an alternative to inheritance

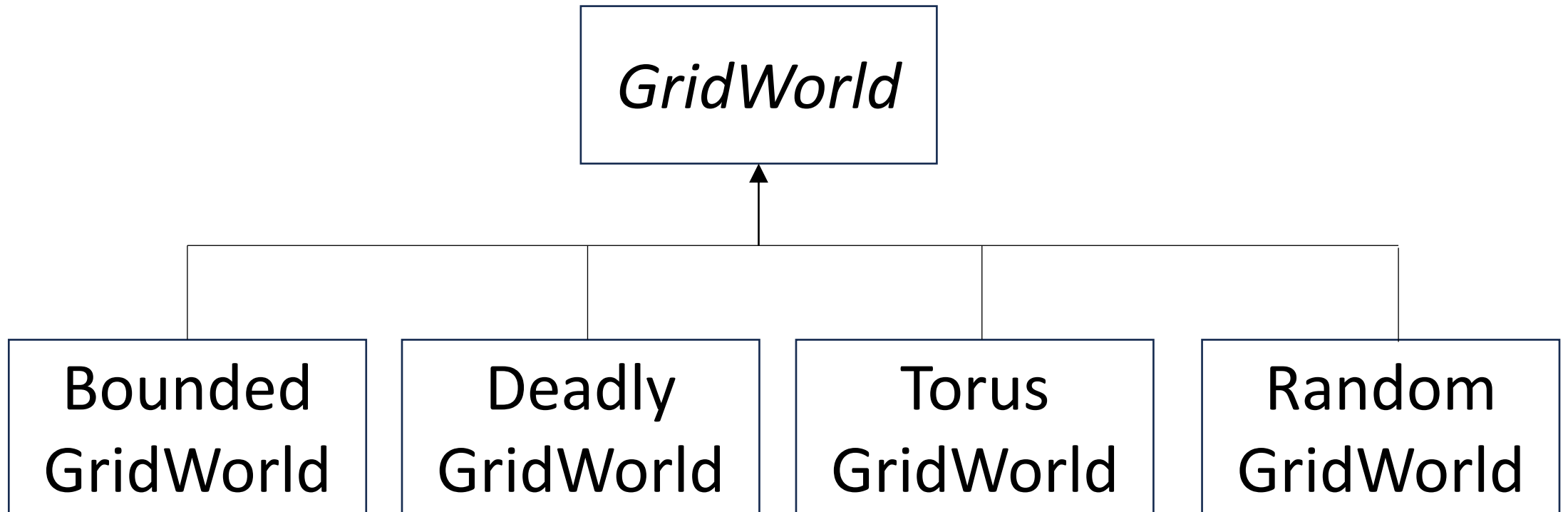
Class diagrams

Useful notation to sketch class relationships during design, or in documentation



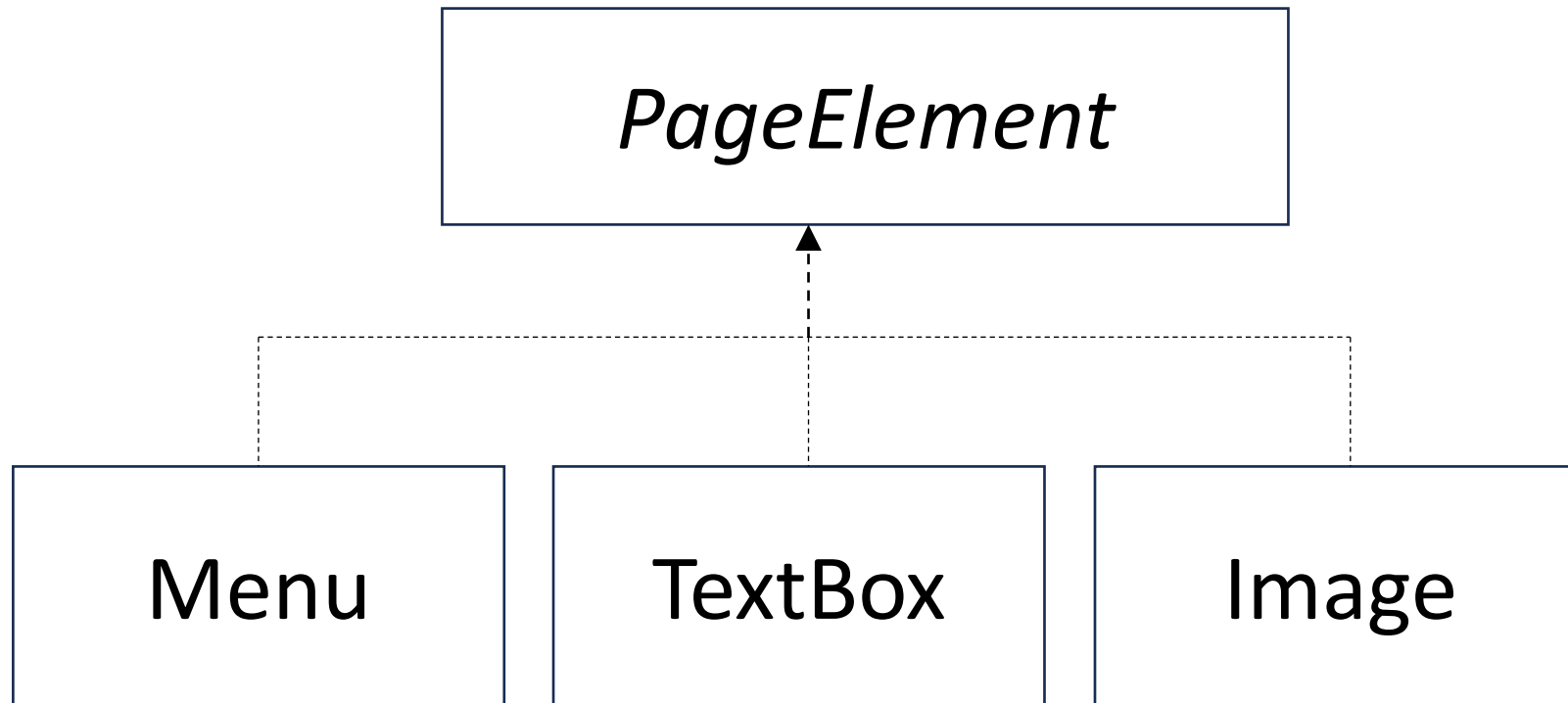
Class diagrams

Sometimes italics are used for abstract class or interface



Class diagrams

Sometimes dashed line used to indicate implementation of interface



Extending interfaces

Interface **B** can **extend** an interface **A** by adding additional method signatures

To implement **B**, a class must provide all the methods required by **A**, plus any additional methods required by **B**

The existing interface **A** specifies a **service**

The interface **B** that extends **A** specifies a **larger service**

MutableIterator extends Iterator

```
interface Iterator<T> {  
    fun next(): T  
    fun hasNext(): Boolean  
}
```

```
interface MutableIterator<T> : Iterator<T> {  
    fun remove()  
}
```

A class implementing `MutableIterator` must provide the service that `Iterator` offers, plus the extra `remove` method

Subtyping

Properties and methods

Let **T** and **U** be class / interface types

U is a **subtype** of **T** if **U** offers at least the **service** that **T** offers

If **U** is a subtype of **T** then **T** is a **supertype** of **U**

Subtyping

Suppose that client code has been designed to work with an object of type **T**

The client depends on the service provided by **T**

Therefore, the client can work with an object of type **U** whenever **U** is a subtype of **T**

Subtyping: Lamp and DimmingLamp

Lamp provides a simple service:

- `pressSwitch`
- `toString`

DimmingLamp provides this service, plus more (up, down)

The DimmingLamp subclass is therefore a **subtype** of Lamp – it provides at least the service that Lamp provides

Subtyping rules: reflexivity and transitivity

Reflexivity: for any type **T**, **T** is a subtype of **T**

Transitivity: for types **T**, **U** and **V**:

- if **V** is a subtype of **U**
- and **U** is a subtype of **T**
- then **V** is a subtype of **T**

Subtyping rules: upper and lower bounds

Let **T** be a class or interface

- **T** is a subtype of `Any`
- `Nothing` is a subtype of **T**

`Nothing` is a type that has no values – it can be used as a return type to indicate that a method is guaranteed to throw an exception

Throwing an exception is not the same as returning a value

Subtyping rules: extending and implementing

Let **T** be a class or interface, and let **U** be a class or interface

- If **U** extends **T** then **U** is a subtype of **T**
- If **U** implements **T** then **U** is a subtype of **T**

In the “implements” case, **U** must be a class and **T** an interface



Subtyping rules: nullability

Let **T** be a class or interface

- **T** is a subtype of **T**?

A weird type: Nothing?

