How flow-sensitive typing works in Kotlin

Nikita Bobko, Software Engineer @ JetBrains

How flow-sensitive typing works in Kotlin

Nikita Bobko, Software Engineer @ JetBrains

lengthOrZero in Java

```java
// JAVA
public static int lengthOrZero(Object any) {
    if (any instanceof String) {
        return ((String) any).length();
    } else {
        return 0;
    }
}
```

lengthOrZero in Java

```java
// JAVA
public static int lengthOrZero(Object any) {
    if (any instanceof String) {
        return ((String) any).length();
    } else {
        return 0;
    }
}
```

Smart-casts in Kotlin

```
// KOTLIN
fun lengthOrZero(any: Any): Int {
  if (any is String) {
     return any.length
  } else {
     return 0
  }
}
```

Smart-casts in Kotlin

```
// KOTLIN
fun lengthOrZero(any: Any): Int {
  if (any is String) {
     return any.length
  } else {
     any.length // error: unresolved reference: length
     return 0
  }
  any.length // error: unresolved reference: length
}
```

Smart-casts in Kotlin are powerful 🤯

```kotlin
fun isNotEmptyString(any: Any): Boolean {
  if (any !is String) return false
  return any.length != 0 // It also works
}

fun isNotEmptyString(any: Any): Boolean {
  return any is String && any.length != 0 // Yeap, works as well
}
```

Smart-casts in Kotlin are powerful 🤯

```kotlin
fun foo(any: Any) {
  if (any is String) any.length else return
  any.length // No problem, Kotlin can do it too
}
```

Smart-casts in Kotlin are powerful 🤯

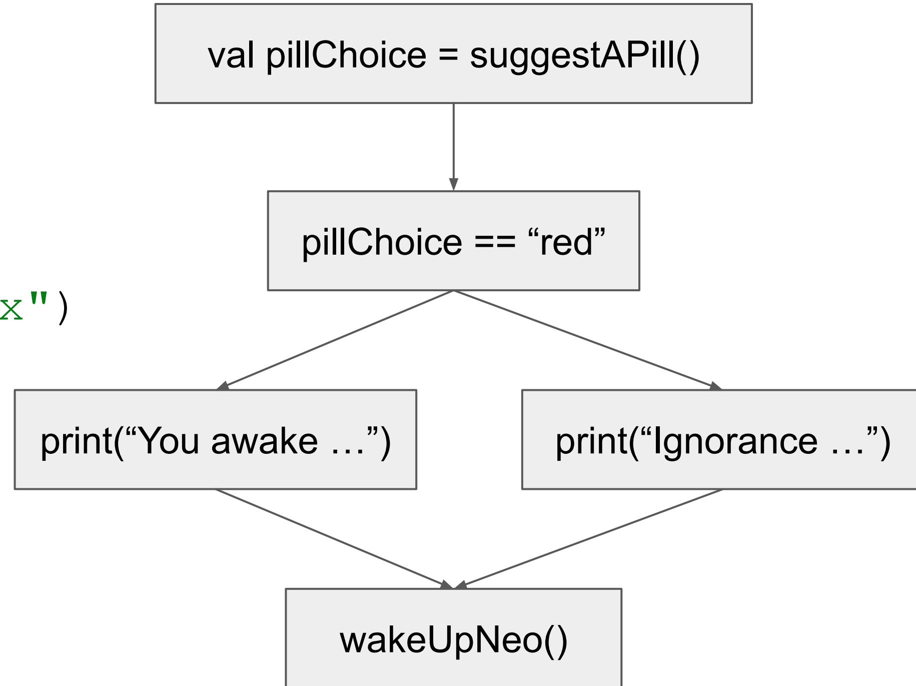
```
fun foo(any: Any) {
  if (any is String) any.length else return
  any.length // No problem, Kotlin can do it too
}
```

How would you implement
Kotlin smart-casts?

Control-flow graph (CFG)

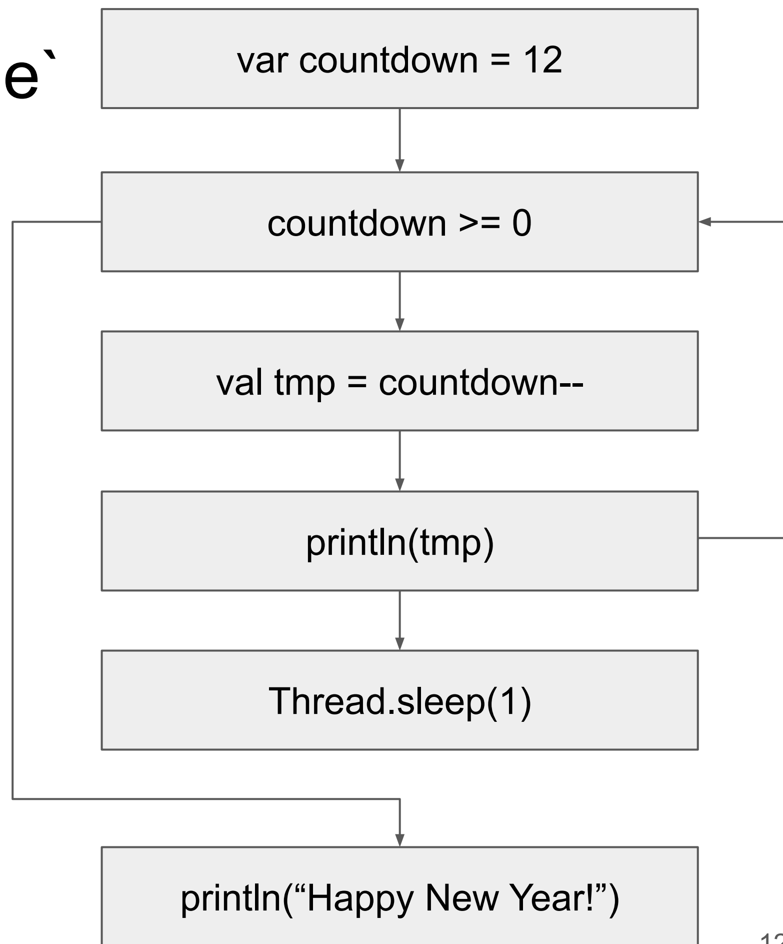
Control-flow graph (CFG) for `if`

```
val pillChoice = suggestAPill()
if (pillChoice == "red") {
  print("You awake from the " +
          "illusion of the Matrix")
} else {
  print("Ignorance is bliss!")
}
wakeUpNeo()
```



Control-flow graph (CFG) for `while`

```
var countdown = 12
while (countdown >= 0) {
  println(countdown--)
  Thread.sleep(1)
}
println("Happy New Year!")
```



Desugaring (aka “Compiler lowering”)

```
var counter = 0

foo(bar(counter++))
```

Desugaring (aka “Compiler lowering”)

