Inheritance and Abstract Classes – Part 3

Example to showcase inheritance and dynamic dispatch

```
package contrivedexample
open class X {
    open fun foo() {
        println("foo of X")
        // This does not have the effect of blocking the subclass
call.
        this@X.bar()
    }
    open fun bar() {
        println("bar of X")
    }
}
class Y : X() {
    override fun bar() {
        println("bar of Y")
    }
}
fun main() {
    val myY: X = Y()
    myY.foo()
}
```

```
package contrivedexample
class VisibilityExample(private val p: Int) {
    fun addProperties(other: VisibilityExample): Int {
        // Notice that "other.p" is visible even though "p" is
private.
        // This is because "private" means "class private", not
"object private"
        return this.p + other.p
    }
}
```

```
package contrivedexample
open class A {
    open fun foo() {
        println("foo of A")
    }
    open fun bar() {
        println("bar of A")
    }
    open fun baz() {
        println("baz of A")
        bar()
    }
}
open class B : A() {
    override fun foo() {
        baz()
        super.foo()
        println("foo in B")
    }
}
class C : B() {
    override fun baz() {
        println("baz in C")
        super.baz()
        super.foo()
    }
    override fun bar() {
        println("bar in C")
    }
}
class D : B() {
    override fun foo() {
        println("foo in D - calls to superclass are blocked.")
    }
}
fun main() {
// Uncomment these lines and step through executin in the debugger
to understand
// which methods actually get called.
//    val myD: A = D()
//    myD.foo()
//    myD.bar()
//    myD.baz()
```

```
// This leads to infinite mutual recursion.
//    val myC: A = C()
//    myC.foo()
//    myC.bar()
//    myC.baz()
    val myD: A = D()
    myD.foo()
    myD.bar()
    myD.baz()
}
```

Inheritance vs. composition

It is common to want to augment the behaviour of a class C

Often you have a choice between:

- **Inheritance**: Extending C, and providing extra behaviour in a subclass
- **Composition**: Creating a new class that implements the same interface as C, delegates behaviour to an instance of C, and provides “added value” before or after delegation

Unmodifiable list

```
package collections
class UnmodifiableImperialMutableList<T>(
    val targetList: ImperialMutableList<T>,
) : ImperialMutableList<T> by targetList {
    override fun clear(): Nothing {
        throw UnsupportedOperationException()
    }
    override fun removeAt(index: Int): Nothing {
        throw UnsupportedOperationException()
    }
    override fun set(index: Int, element: T): Nothing {
        throw UnsupportedOperationException()
    }
    override fun remove(element: T): Nothing {
        throw UnsupportedOperationException()
    }
    override fun add(index: Int, element: T): Nothing {
```

```
        throw UnsupportedOperationException()
    }
    override fun add(element: T): Nothing {
        throw UnsupportedOperationException()
    }
    override fun addAll(other: ImperialMutableList<T>): Nothing {
        throw UnsupportedOperationException()
    }
    override fun addAll(index: Int, other: ImperialMutableList<T>):
Nothing {
        throw UnsupportedOperationException()
    }
    override fun toString(): String {
        return targetList.toString()
    }
}

fun main() {
    // This example illustrates the use of our version of an
unmodifiable wrapper.
    val someList = SinglyLinkedList<Int>()
    val unmodifiableCopy =
UnmodifiableImperialMutableList(someList)
    println("Lists are initially empty:")
    println(someList)
    println(unmodifiableCopy)
    someList.add(1)
    someList.add(2)
    someList.add(3)
    println("After adding some elements to someList")
    println(someList)
    println(unmodifiableCopy)
    println("Attempting to add an element to unmodifiableCopy")
    unmodifiableCopy.add(4)
}

class UnmodifiableSinglyLinkedList<T>(vararg values: T) :
SinglyLinkedList<T>() {
    init {
        for (value in values) {
            super.add(value)
        }
    }
    // Block all of the mutator methods
```

```
    override fun add(element: T): Nothing {
        throw UnsupportedOperationException()
    }
    override fun add(index: Int, element: T): Nothing {
        throw UnsupportedOperationException()
    }
    override fun remove(element: T): Nothing {
        throw UnsupportedOperationException()
    }
    override fun removeAt(index: Int): Nothing {
        throw UnsupportedOperationException()
    }
    override fun clear(): Nothing {
        throw UnsupportedOperationException()
    }
    override fun set(index: Int, element: T): Nothing {
        throw UnsupportedOperationException()
    }
    override fun addAll(other: ImperialMutableList<T>): Nothing {
        throw UnsupportedOperationException()
    }
    override fun addAll(index: Int, other: ImperialMutableList<T>):
Nothing {
        throw UnsupportedOperationException()
    }
}

fun main() {
    val myList: ImperialMutableList<Int> =
        UnmodifiableSinglyLinkedList(1, 2, 3, 4)
    for (v in myList) {
        println(v)
    }
    myList.add(42)
}
```