Nothing has no values, so `Nothing?` has one possible value: **null**

```
fun handleString(s: String) {  
    ...  
}
```

```
fun main() {  
    val x = null  
    handleString(x)  
}
```

The inferred type for `x` is `Nothing?`

Compile error: Type mismatch.

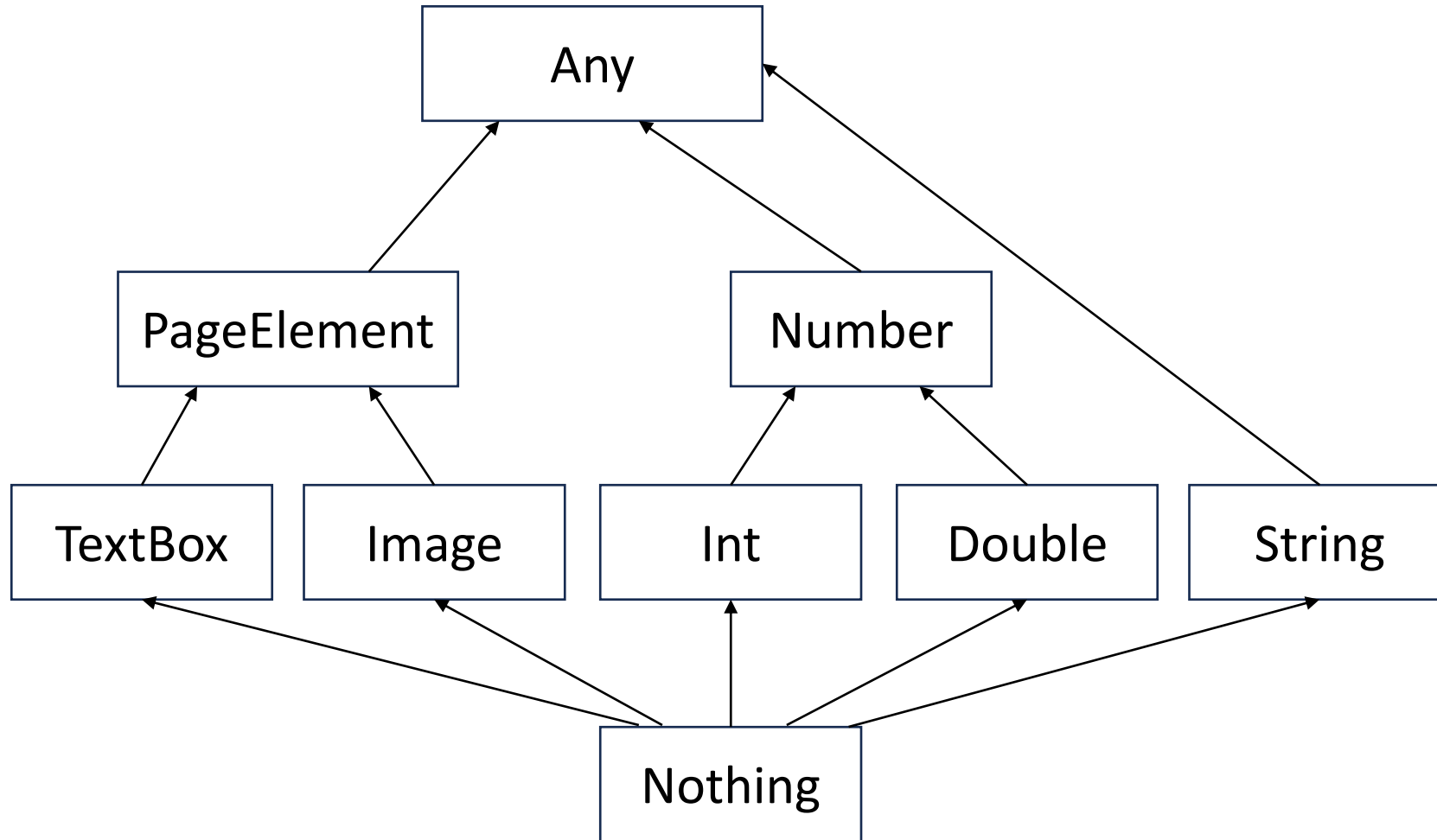
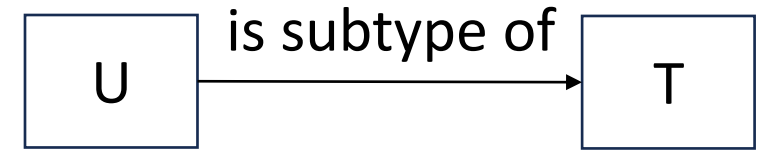
Required:

String

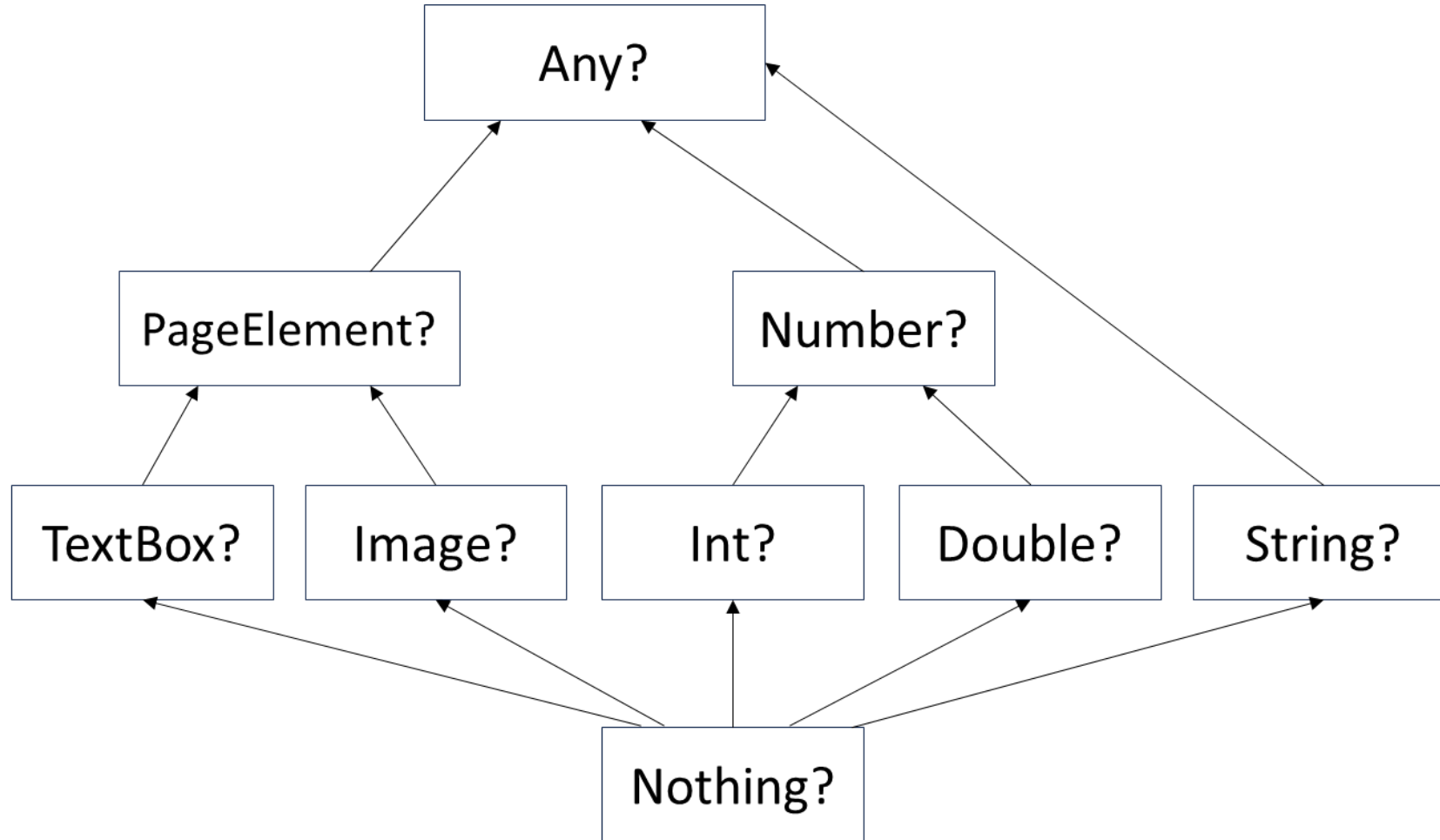
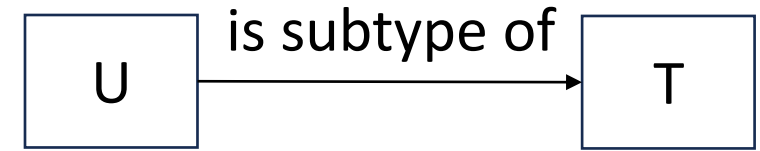
Found:

`Nothing?`

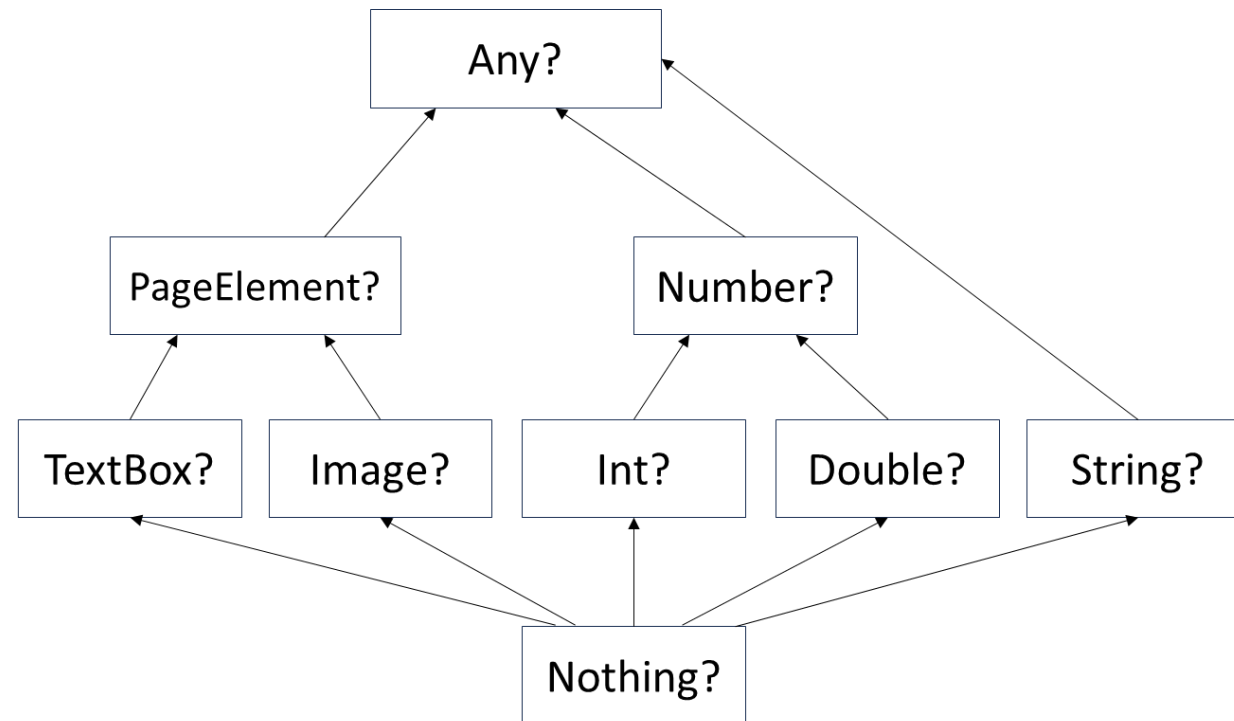
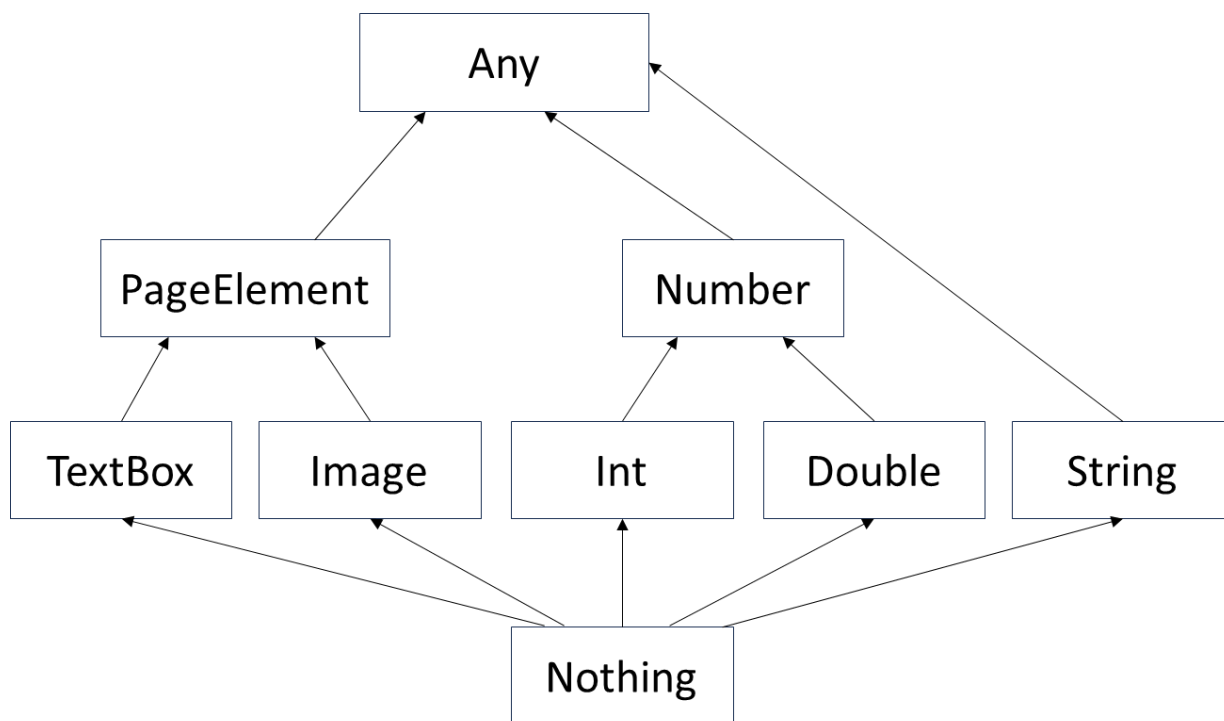
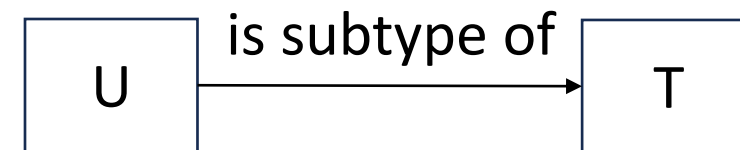
Type hierarchy



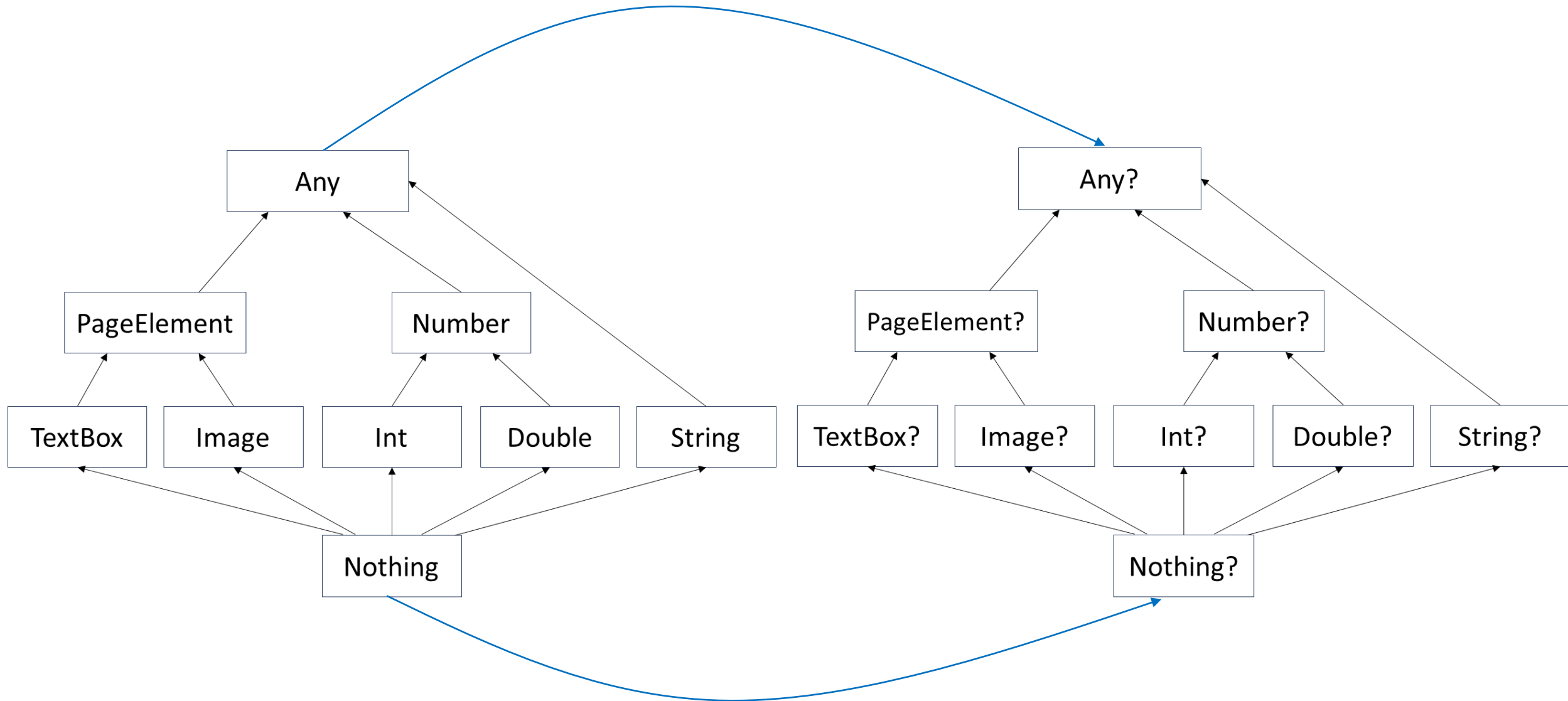
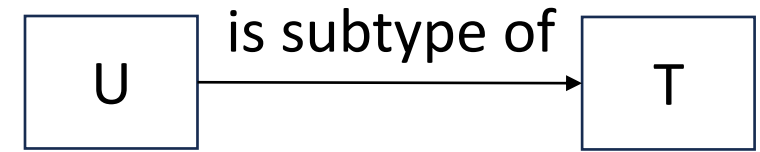
Type hierarchy