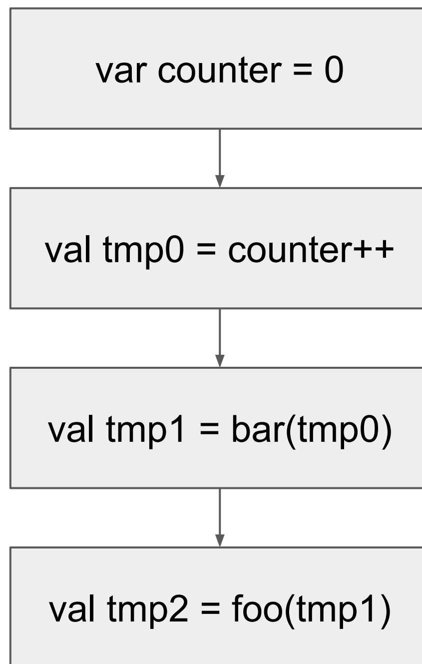
```
var counter = 0

val tmp0 = counter++

val tmp1 = bar(tmp0)

val tmp2 = foo(tmp1)
```

Split the program into minimal units.
Each unit has only one side-effect

Desugaring (aka “Compiler lowering”)

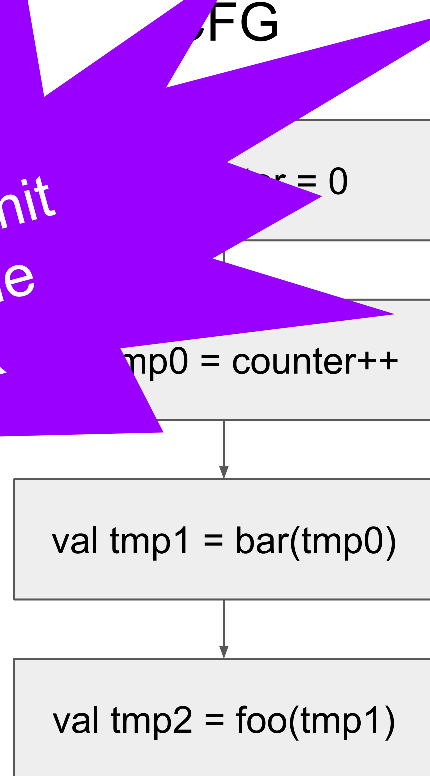
```
var counter = 0

val tmp0 = counter++

val tmp1 = bar(tmp0)

val tmp2 = foo(tmp1)
```

CFG



Desugaring (aka “Compiler lowering”)

```
var counter = 0
val tmp0 = counter++
val tmp1 = bar
val tmp2
```



Important! I will omit desugaring for the rest of the talk

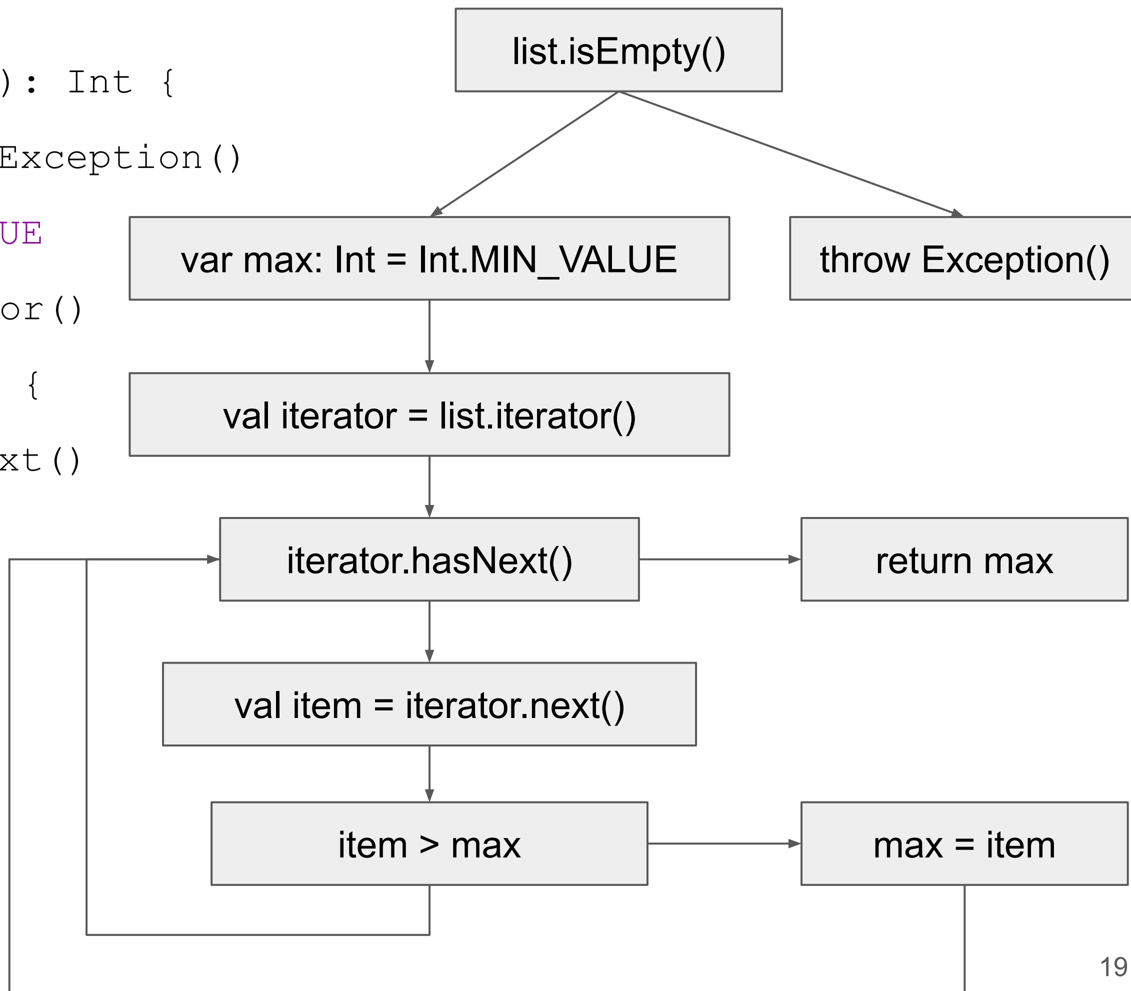
```kotlin
fun maxInList(list: List<Int>): Int {
    if (list.isEmpty()) throw Exception()
    var max: Int = Int.MIN_VALUE

    for (item in list) {

        if (item > max) {
            max = item
        }
    }
    return max
}
```

```kotlin
fun maxInList(list: List<Int>): Int {
    if (list.isEmpty()) throw Exception()
    var max: Int = Int.MIN_VALUE
    val iterator = list.iterator()
    while (iterator.hasNext()) {
        val item = iterator.next()
        if (item > max) {
            max = item
        }
    }
    return max
}
```