Provide an “unmodifiable” version of the mutable list classes that we have created

- Try it using **inheritance**
- Try it using **composition**

What are the pros and cons?

Logging list

```
class LoggingImperialMutableList<T>(
    private val targetList: ImperialMutableList<T>,
) : ImperialMutableList<T> by targetList {
    var numberOfExternalCalls: Int = 0
        private set
    override fun add(element: T) {
        targetList.add(element)
        numberOfExternalCalls++
    }
    override fun add(index: Int, element: T) {
        targetList.add(index, element)
        numberOfExternalCalls++
    }
    override fun toString(): String {
        return targetList.toString()
    }
}

fun main() {
    val myLoggingList = LoggingImperialMutableList<Int>
(SinglyLinkedList())
    myLoggingList.add(1)
    myLoggingList.add(1)
    myLoggingList.add(1)
    myLoggingList.add(1)
    myLoggingList.add(1)
    println(myLoggingList)
    println("Number of external calls to add:
${myLoggingList.numberOfExternalCalls}")
}

// A sublcass of ResizingArray list that logs external calls to
"add"
class LoggingResizingArrayList<T> : ResizingArrayList<T>() {
    var numberOfExternalCallsToAdd: Int = 0
        private set
    private var numberOfInternalCalls: Int = 0
    override fun add(element: T) {
        // Add this logic is horrible and is there to prevent
double
        // increments of "numberOfExternalCallsToAdd".
```

```
        // It is a sign that inheritance is not the right tool
here.
        numberOfInternalCalls++
        super.add(element)
        numberOfInternalCalls--
        if (numberOfInternalCalls == 0) {
            numberOfExternalCallsToAdd++
        }
    }
    override fun add(index: Int, element: T) {
        numberOfInternalCalls++
        super.add(index, element)
        numberOfInternalCalls--
        if (numberOfInternalCalls == 0) {
            numberOfExternalCallsToAdd++
        }
    }
}

fun main() {
    val myLoggingList = LoggingResizingArrayList<Int>()
    myLoggingList.add(1)
    myLoggingList.add(1)
    myLoggingList.add(1)
    myLoggingList.add(1)
    myLoggingList.add(1)
    println(myLoggingList)
    println("Number of external calls to add:
${myLoggingList.numberOfExternalCallsToAdd}")
}
```

Provide functionality to log the number of times certain methods of a list class are called by clients

- Try using **inheritance**
- Try using **composition**

What are the pros and cons?

Unmodifiable logging list

Coding demo
28/02/2024

How easy is it to have an unmodifiable logging list if we use:

- Inheritance
- Composition

Use inheritance sparingly

Complex inheritance hierarchies can make code hard to understand and maintain
“B extends A” implies that every B *is* an A

- Does this relationship between A and B really hold?
- Do you expect it to always hold?
- What if it also makes sense to say: every B is a C?
- Having committed to “B extends A”, you cannot also say “B extends C”
- “B extends C extends A” might not make sense

Favour composition over inheritance where possible

Abstract classes vs. interfaces: do we need both?