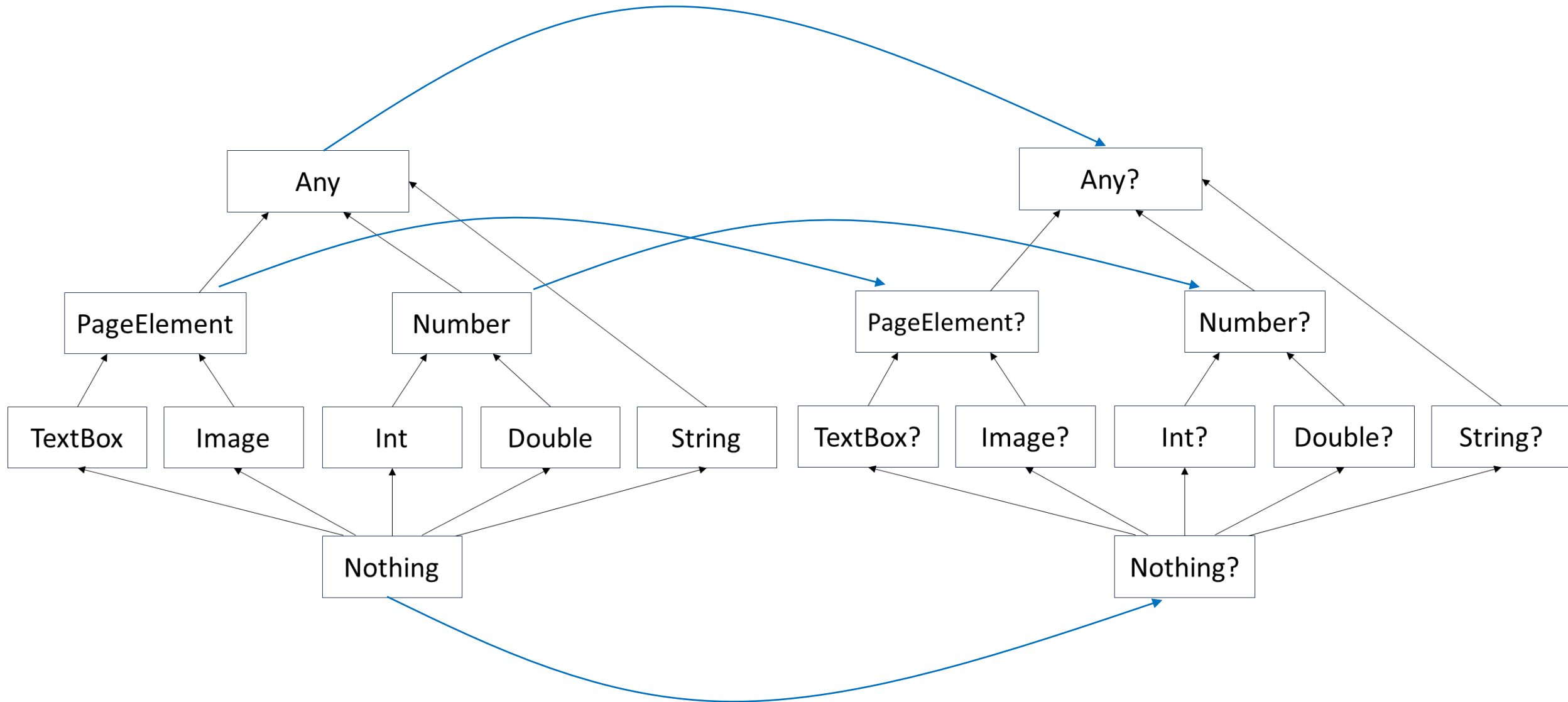
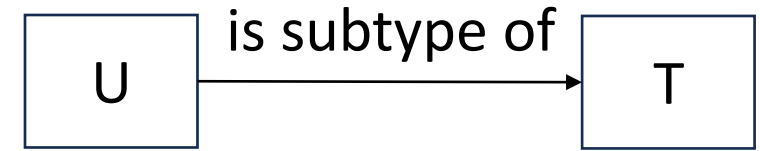
Type hierarchy



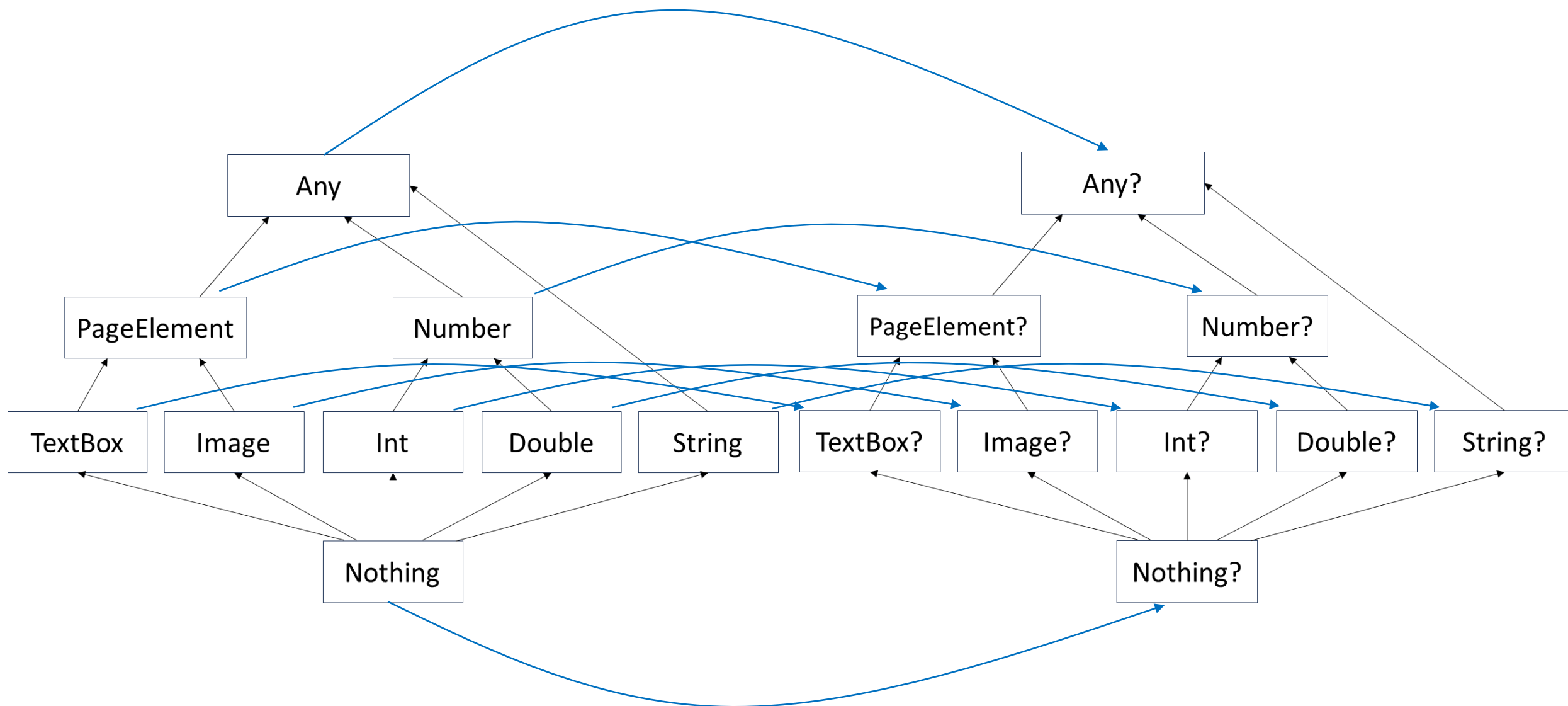
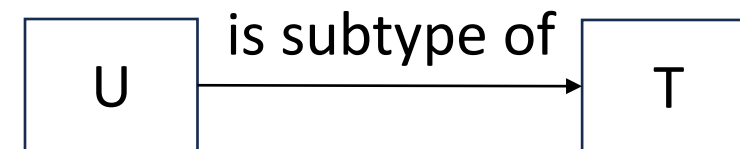
Type hierarchy