Desugar

(aka "Compiler lowering")

```kotlin
fun maxInList(list: List<Int>): Int {
    if (list.isEmpty()) throw Exception()
    var max: Int = Int.MIN_VALUE
    val iterator = list.iterator()
    while (iterator.hasNext()) {
        val item = iterator.next()
        if (item > max) {
            max = item
        }
    }
    return max
}
```



Control flow analysis applications

- Dead code elimination optimization
- Loop unrolling optimization
- Escape analysis optimization
  - what variables should be allocated on the stack and which ones should escape to the heap
  - Allocations eliminations
- Check that variable is initialized before used
- IDE analysis
- …

Control flow analysis applications

- Dead code elimination optimization
- Loop unrolling optimization
- Escape analysis optimization
  - what variables should be allocated on the stack and which ones should escape to the heap
  - Allocations eliminations
- Check that variable is initialized before used
- IDE analysis
- …
- Flow-sensitive typing implementation

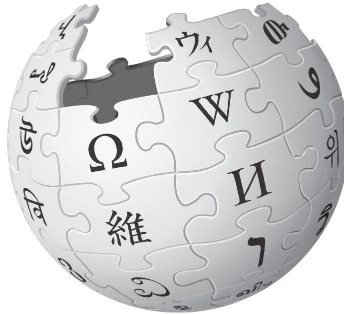
How flow-sensitive typing works in Kotlin

Nikita Bobko, Software Engineer @ JetBrains

Flow-sensitive typing. The definition. Finally!

In programming language theory, **flow-sensitive typing** (also called flow typing or occurrence typing) is a type system where the type of an expression depends on its position in the control flow.

Smart-casts in Kotlin is a special case of flow-sensitive typing



Wikipedia
The Free Encyclopedia

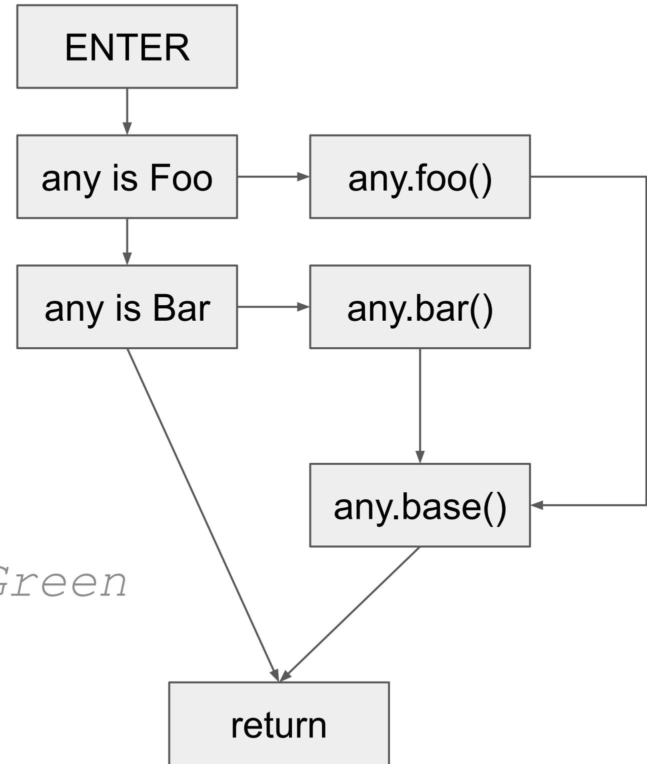
Data-flow (DF) framework

Data-flow (DF) framework

```
interface Base { fun base() }
interface Foo : Base { fun foo() }
interface Bar : Base { fun bar() }
fun main(any: Any) {
    if (any is Foo) any.foo() // Green
    else if (any is Bar) any.bar() // Green
    else return
    any.base() // Green (in Kotlin 2.0)
}
```

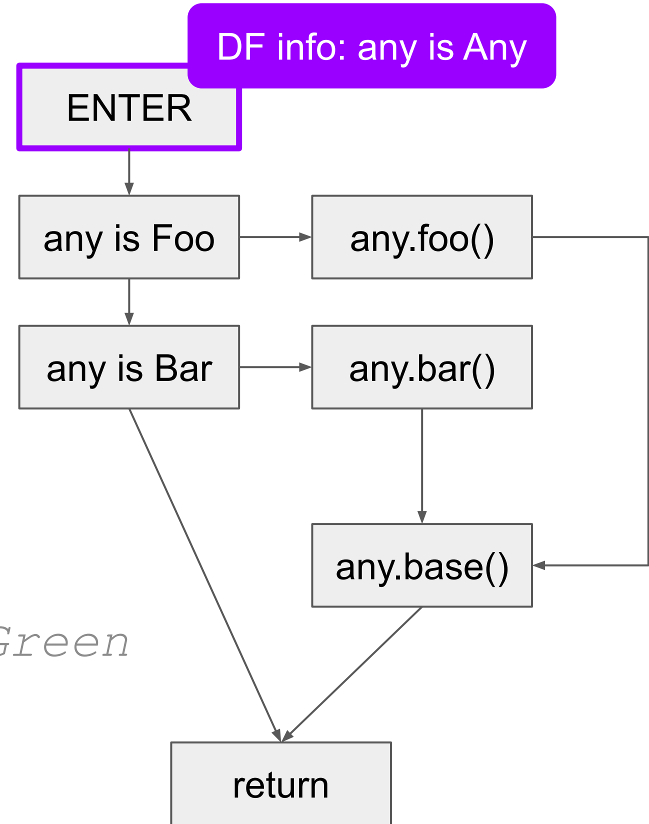
Data-flow (DF) framework

```
interface Base { fun base() }
interface Foo : Base { fun foo() }
interface Bar : Base { fun bar() }
fun main(any: Any) {
    if (any is Foo) any.foo() // Green
    else if (any is Bar) any.bar() // Green
    else return
    any.base() // Green (in Kotlin 2.0)
}
```