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

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

import kotlin.math.max
// It makes sense for this class to directly implement the
PageElement interface
// because it will have *computed* width and height properties. If
we made
// PageElement an abstract class and equipped it with width and
height
// properties that used backing fields, then TabularPageElement
would need to
// *hide* these properties by overriding them with its own
versions. The backing
// fields would still be there, but unused and set with some
default value. This
// would not be a good design.
class TabularPageElement(
    private val pageElements: Array<Array<PageElement>>,
) : PageElement {
    override val width: Int
        get() =
            pageElements.map { row ->
                row.map(PageElement::width)
                    .reduce(Int::plus)
            }.max()
    override val height: Int
        get() {
            if (pageElements.isEmpty()) {
                return 0
            }
            var result: Int = 0
            // For each column
            for (i in 0..<pageElements[0].size) {
                // Compute the height of the column and see if this
is bigger than the tallest column seen so far.
```

```
                result = max(
                    result,
                    pageElements.map { row -> row[i].height
}.sum(),
                )
            }
            return result
        }
}
```

Remember the `PageElement` interface, with Image, Menu and TextBox subclasses?

- Could we have made `PageElement` an abstract class?
- What if we then want a `TabularPageElement`?

Abstract classes vs. interfaces: do we need both?

```
abstract class AbstractImperialMutableList<T> :
ImperialMutableList<T> {
    final override var size: Int = 0
        protected set
    protected fun checkIndexInBounds(index: Int, inclusive: Boolean
= false): Unit =
        if (index !in 0..<(if (inclusive) size + 1 else size)) {
            throw IndexOutOfBoundsException()
        } else {
            Unit
        }
}
interface ImperialMutableList<T> {
    val size: Int
    operator 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
    operator fun set(index: Int, element: T): T
```

```
    operator fun iterator(): Iterator<T>
    fun isEmpty(): Boolean = size <= 0
    fun addAll(other: ImperialMutableList<T>) {
        for (element in other) {
            add(element)
        }
    }
    fun addAll(index: Int, other: ImperialMutableList<T>) {
        for (index in 0..<other.size) {
            add(other.get(index))
        }
    }
}
```

What common state and behaviour can we abstract from our mutable list classes?
Would it be worth introducing an abstract class to capture it?
Could we then dispense with the interface?

Sealed interfaces and classes

```
import kotlin.math.exp
sealed interface Expr {
    class Add(val lhs: Expr, val rhs: Expr) : Expr
    class Mul(val lhs: Expr, val rhs: Expr) : Expr
    class Literal(val value: Int) : Expr
    class Factorial(val target: Expr) : Expr
}
fun evaluate(expr: Expr): Int =
    when (expr) {
        is Expr.Add -> evaluate(expr.lhs) + evaluate(expr.rhs)
        is Expr.Factorial ->
computeFactorial(evaluate(expr.target))
        is Expr.Literal -> expr.value
        is Expr.Mul -> evaluate(expr.lhs) * evaluate(expr.rhs)
    }
fun computeFactorial(n: Int): Int =
    if (n < 0) {
        throw IllegalArgumentException("Attempt to compute
factorial of a negative number")
    } else if (n == 0) {
        1
    } else {
```

```
        n * computeFactorial(n - 1)
    }
fun countLiterals(expr: Expr): Int =
    when(expr) {
        is Expr.Add -> countLiterals(expr.lhs) +
countLiterals(expr.rhs)
        is Expr.Factorial -> countLiterals(expr.target)
        is Expr.Literal -> 1
        is Expr.Mul -> countLiterals(expr.lhs) +
countLiterals(expr.rhs)
    }
```

Let's write an interface and classes for a simple language of expressions, that supports:

- Integer literals
- Addition
- Multiplication
- Factorial

Then let's write:

- An expression evaluator
- A "literal counter" – counts the number of literals that appear in an expression

Sealed = Can only be implemented in this package

Implementing multiple interfaces

```
interface StopWatch {
    fun start()
    fun stop()
}
```

```
interface AlarmClock {
    fun setTime(newTime: Time)
    fun setAlarm(wakeTime: Time, toWake: Wakeable)
}
```