Type hierarchy



Type hierarchy



Subtyping: notation

Common to write `<:` to mean “is a subtype of”, and `:>` to mean “is a supertype of”

Some example subtyping relationships:

- `String <: Any`
- `Any <: Any?`
- `TextBox <: PageElement <: PageElement? <: Any?`
- `Nothing <: String`
- `Nothing <: Nothing?`

Passing subtypes as parameters

Rule: if a function requires a parameter of type **T**, it is OK to pass an argument of type **U** $\text{U} <: \text{T}$

```
fun foo(n: Number) {  
    ...  
}
```

```
fun main() {  
    val i: Int = 42  
    foo(i) ← OK: Int <: Number  
}
```


Receiving a return value into a supertype

Rule: if a function returns a parameter of type **U**, it is OK to receive this into a variable or property of type **T :> U**

```
fun produceString() : String = ...
```

```
fun main() {  
    var maybeString: String? = null  
    maybeString = produceString()  
}
```


OK: String? :> String



Subtyping in action: assigning to a property or variable

Rule: in an assignment statement of the form $x = e$, the type of e must be a subtype of the type of x

```
class SmartHome {  
    private var lamp: Lamp = Lamp(isOn = true)  
  
    fun upgradeLamp () {  
        lamp = DimmingLamp(isOn = true)  
    }  
}
```



OK: `DimmingLamp <: Lamp`

Late binding

```
fun manipulateLamp(lamp: Lamp) {  
    ...  
    lamp.pressSwitch()  
    ...  
}
```

Which method should be called?

pressSwitch from Lamp?

pressSwitch from DimmingLamp?

pressSwitch from some other Lamp subclass?

The method that should be called depends on the type of the object to which `lamp` refers **when the program runs**

Late binding

Determining which method to call based on the runtime type of an object is called *late binding*

Binding: the link between the method name in the source code and the code of the method that should be executed

Late: the binding is not determined when the program is compiled – it is determined **later**, when the program is running

Apparent vs. actual types

Apparent type of an object reference: the type provided in the declaration (or the type inferred by the compiler)

Actual type of an object reference: the type of the object to which the reference refers when the program is running

Apparent type never changes: the types of declarations are fixed

Actual type can change if the reference gets re-assigned

Actual type is always a **subtype** of **apparent type**

Apparent vs. actual types: example

```
fun main(args: Array<String>) {  
    val lamp1: Lamp = Lamp(isOn = true)  
}
```

Apparent type of `lamp1` is `Lamp` – provided in declaration

Actual type of `lamp1` is `Lamp`, due to `Lamp` constructor

Apparent vs. actual types: example

```
fun main(args: Array<String>) {  
    val lamp1: Lamp = Lamp(isOn = true)  
    var lamp2 = Lamp(isOn = false)
```

Apparent type of lamp2 is Lamp – inferred by compiler

Actual type of lamp2 is Lamp, due to Lamp constructor

Apparent vs. actual types: example

```
fun main(args: Array<String>) {  
    val lamp1: Lamp = Lamp(isOn = true)  
    var lamp2 = Lamp(isOn = false)  
    val lamp3 = DimmingLamp(isOn = true)  
}
```

Apparent type of lamp3 is DimmingLamp – inferred by compiler

Actual type of lamp3 is DimmingLamp, due to DimmingLamp constructor

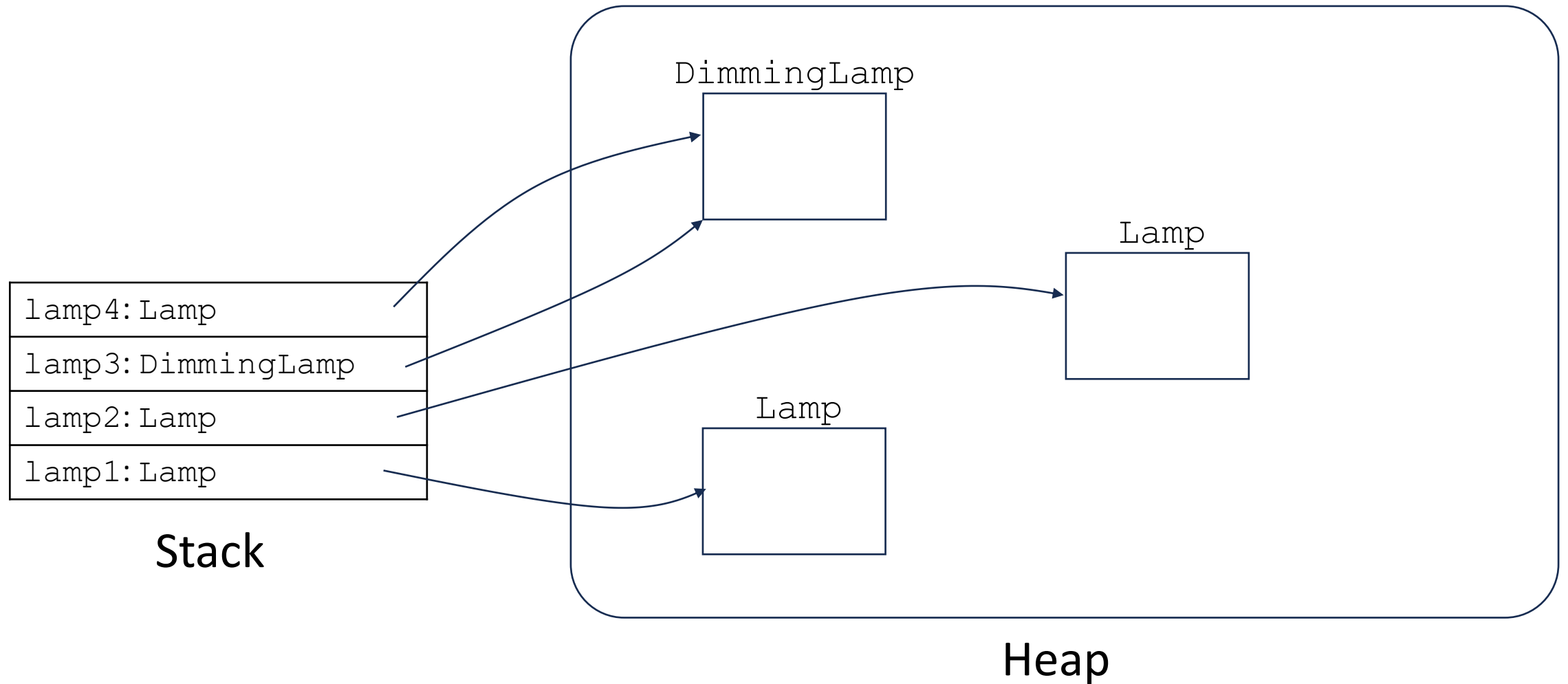
Apparent vs. actual types: example

```
fun main(args: Array<String>) {  
    val lamp1: Lamp = Lamp(isOn = true)  
    var lamp2 = Lamp(isOn = false)  
    val lamp3 = DimmingLamp(isOn = true)  
    val lamp4: Lamp = lamp3  
    // What do stack and heap look like here?  
}
```