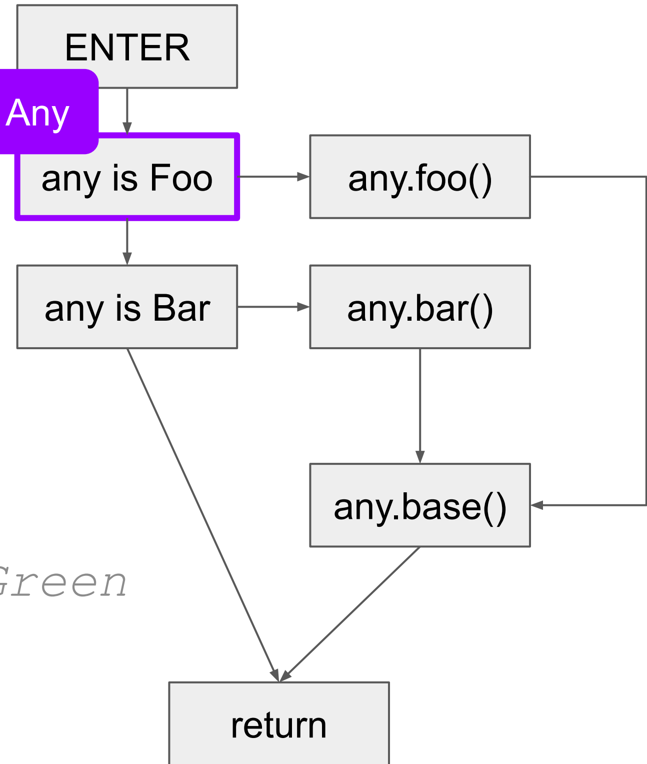
Data-flow (DF) framework

```
interface Base { fun base() }
interface Foo : Base { fun foo() }
interface Bar : Base { fun bar() }
fun main(any: Any) {
    if (any is Foo) any.foo() // Green
    else if (any is Bar) any.bar() // Green
    else return
    any.base() // Green (in Kotlin 2.0)
}
```



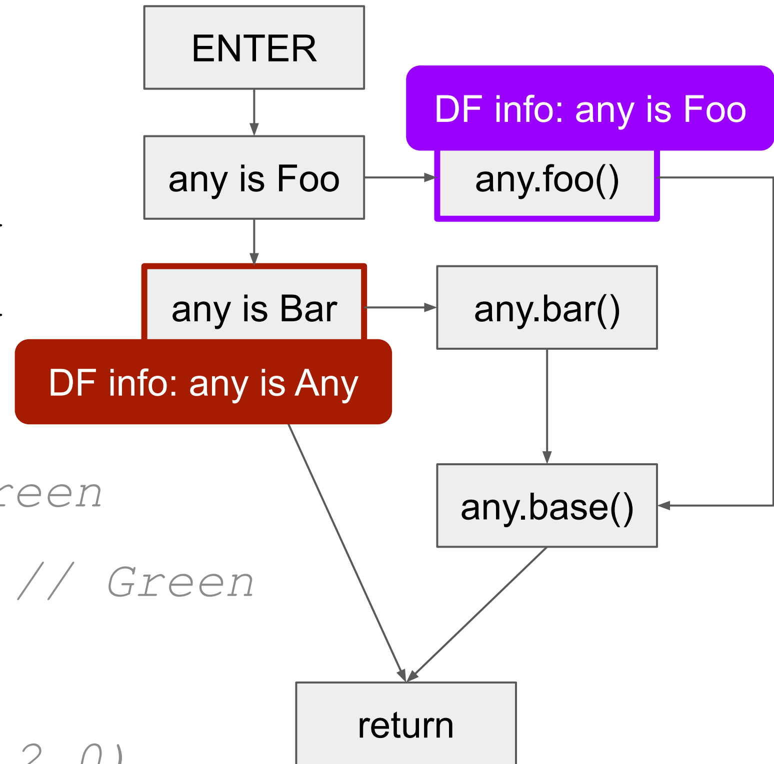
Data-flow (DF) framework

```
interface Base { fun base() }
interface Foo : Base { fun foo() }
interface Bar : Base { fun bar() }
fun main(any: Any) {
    if (any is Foo) any.foo() // Green
    else if (any is Bar) any.bar() // Green
    else return
    any.base() // Green (in Kotlin 2.0)
}
```



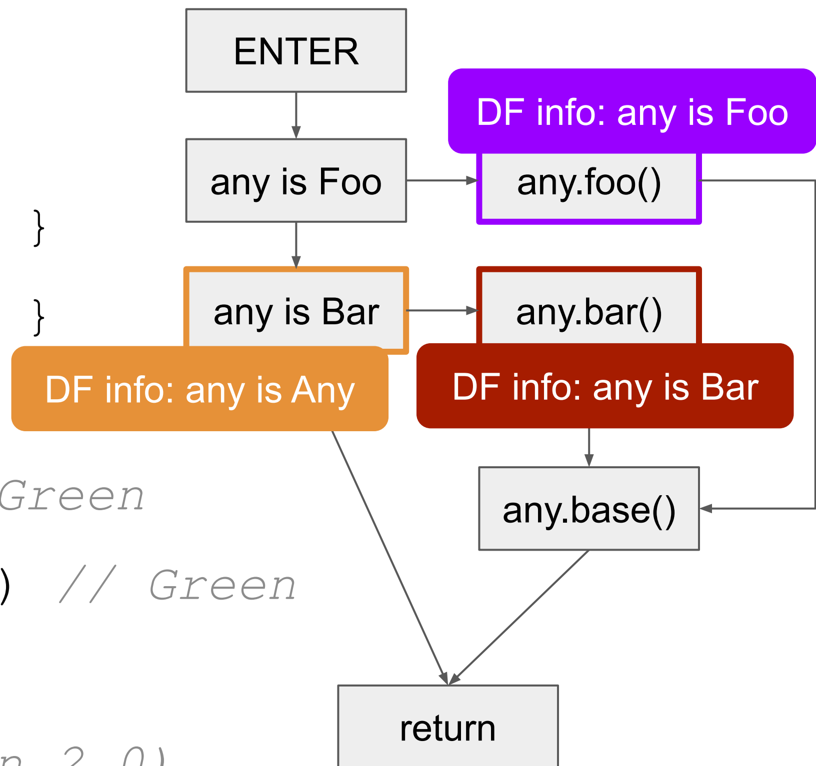
Data-flow (DF) framework

```kotlin
interface Base { fun base() }
interface Foo : Base { fun foo() }
interface Bar : Base { fun bar() }
fun main(any: Any) {
    if (any is Foo) any.foo() // Green
    else if (any is Bar) any.bar() // Green
    else return
    any.base() // Green (in Kotlin 2.0)
}
```