```
class SmartPhone : StopWatch, AlarmClock {
    override fun start() { … }
    override fun stop() { … }
    override fun setTime(newTime: Time) { … }
    override fun setAlarm(wakeTime: Time, toWake: Wakeable) { … }
}
```

Satisfies requirements of AlarmClock

Satisfies requirements of StopWatch

Conflict resolution rules

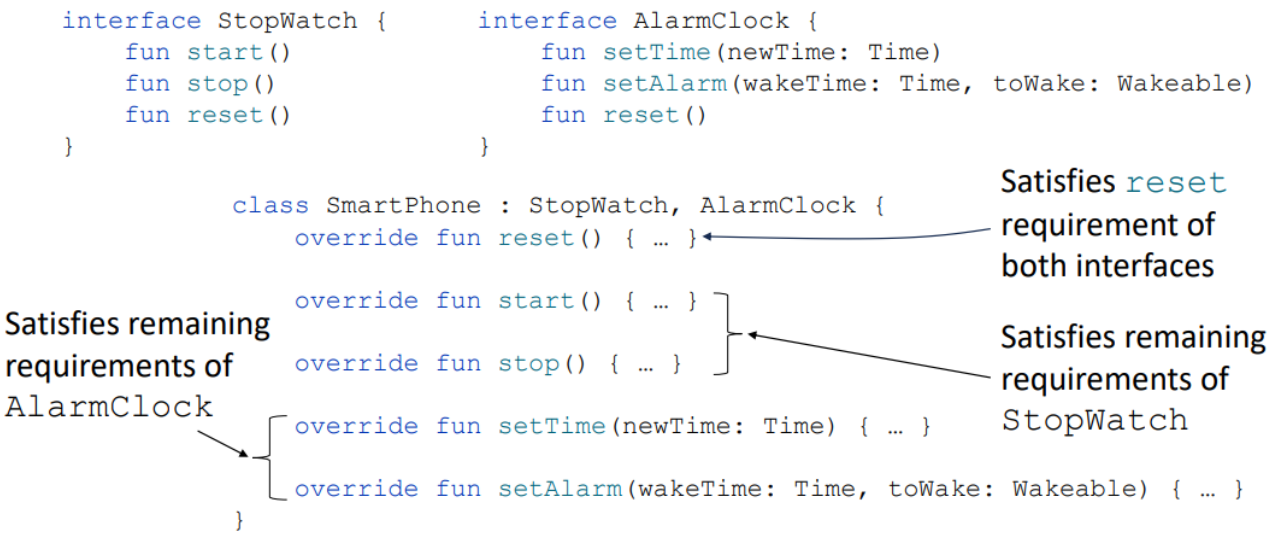
Classes can implement multiple interfaces

- Name conflicts: methods from different interfaces, with same name

Conflict resolution rules:

- Different argument types: OK – this is overloading
- Same argument types, same return type: OK – same method!
  - Direct conflict is OK
- Same argument types, different return type: Compile error
  - In Kotlin, overloading based on return type is not allowed

Example: overlapping interfaces



Direct conflict with default methods?

```
interface StopWatch {
    fun start()
    fun stop()
    fun reset() = stop()
}
```

```
interface AlarmClock {
    fun setTime(newTime: Time)
    fun setAlarm(wakeTime: Time, toWake: Wakeable)
    fun reset() = setTime(MIDNIGHT)
}
```

```
class SmartPhone : StopWatch, AlarmClock {
    override fun start() { … }

    override fun stop() { … }

    override fun setTime(newTime: Time) { … }

    override fun setAlarm(wakeTime: Time, toWake: Wakeable) { … }
}
```

Compiler error: Class 'SmartPhone' must override public open fun reset(): Unit because it inherits multiple interface methods of it

```
interface StopWatch {
    fun start()
    fun stop()
    fun reset() = stop()
}
```

```
interface AlarmClock {
    fun setTime(newTime: Time)
    fun setAlarm(wakeTime: Time, toWake: Wakeable)
    fun reset() = setTime(MIDNIGHT)
}
```

```
class SmartPhone : StopWatch, AlarmClock {

    override fun reset() {
        super<StopWatch>.reset()
        super<AlarmClock>.reset()
    }

    override fun start() { … }

    override fun stop() { … }
    …
```

One possible implementation: invoke both default methods

Kotlin interface hierarchies – list and mutable list, and their iterators

Kotlin `List` interface (slightly simplified)