Apparent type of lamp4 is Lamp – provided in declaration

Actual type of lamp4 is DimmingLamp: it refers to the same DimmingLamp as lamp3

Apparent vs. actual types: example

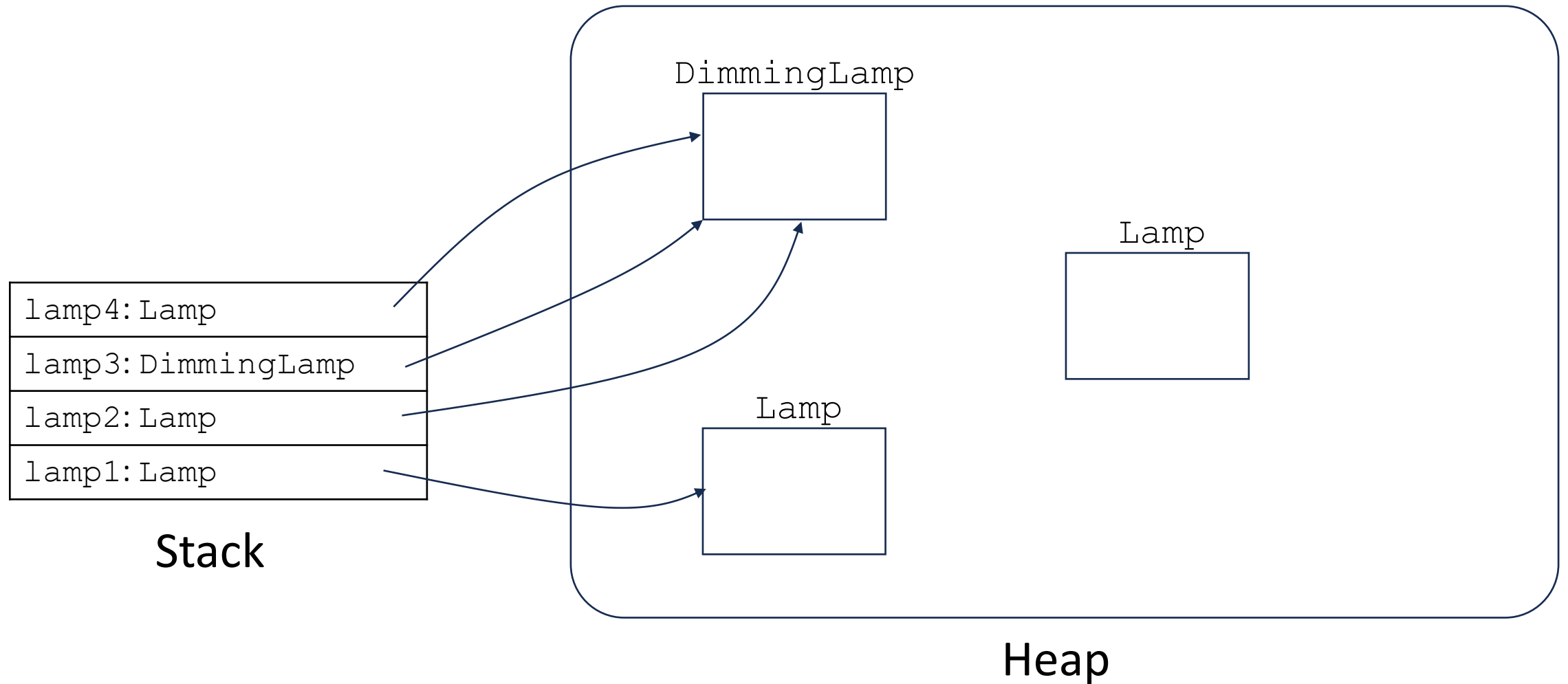


Apparent vs. actual types: example

```
fun main(args: Array<String>) {  
    val lamp1: Lamp = Lamp(isOn = true)  
    var lamp2 = Lamp(isOn = false)  
    val lamp3 = DimmingLamp(isOn = true)  
    val lamp4: Lamp = lamp3  
    // What do stack and heap look like here?  
    lamp2 = lamp3  
    // What do stack and heap look like here?  
}
```

Actual type of lamp2 has changed to DimmingLamp:
lamp2 has been reassigned to refer to a different object

Apparent vs. actual types: example



Apparent types and properties / methods

Properties and methods are available based on **apparent type**

Apparent type: Lamp

Actual type: DimmingLamp

```
val lamp: Lamp = DimmingLamp(isOn = true)
lamp.pressSwitch()
lamp.down()
```

The diagram shows three arrows: one from 'Apparent type: Lamp' to the 'Lamp' in the variable declaration; one from 'Actual type: DimmingLamp' to the 'DimmingLamp' in the constructor; and one from the 'OK' text to the 'pressSwitch()' method call.

OK: apparent type (Lamp) provides method `pressSwitch`

The diagram shows an arrow pointing from the 'Compile error' text to the 'down()' method call in the code block above.

Compile error: apparent type (Lamp) does not provide method `down`

Apparent types and properties / methods

```
fun main() {  
    var lamp = Lamp(isOn = true)  
    println(lamp)  
    lamp.pressSwitch()  
    println(lamp)  
    // lamp.down() // Compile error  
    val dimmingLamp = DimmingLamp(isOn = false)  
    println(dimmingLamp)  
    dimmingLamp.pressSwitch()  
    dimmingLamp.down()  
    println(dimmingLamp)  
    lamp = DimmingLamp(isOn = false)  
    println(lamp)  
    lamp.pressSwitch()  
    println(lamp)  
    // lamp.down() // Compile error  
}
```

Output:

LIGHT

(darkness)

(darkness)

LIGHT: *****

(darkness)

LIGHT: *****

Late binding again

```
fun manipulateLamp(lamp: Lamp) {  
    ...  
    lamp.pressSwitch()  
    ...  
}
```

Which method should be called?

pressSwitch from Lamp?

pressSwitch from DimmingLamp?

pressSwitch from some other Lamp subclass?

At runtime, the **actual type** of `lamp` is used to determine which version of the `pressSwitch` method to call

Late binding again

Late binding is also called **dynamic method dispatch** (or just dynamic dispatch)

Method dispatch: the process of determining which code should execute in response to a method call

Dynamic method dispatch: the code to be executed is determined at runtime – i.e., based on **dynamic** information

Static method dispatch: the code to be executed is already determined when the program is compiled – based purely on **static** information

Static == “at compile time”

Dynamic == “at runtime”

Dynamic vs. static dispatch

Static dispatch is slightly more efficient than dynamic dispatch:

- **Statically dispatched call:** program counter is set to the start address of the method
- **Dynamically dispatched call:** start address of the required method is determined based on object's actual type; then program counter is set to this address

Static method dispatch in Kotlin

A method call can be statically dispatched if the method is not abstract, and there is no possibility of the method being overridden

This is applies to:

- Top level functions
- Final methods (the default)
- Private methods

Careful: extension methods use **static** dispatch based on **apparent type**

```
open class A {  
    open fun sayHello()  
        = println("Hello from A")  
}  
  
class B : A() {  
    override fun sayHello()  
        = println("Hello from B")  
}  
  
fun A.sayGoodbye()  
    = println("Goodbye from A")  
  
fun B.sayGoodbye()  
    = println("Goodbye from B")
```

```
fun main() {  
    val myA: A = B()  
    val myB: B = B()  
    myA.sayHello()  
    myB.sayHello()  
    myA.sayGoodbye()  
    myB.sayGoodbye()  
}
```

Dynamic dispatch:
actual type is B

Static dispatch:
apparent type is A

Output:

```
Hello from B  
Hello from B  
Goodbye from A  
Goodbye from B
```

Programming against an interface

Best practice: program against **interfaces**

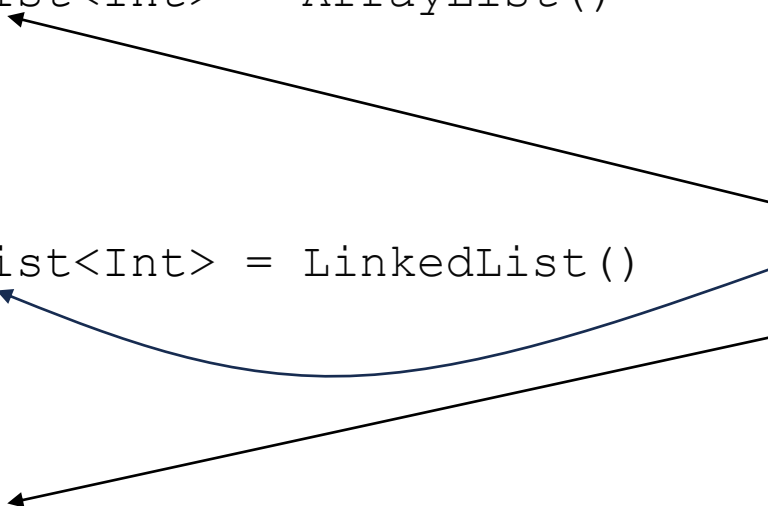
Example: when you need a mutable list property, use `MutableList` as apparent type for the property:

```
class DataStore {  
    private val data: MutableList<Int> = ArrayList()  
    ...  
}
```

```
class DataStore {  
    private val data: MutableList<Int> = LinkedList()  
    ...  
}
```

```
class DataStore {  
    private val data: MutableList<Int> = mutableListOf()  
    ...  
}
```

All fine: the
apparent type is
`MutableList`

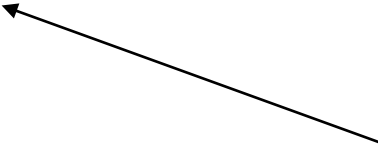


Programming against an interface

Do not use a more specific type as apparent type


```
class DataStore {  
    private val data: ArrayList<Int> = ArrayList()  
    ...  
}
```

Bad: apparent type is `ArrayList`,
an implementing class



```
class DataStore {  
    private val data = LinkedList()  
    ...  
}
```

Bad: apparent type is (inferred to
be) `LinkedList`, an
implementing class




Programming against an interface

Do not use a more specific type as apparent type

Why not?