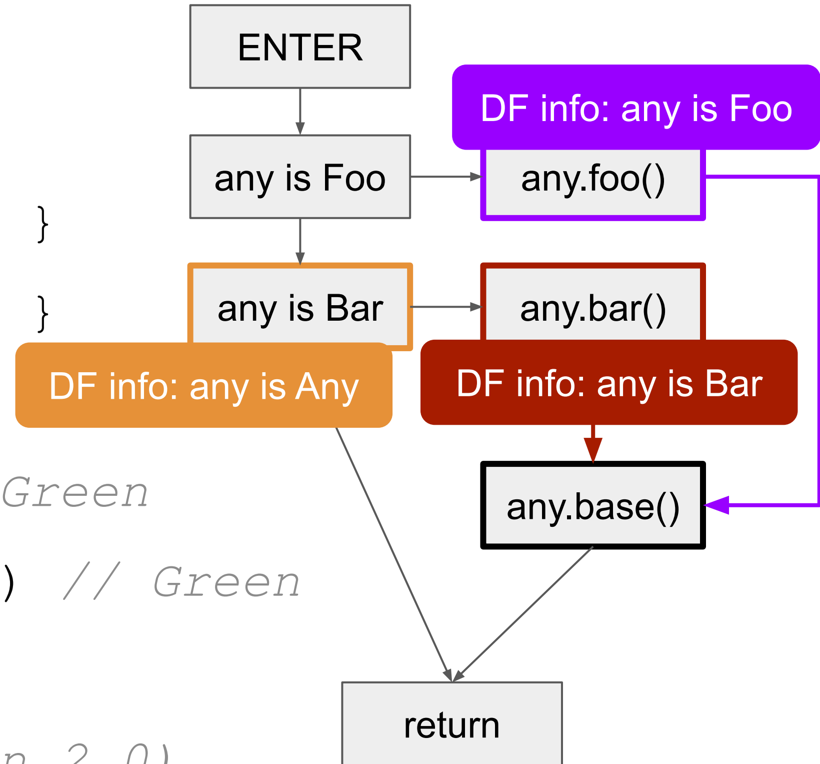
Data-flow (DF) framework

```
interface Base { fun base() }
interface Foo : Base { fun foo() }
interface Bar : Base { fun bar() }
fun main(any: Any) {
    if (any is Foo) any.foo() // Green
    else if (any is Bar) any.bar() // Green
    else return
    any.base() // Green (in Kotlin 2.0)
}
```



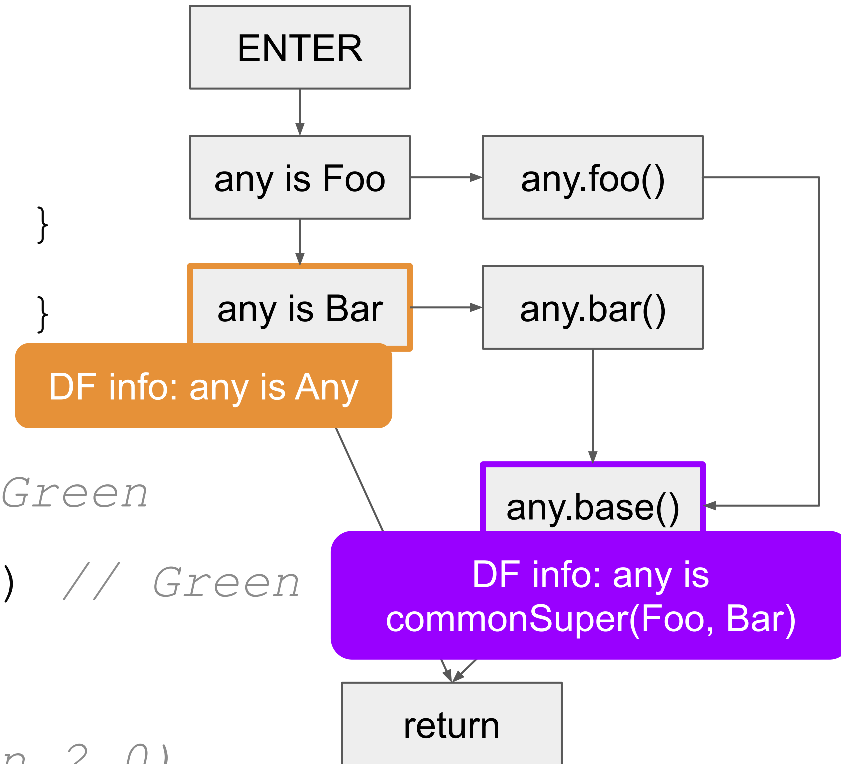
Data-flow (DF) framework

```
interface Base { fun base() }
interface Foo : Base { fun foo() }
interface Bar : Base { fun bar() }
fun main(any: Any) {
    if (any is Foo) any.foo() // Green
    else if (any is Bar) any.bar() // Green
    else return
    any.base() // Green (in Kotlin 2.0)
}
```



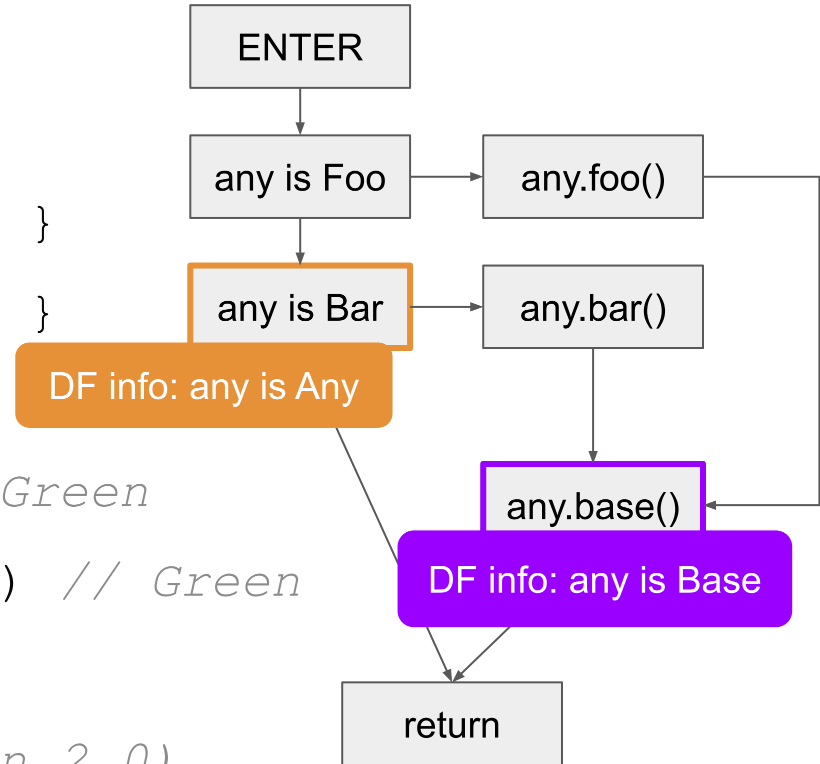
Data-flow (DF) framework

```
interface Base { fun base() }
interface Foo : Base { fun foo() }
interface Bar : Base { fun bar() }
fun main(any: Any) {
    if (any is Foo) any.foo() // Green
    else if (any is Bar) any.bar() // Green
    else return
    any.base() // Green (in Kotlin 2.0)
}
```