```
interface List<E> : Collection<E> {

    override val size: Int
    override fun isEmpty(): Boolean
    override fun contains(element: E): Boolean
    override fun iterator(): Iterator<E>

    override fun containsAll(elements: Collection<E>): Boolean

    operator fun get(index: Int): E

    fun indexOf(element: E): Int

    fun lastIndexOf(element: E): Int

    fun listIterator(): ListIterator<E>

    fun listIterator(index: Int): ListIterator<E>

    fun subList(fromIndex: Int, toIndex: Int): List<E>
}
```

Kotlin `MutableList` interface (slightly simplified)

```
interface MutableList<E> : List<E>, MutableCollection<E> {
    override fun add(element: E): Boolean

    override fun remove(element: E): Boolean

    override fun addAll(elements: Collection<E>): Boolean

    fun addAll(index: Int, elements: Collection<E>): Boolean

    override fun removeAll(elements: Collection<E>): Boolean

    override fun retainAll(elements: Collection<E>): Boolean

    override fun clear(): Unit

    operator fun set(index: Int, element: E): E

    fun add(index: Int, element: E): Unit

    fun removeAt(index: Int): E

    override fun listIterator(): MutableListIterator<E>

    override fun listIterator(index: Int): MutableListIterator<E>

    override fun subList(fromIndex: Int, toIndex: Int): MutableList<E>
}
```

Kotlin `Iterator` and `MutableIterator` interfaces

```
interface Iterator<T> {
    operator fun next(): T

    operator fun hasNext(): Boolean
}

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

Kotlin `ListIterator` interface

```
interface ListIterator<T> : Iterator<T> {
    fun hasPrevious(): Boolean

    fun previous(): T

    fun nextIndex(): Int

    fun previousIndex(): Int
}
```

Kotlin `MutableListIterator` interface

```
interface MutableListIterator<T> : ListIterator<T>, MutableIterator<T> {
    fun set(element: T): Unit

    fun add(element: T): Unit
}
```

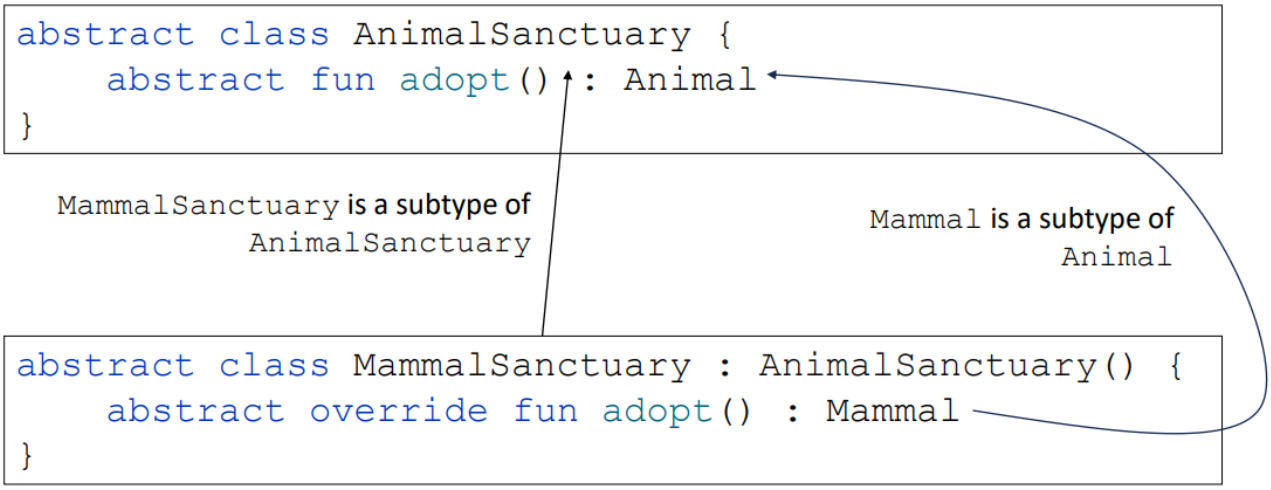
Method overriding: covariant return types