- If your code only depends on the service provided by the interface, your code is more **flexible**
- By using `MutableList` as apparent type, we cannot write code that accidentally depends on methods only provided by e.g. `ArrayList`
- This makes it trivial to change our code to use a different kind of list:

```
class DataStore {  
    private val data: MutableList<Int> = ArrayList() LinkedList()  
    ...  
}
```



No other changes needed: `data` only offered the `MutableList` service, which `LinkedList` provides

Programming against an interface

What's bad about this?

```
fun filterBadWords(  
    wordList: ArrayList<String>,  
    badWords: Set<String>,  
) : List<String> = wordList.filter { it !in badWords }
```

It has been programmed against a specific kind of list: `ArrayList`

The function is thus limited to filtering `ArrayLists`

Programming against an interface

Better:

```
fun filterBadWords(  
    wordList: List<String>,  
    badWords: Set<String>,  
) : List<String> = wordList.filter { it !in badWords }
```

Now the function is programmed against the `List` interface

The function can filter any kind of `List`

But ... don't implementing classes provide added value?

Example:


- `ArrayList` provides an `ArrayList`-specific method, `trimToSize`
- `trimToSize` resizes the list so that its capacity matches the current list size – gets rid of spare capacity
- This can save memory if list is not expected to grow in the future

We cannot call `trimToSize` if we program strictly against the `MutableList` interface

The interface does not offer `trimToSize` as part of its service

Solution: `is` and smart casts

```
class DataManager {  
    val lotsOfData: MutableList<MutableList<String>>  
        = mutableListOf()  
  
    ...  
  
    fun finaliseData() {  
        for (list in lotsOfData) {  
  
            list.trimToSize()  
  
        }  
    }  
}
```



Compile error: Unresolved reference: trimToSize

Solution: `is` and smart casts

```
class DataManager {  
    val lotsOfData: MutableList<MutableList<String>>  
        = mutableListOf()  
    ...  
    fun finaliseData() {  
        for (list in lotsOfData) {  
            if (list is ArrayList) {  
                list.trimToSize()  
            }  
        }  
    }  
}
```

Check whether **actual type** of `list` is `ArrayList`

Control only reaches here if actual type is `ArrayList`, so `list` can be used with **apparent type** `ArrayList` – this is a **smart cast**

Equivalent code

```
class DataManager {  
    val lotsOfData: MutableList<MutableList<String>>  
        = mutableListOf()
```

...

```
for (list in lotsOfData) {  
    if (list !is ArrayList) {  
        continue
```

```
    }  
    list.trimToSize()
```

```
}
```

```
}
```

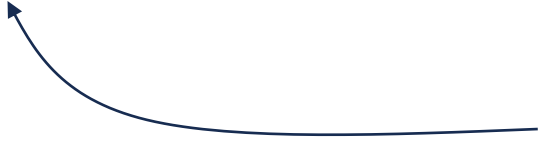
Check whether **actual type** of list is ArrayList

Control only reaches here if actual type is ArrayList, so list can be used with **apparent type** ArrayList – this is a **smart cast**

Limits of smart casts

```
class SmartHome {  
    private var lamp: Lamp = Lamp(isOn = true)  
  
    fun lowerBrightness(notches: Int) {  
        if (lamp is DimmingLamp) {  
            for (i in 1..notches) {  
                lamp.down()  
            }  
        }  
    }  
  
    fun updateLamp(newLamp: Lamp) {  
        lamp = newLamp  
    }  
}
```

Compile error: Smart cast to 'DimmingLamp' is impossible, because 'lamp' is a mutable property that could have been changed by this time



Downcasting, via `as`

If you are **sure** the actual type of an object reference will be some more specific subtype, you can use `as` to downcast the object reference to the more specific type

Expression Type name

 ↓ ↙

e `as` ***T***



Throws a `ClassCastException` if actual type of *e* is not a subtype of *T*

Otherwise: has no effect on *e*, but the expression (*e as T*) has apparent type *T*

This allows access to the *T*-specific services of *e*

Using a downcast

```
class SmartHome {  
    private var lamp: Lamp = Lamp(isOn = true)  
  
    fun lowerBrightness(notches: Int) {  
        if (lamp is DimmingLamp) {  
            for (i in 1..notches) {  
                (lamp as DimmingLamp).down()  
            }  
        }  
    }  
  
    fun updateLamp(newLamp: Lamp) {  
        lamp = newLamp  
    }  
}
```

Downcast: fails with
ClassCastException
if used incorrectly

Apparent type is
DimmingLamp, so
down is available

Exercise

Without changing the code of the previous slide, write a program that will use a `SmartHome` in a way that causes the downcast to **fail**

The fact that the downcast has the potential to fail is why a smart cast is not possible

Kotlin only allows a smart cast when the compiler can be sure a downcast would succeed

Design challenge

A `User` has a set of friends that can change over time

- Part of the service `User` provides: reveal who its set of friends are
- **Not** part of the `User` service: allow clients to modify this set

How should a `User`'s friends be modelled using properties?

Option 1: provide read-only view of mutable data

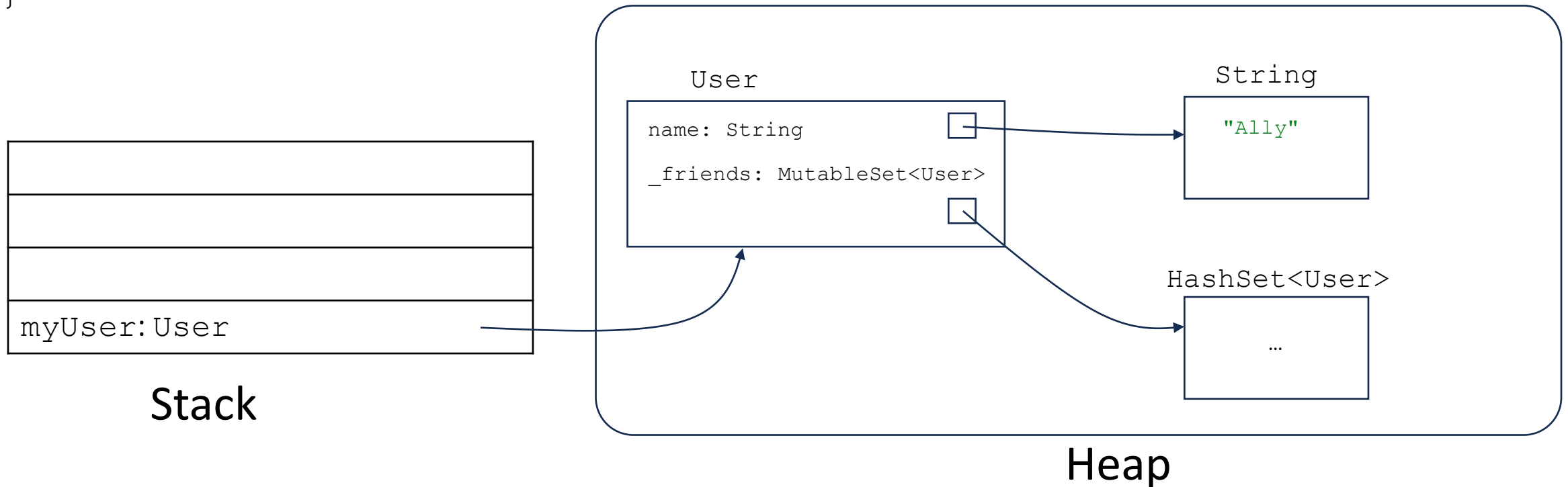
`_friends` equips a User with a mutable set of friends for internal use

```
class User(val name: String) {  
    private val _friends: MutableSet<User> = mutableSetOf()  
  
    val friends: Set<User>  
        get() = _friends  
  
    fun considerFriendRequest(otherUser: User) {  
        ...  
    }  
}
```

`friends` reveals the mutable set of friends to clients, but provides it with apparent type Set – the mutable part of the set's service is hidden