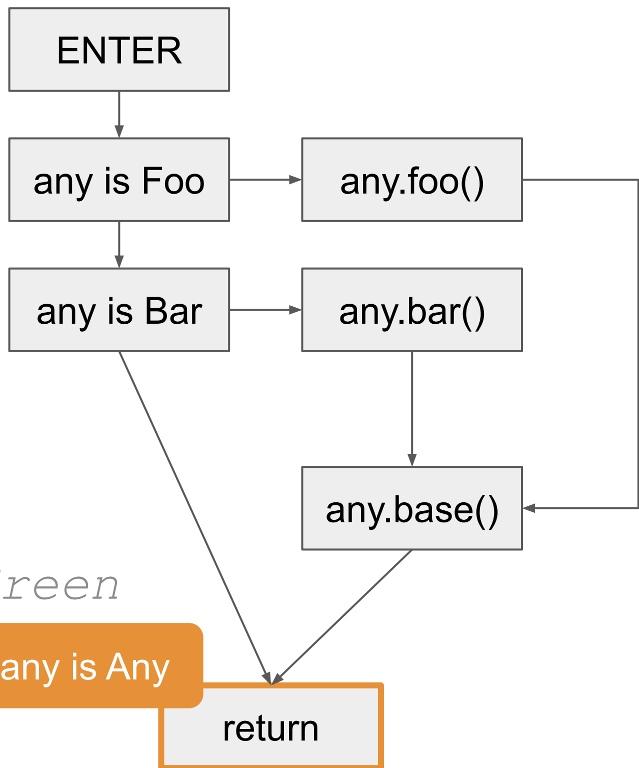
Data-flow (DF) framework

```
interface Base { fun base() }
interface Foo : Base { fun foo() }
interface Bar : Base { fun bar() }
fun main(any: Any) {
    if (any is Foo) any.foo() // Green
    else if (any is Bar) any.bar() // Green
    else return
    any.base() // Green (in Kotlin 2.0)
}
```



Data-flow (DF) framework

```
interface Base { fun base() }
interface Foo : Base { fun foo() }
interface Bar : Base { fun bar() }
fun main(any: Any) {
    if (any is Foo) any.foo() // Green
    else if (any is Bar) any.bar() // Green
    else return
    any.base() // Green (in Kotlin 2.0)
}
```



Symbol resolution

“Symbol resolution” depends on “Smart-casts inference”

```
class Foo { fun foo() { /*...*/ } }

class Bar { fun foo() { /*...*/ } }

fun function(any: Any) {
   if (any is Bar) any.foo()
   if (any is Foo) any.foo()
}
```

“Symbol resolution” depends on “Smart-casts inference”

```
class Foo { fun foo() { /*...*/ } }

class Bar { fun foo() { /*...*/ } }

fun function(any: Any) {
    if (any is Bar) any.foo()
    if (any is Foo) any.foo()
}
```

“Resolves to”

“Resolves to”

“Resolves to” relation works like “Go to definition” in your IDE

“Smart-casts inference” depends on “Symbol resolution”

```
val foo: Any = ""
fun bar() {
    if (foo is String) {
        foo.length


        val foo: Any = ""


        foo.length // error: unresolved reference: 'length'
    }
}
```

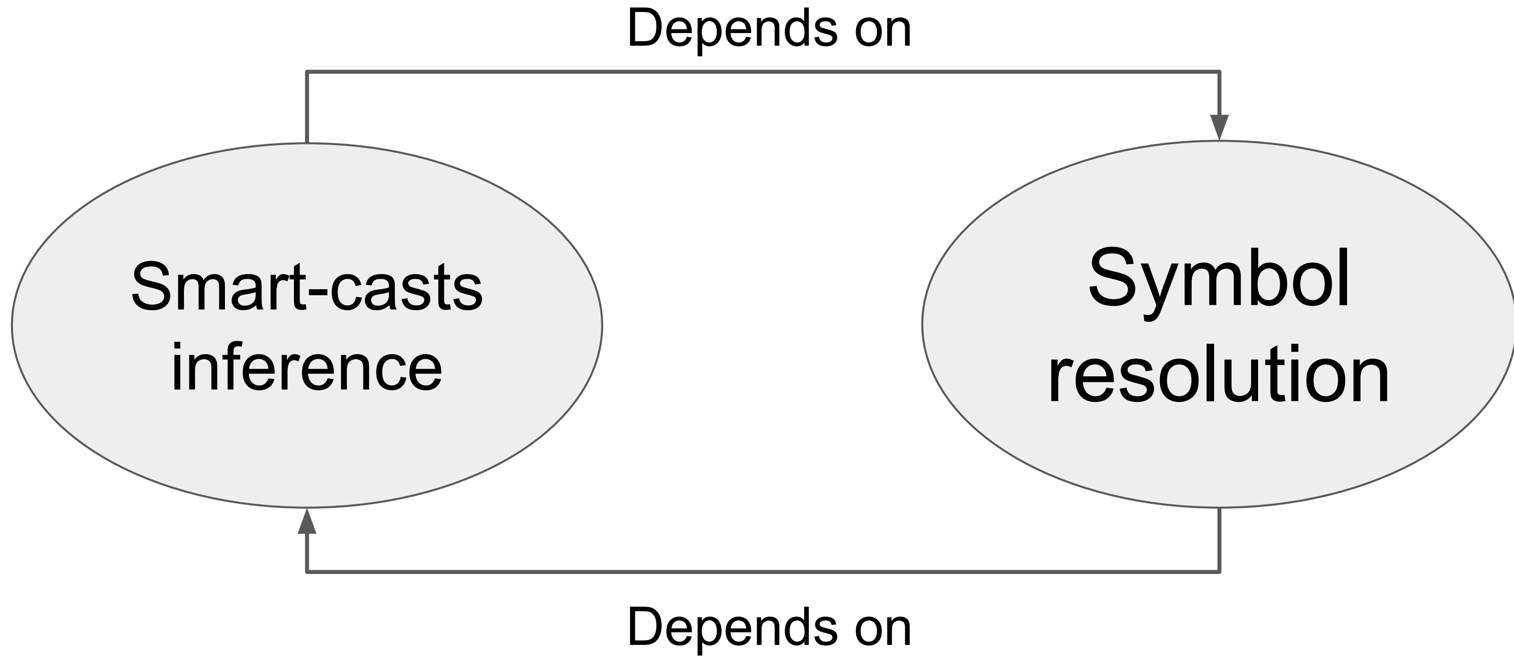
“Smart-casts inference” depends on “Symbol resolution”