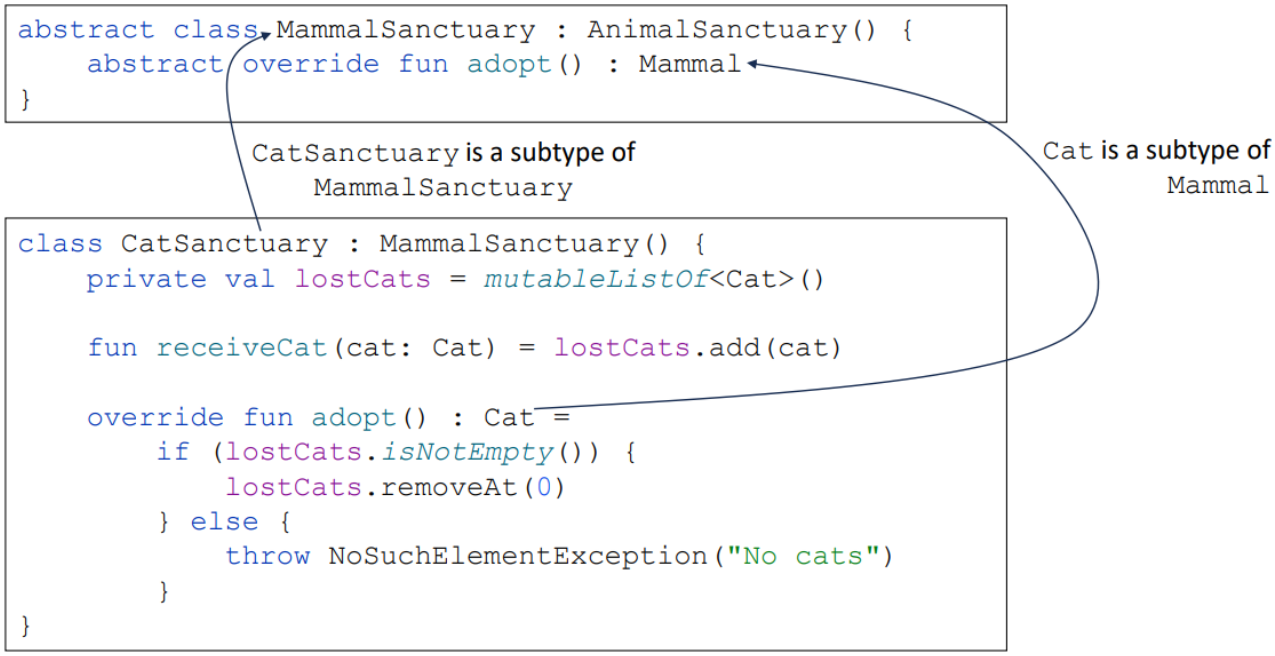
When overriding a method, we can make the return type more specific

```
abstract class Animal

abstract class Mammal : Animal()

class Cat : Mammal()
```





Method overriding: covariant return types

In general:

- Suppose open class A has open method `foo` with return type `T`
- Let U be a subtype of T
- Then subclass B of A can override `foo` with return type `U`

This is called covariance:

- Return type varies in the same direction as the enclosing class type
- Subclass B is below superclass A in the subtype relation
- Return type U is also below return type T in the subtype relation

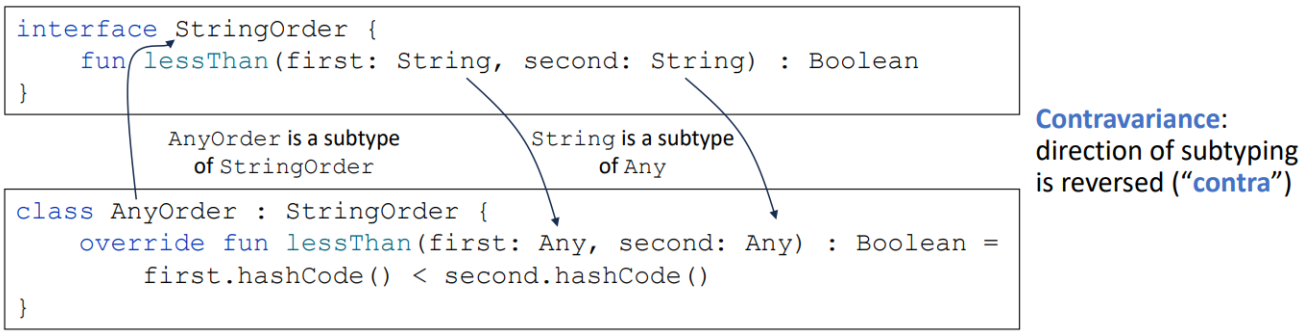
- The direction of subtyping is the same: this is what “co” in “covariance” means