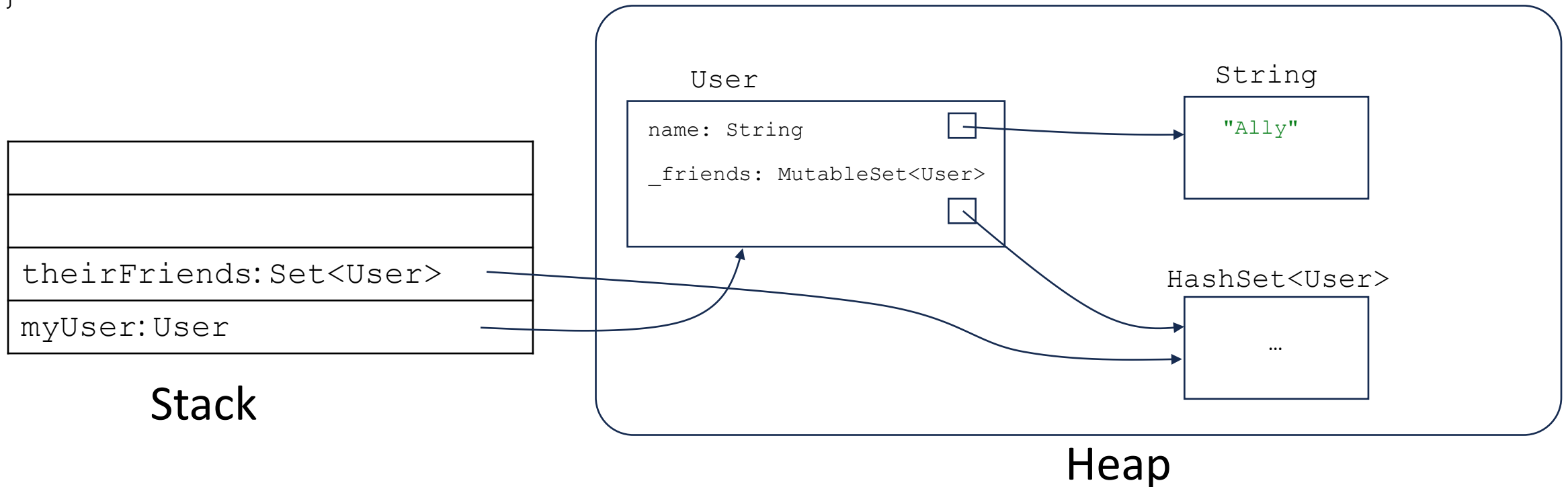
This design does **not** duplicate the `_friends` set

```
fun main() {  
    val myUser = User("Ally")  
    ...  
}
```



This design does **not** duplicate the `_friends` set

```
fun main() {  
    val myUser = User("Ally")  
    ...  
    val theirFriends = myUser.friends  
}
```



Exercise

With this design, show that it is still possible for a client to add a user to the set of friends, despite the `friends` property having apparent type `Set<User>`

Option 2: return a copy

`_friends` equips a User with
a mutable set of friends for
internal use

```
class User(val name: String) {
```

```
    private val _friends: MutableSet<User> = mutableSetOf()
```

```
    val friends: Set<User>
        get() = _friends.toSet()
```

```
    fun considerFriendRequest(otherUser: User) {
```

```
        ...
```

```
    }
```

```
}
```

`friends` yields a new set that contains all the
Users in `_friends`

Exercise

Is it still possible to mutate the set returned by `friends` despite it being a `Set`, not a `MutableSet`?

What impact does this have on the user whose `friends` were revealed via the `friends` property?

Option 3: return the set via a protective wrapper

`_friends` equips a User with a mutable set of friends for internal use

```
class User(val name: String) {  
    private val _friends: MutableSet<User> = mutableSetOf()  
  
    val friends: Set<User>  
        get() = Collections.unmodifiableSet(_friends)  
  
    fun considerFriendRequest(otherUser: User) {  
        ...  
    }  
}
```

`friends` yields an object that **wraps** `_friends` and rejects any mutating calls

Exercise

Is it still possible to mutate the set returned by `friends` despite it being a `Set`, not a `MutableSet`?

Exercise

Consider all three design options

Suppose that client code is well behaved – it does not attempt to add friends to a `User` via a “back door”

Do the three approaches make any difference to the way a client perceives the friends of a `User`?

Overriding properties

With inheritance, we have focused on subclasses overriding **methods**

However, properties can be overridden too – let's see an example

A `Cell` class – wraps a value of some type

Allows `Cell` to
have **subclasses**



Allows the `value`
property to be **overridden**



```
open class Cell<T>(open var value: T) {  
    override fun toString(): String {  
        return "$value"  
    }  
}
```

BackedUpCell subclass – allows previous cell values to be restored

```
class BackedUpCell<T>(value: T) : Cell<T>(value) {  
    private val backups: MutableList<T> = mutableListOf()  
  
    override var value: T  
        get() = super.value  
        set(value) {  
            backups.add(0, super.value)  
            super.value = value  
        }  
  
    fun restore() {  
        if (!backups.isEmpty()) {  
            super.value = backups.removeAt(0)  
        }  
    }  
}
```

Overriding the `value` property

Exercise: why is it important to use `super` here?

Alternatives to inheritance

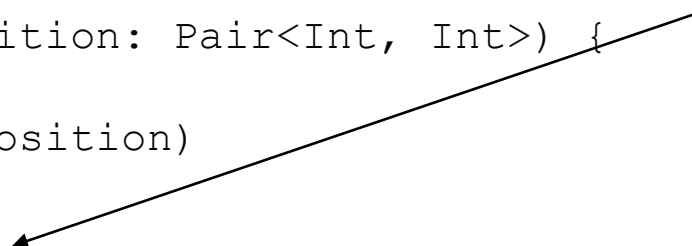
Inheritance can be useful, but is not always the best approach

Let us look at an alternative way to model `GridWorlds` that avoids inheritance

Remember the GridWorld abstract class?

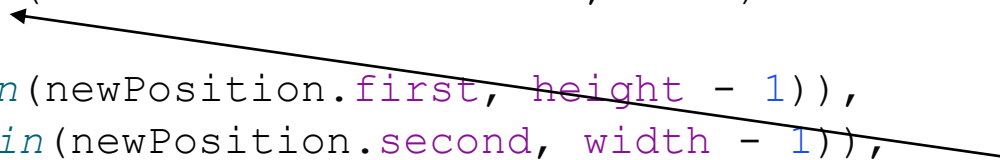
```
abstract class GridWorld(...) {  
    ...  
    private fun updatePosition(newPosition: Pair<Int, Int>) {  
        ...  
        position = handleOverrun(newPosition)  
    }  
    protected abstract fun handleOverrun(newPosition: Pair<Int, Int>): Pair<Int, Int>  
    ...  
}
```

Details of how to
handle overruns left
abstract: subclasses
fill them in



```
class BoundedGridWorld(...) : GridWorld(...) {  
    override fun handleOverrun(newPosition: Pair<Int, Int>): Pair<Int, Int> =  
        Pair(  
            first = max(0, min(newPosition.first, height - 1)),  
            second = max(0, min(newPosition.second, width - 1)),  
        )  
}
```

One example of a
subclass “filling in
the blanks”



Alternative design

Have just one `GridWorld` class – not abstract, not open

Equip this class with a `handleOverrun` **property**: a function that determines how overruns should be handled

A function property that parameterises a class is called a **strategy**

Let's see this in action

GridWorld using a strategy

Strategy for handling
overruns

```
class GridWorld(  
    private val width: Int,  
    private val height: Int,  
    private val handleOverrun: (Pair<Int, Int>) -> Pair<Int, Int>,  
) {  
    private val grid: Array<Array<Terrain>> = randomTerrain()  
    private var position: Pair<Int, Int> = randomPosition()  
  
    fun up() = updatePosition(position.copy(second = position.second + 1))  
    fun down() = updatePosition(position.copy(second = position.second - 1))  
  
    // left() and right() - similar  
  
    private fun updatePosition(newPosition: Pair<Int, Int>) {  
        if (newPosition.first in 0..<width &&  
            newPosition.second in 0..<height) {  
            position = newPosition  
            return  
        }  
        position = handleOverrun(newPosition)  
    }  
    ...  
}
```

Invoke the strategy method
to handle a potential overrun

Creating specific GridWorlds

Aside: functions used to create object instances are called **factories**

Easy: pass a suitable strategy

```
fun createBoundedGridWorld(width: Int, height: Int): GridWorld =
    GridWorld(width, height) {
        Pair(
            first = max(0, min(it.first, height - 1)),
            second = max(0, min(it.second, width - 1)),
        )
    }

fun createDeadlyGridWorld(width: Int, height: Int): GridWorld =
    GridWorld(width, height) {
        throw DeadPlayerException("Fell off world!")
    }
```

Exercise

What are the advantages and disadvantages of the inheritance-based approach vs. the strategy approach to modelling `GridWorlds`?