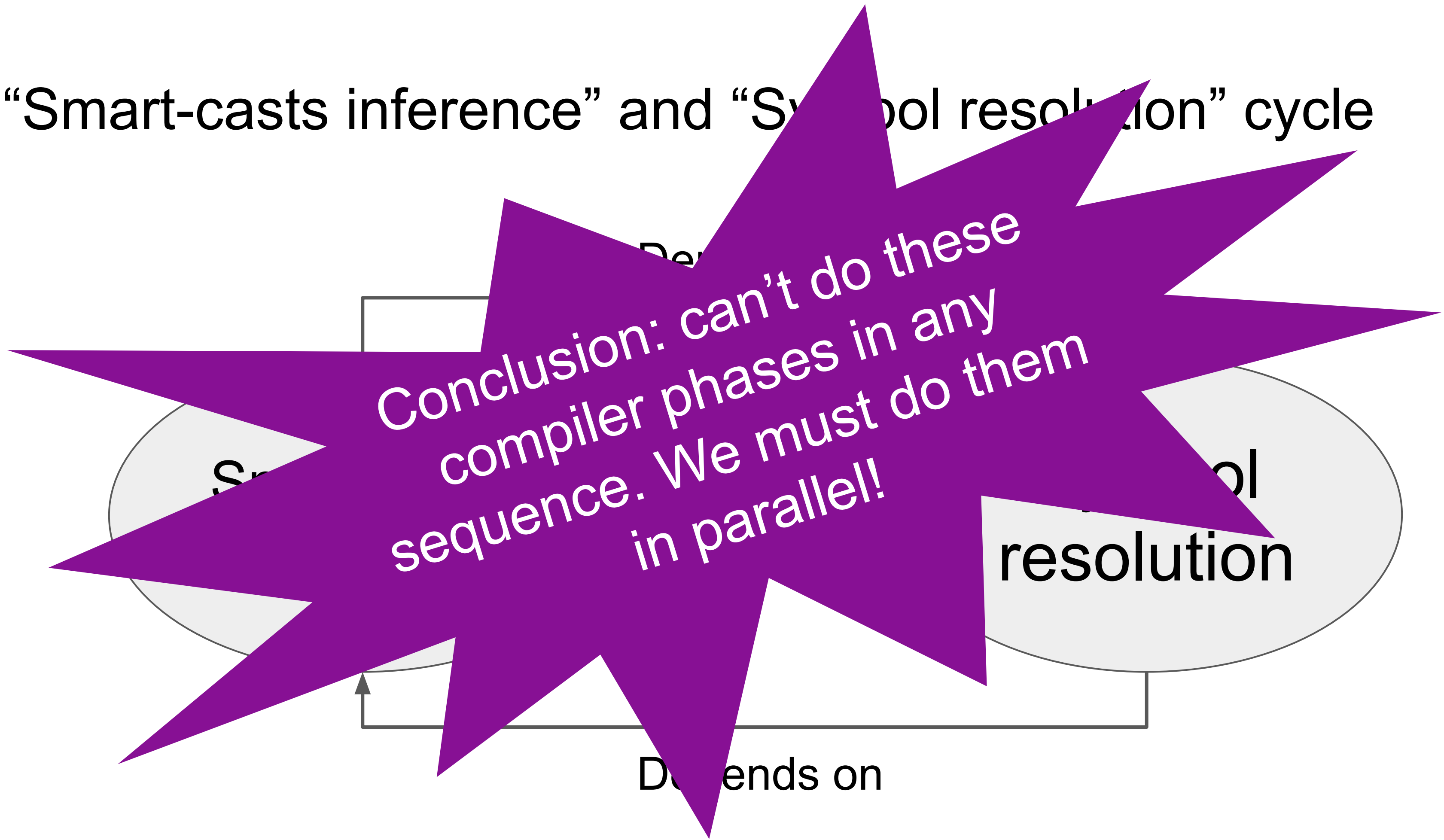
```
val foo: Any = ""
fun bar() {
    if (foo is String) {
        foo.length
```

```
        val foo: Any = ""
        foo.length // error: unresolved reference: 'length'
    }
}
```

Won’t smart-cast this ‘foo’ because it **resolves to** a different ‘foo’

“Smart-casts inference” and “Symbol resolution” cycle




“Smart-casts inference” and
cycle
Conclusion: can’t do these
compiler phases in any
sequence. We must do them
in parallel!
resolution

Resolution and smart-casts are performed together

```
val foo: Any = ""

fun bar() {

    if (foo is String) {

        foo.length


        val foo: Any = ""


        foo.length // error: unresolved reference: 'length'

    }

}
```

Current step:

foo is resolved to global.foo

The analysis is performed from top to bottom in CFG, together with smart-casts

Resolution and smart-casts are performed together

```
val foo: Any = ""

fun bar() {

    if (foo is String) {

        foo.length


        val foo: Any = ""


        foo.length // error: unresolved reference: 'length'

    }

}
```

Current step:

global.foo is smart-casted to String

The analysis is performed from top to bottom in CFG, together with smart-casts

Resolution and smart-casts are performed together

```
val foo: Any = ""

fun bar() {
    if (foo is String) {
        foo.length

        val foo: Any = ""

        foo.length // error: unresolved reference: 'length'
    }
}
```

Current step:

foo is resolved to global.foo

The analysis is performed from top to bottom in CFG, together with smart-casts

Resolution and smart-casts are performed together

```
val foo: Any = ""

fun bar() {

    if (foo is String) {

        foo.length


            val foo: Any = ""


            foo.length // error: unresolved reference: 'length'

    }

}
```

Current step:

Smart-cast is applied to foo

The analysis is performed from top to bottom in CFG, together with smart-casts

Resolution and smart-casts are performed together

```
val foo: Any = ""

fun bar() {

    if (foo is String) {

        foo.length


        val foo: Any = ""


        foo.length // error: unresolved reference: 'length'

    }

}
```

Current step:

String.length is resolved

The analysis is performed from top to bottom in CFG, together with smart-casts

Resolution and smart-casts are performed together

```
val foo: Any = ""

fun bar() {

    if (foo is String) {

        foo.length


        val foo: Any = ""


        foo.length // error: unresolved reference: 'length'

    }

}
```

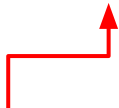
Current step:

New variable `foo` is defined

The analysis is performed from top to bottom in CFG, together with smart-casts

Resolution and smart-casts are performed together

```
val foo: Any = ""

fun bar() {

    if (foo is String) {

        foo.length


        val foo: Any = ""



        foo.length // error: unresolved reference: 'length'

    }

}
```



Current step:

foo is resolved to local.foo

The analysis is performed from top to bottom in CFG, together with smart-casts

Resolution and smart-casts are performed together

```
val foo: Any = ""

fun bar() {

    if (foo is String) {

        foo.length


        val foo: Any = ""


        foo.length // error: unresolved reference: 'length'

    }

}
```

Current step:

Any.length can’t be resolved

The analysis is performed from top to bottom in CFG, together with smart-casts

Loops analysis

Will it compile?

```
var any: Any = ""
if (any is String) {
   any.length
   while (true) {
       any.length
       if (any is String) any = 1
       else if (any is Int) any = ""
   }
}
```

Compilation error. How does Kotlin compiler understand?

```kotlin
var any: Any = ""
if (any is String) {
    any.length // Green code
    while (true) {
        any.length // error: unresolved reference: length
        if (any is String) any = 1
        else if (any is Int) any = ""
    }
}
```

Compilation error. How does Kotlin compiler understand?

```
var any: Any = ""
if (any is String) {
    any.length // Green code
    while (true) {
        any.length // error: unresolved reference: length
        if (any is String) any = 1
        else if (any is Int) any = ""
    }
}
```

Before analyzing loops in CFG, Kotlin discards all data-flow information for symbols that are mutated inside the loop