Intuition for why return type covariance is OK:

- A client of A is prepared to receive any value of type T
- If the client turns out to actually be working with a B instance, it will receive a value of type U
- This is fine: a value of type U is a value of type T

Method overriding: contravariant argument types?

This could be allowed **in principle** (but is not allowed in Kotlin):



A client of `StringOrder` would use a `StringOrder` to order `String`s

An `AnyOrder` would do just fine instead: it can order `String`s because it can order `Any` thing

However, argument type contravariance is **not supported** in Kotlin (nor Java)

Function subtyping

Subtyping for function types is more general:

- Return type covariance is supported (as for method overriding)
- Argument type contravariance is supported (unlike for method overriding)

Argument type contravariance with function types

```
fun sortStrings(
    strings: List<String>,
    order: (String, String) -> Boolean,
) : List<String> {
    …
}
```

`(String, String) -> Boolean` is a subtype of `(Any, Any) -> Boolean` because `String` is a subtype of `Any`

```
fun main () {
    val myStrings = listOf("Hi", "There", "John")
    val anyOrder: (Any, Any) -> Boolean = { first, second ->
        first.hashCode() < second.hashCode()
    }
    println(sortStrings(myStrings, anyOrder))
}
```

This works

Function subtyping in general

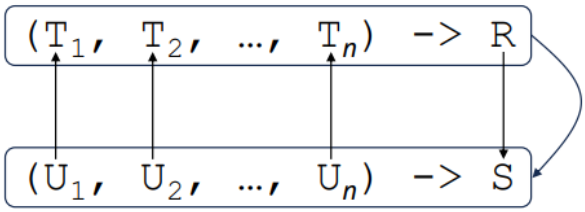
Consider a function type: $(U_1, U_2, \ldots, U_n) \rightarrow R$

Suppose U_i is a subtype of T_i (for each i)

Suppose S is a subtype of R

Then:

is a subtype of:



$B \rightarrow A$ means that B is a subtype of A

Picture is wrong, **get newest version!**

Increasing visibility when overriding

It is also OK to increase visibility when overriding:

```
abstract class AnimalSanctuary {
    protected abstract fun adopt() : Animal
}

abstract class MammalSanctuary : AnimalSanctuary() {
    abstract override fun adopt() : Mammal
}

class CatSanctuary : MammalSanctuary() {
    …

    public override fun adopt() : Cat = …
}
```

This method is implicitly `protected`: it overrides a `protected` method

We can increase visibility to `public` when overriding

```
fun tryToAdopt() {
    val catSanctuary: CatSanctuary = CatSanctuary()
    val mammalSantuary: MammalSanctuary = catSanctuary
    val animalSanctuary: AnimalSanctuary = catSanctuary

    catSanctuary.adopt()
    mammalSantuary.adopt()
    animalSanctuary.adopt()
}
```

At this point we have three references to a `CatSanctuary`, but with different apparent types

OK: adopt is `public` in `CatSanctuary`

Compile error: adopt is `protected` in `MammalSanctuary` and `AnimalSanctuary`

Overloading and nullable

Can we have these two overloads?

```
fun foo(arg: String?) = …

fun foo(arg: String) = …
```

Compile error: Platform declaration clash
The following methods have the same JVM signature: …

Not when targeting the JVM:

- Nullability is not a Java concept – in Java, any reference can potentially be null
- Java bytecode has no notion of nullability
- These methods compile to Java bytecode methods with the same signature