Compilation error. How does Kotlin compiler understand?

```
var any: Any = ""
if (any is String) {
    any.length // Green code
    while (true) {
        any.length // error: unresolved reference: length
        if (any is String) any = 1
        else if (any is Int) any = ""
    }
}
```

Before analyzing loops in CFG, Kotlin discards all data-flow information for symbols that are mutated inside the loop

Compilation error. How does Kotlin compiler understand?

```kotlin
var any: Any = ""
if (any is String) {
    any.length // Green code
    while (true) {
        any.length // error: unresolved reference: length
        if (any is String) any = 1
        else if (any is Int) any = ""
    }
}
```

Before analyzing loops in CFG, Kotlin discards all data-flow information for **symbols** that are mutated inside the loop

Not yet resolved symbols
Not yet visited part of CFG
Unresolved code

We don't know whether those symbols are the same

Compilation error. How does Kotlin compiler understand?

```
var any: Any = ""
if (any is String) {
    any.length // Green code
    while (true) {
        any.length // error: unresolved reference: length
        if (any is String) any = 1
        else if (any is Int) any = ""
    }
}
```

The same name

Before analyzing loops in CFG, Kotlin discards all data-flow information for symbols with the same names that are mutated inside the loop

Approximation!

Will it compile?

```
var any: Any = ""
if (any is String) {
    any.length
    while (true) {
        any.length
        var any: String = ""
        any = ""
    }
}
```

Before analyzing loops in CFG, Kotlin discards all data-flow information for symbols with the same names that are mutated inside the loop

Compilation error. False positive :(

```
var any: Any = ""
if (any is String) {
    any.length // Green code
    while (true) {
        any.length // error: unresolved reference: length
        var any: String = ""
        any = ""
    }
}
```

Before analyzing loops in CFG, Kotlin discards all data-flow information for symbols **with the same names** that are mutated inside the loop

Approximation!

Compilation error. False positive :(

```
var any: Any = ""
if (any is String) {
    any.length // Green code
    while (true) {
        any.length // Green code
        // var any: String = ""
        // any = ""
    }
}
```

Before analyzing loops in CFG, Kotlin discards all data-flow information for symbols **with the same names** that are mutated inside the loop

Capturing closures/lambdas analysis

Will it compile?

```
var any: Any = ""
if (any is String) {
    any.length
    Thread({
        any.length
    }).start()
    any = 1
}
// etc
```

Compilation error. How does Kotlin compiler understand?

```
var any: Any = ""
if (any is String) {
    any.length // Green code
    Thread({
        any.length // error: SMARTCAST_IMPOSSIBLE
    }).start()
    any = 1
}
// etc
```

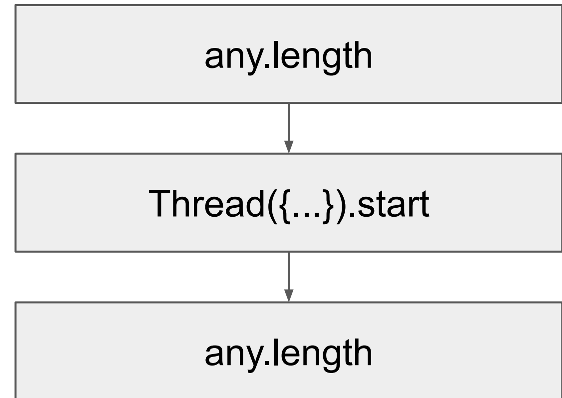
Compilation error. How does Kotlin compiler understand?

```
var any: Any = ""
if (any is String) {
    any.length // Green code
    Thread({
        any = 1
    }).start()
    any.length // error: SMARTCAST_IMPOSSIBLE
}
// etc
```

Compilation error. How does Kotlin compiler understand?

```
var any: Any = ""
if (any is String) {
    any.length // Green code
    Thread({
        any = 1
    }).start()
    any.length // error: SMARTCAST_IMPOSSIBLE
}
// etc
```

The CFG is linear, no branching!
The lambda has its own CFG



Compilation error. How does Kotlin compiler understand?

```kotlin
var any: Any = ""
if (any is String) {
    any.length // Green code
    Thread({
        any = 1
    }).start()
    any.length // error: SMARTCAST_IMPOSSIBLE
}
// etc
```

(1)

(2)

Before analyzing (1) and (2) CFG subgraphs, Kotlin forbids smart-casts for symbols that are mutated in (1) and (2)

Important! (1) and (2) mark all CFG nodes reachable from the beginning of (1) and (2)

Compilation error. How does Kotlin compiler understand?

```
var any: Any = ""
if (any is String) {
    any.length // Green code
    Thread({
        any = 1
    }).start()
    any.length // error: SMARTCAST_IMPOSSIBLE
}
// etc
```

(1)

(2)

Before analyzing (1) and (2) CFG subgraphs, Kotlin forbids smart-casts for symbols that are mutated in (1) and (2)

Important! (1) and (2) mark all CFG nodes reachable from the beginning of (1) and (2)

Compilation error. How does Kotlin compiler understand?

```kotlin
var any: Any = ""
if (any is String) {
    any.length // Green code
    Thread({
        any = 1
    }).start()
    any.length // error: SMARTCAST_IMPOSSIBLE
}
// etc
```

(1)

(2)

Before analyzing (1) and (2) CFG subgraphs, Kotlin forbids smart-casts for **symbols with the same names** that are mutated in (1) and (2)

Approximation!

Important! (1) and (2) mark all CFG nodes reachable from the beginning of (1) and (2)

False positive compilation error. Again :(

```
var any: Any = ""
if (any is String) {
    any.length // Green code
    Thread({
        var any: String = ""
        any = ""
    }).start()
    any.length // error: SMARTCAST_IMPOSSIBLE
}
```

Backwards edges + capturing closures feature interaction

Will it compile?

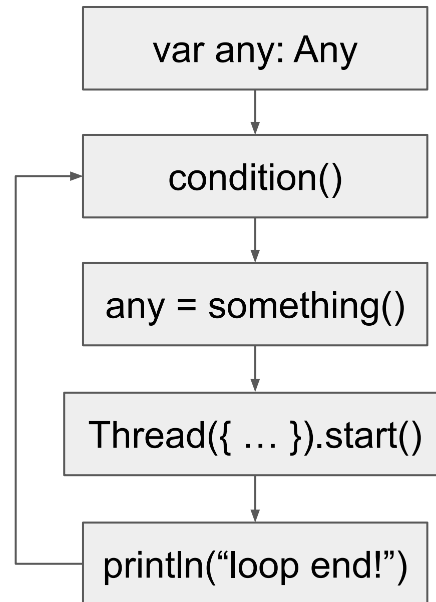
```
fun something(): String {/*...*/}


var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```

This code is fine!

```
fun something(): String {/*...*/}


var any: Any
while (condition()) {
    any = something()
    Thread({ any.length }) // Green code
        .start()
    println("loop end!")
}
```

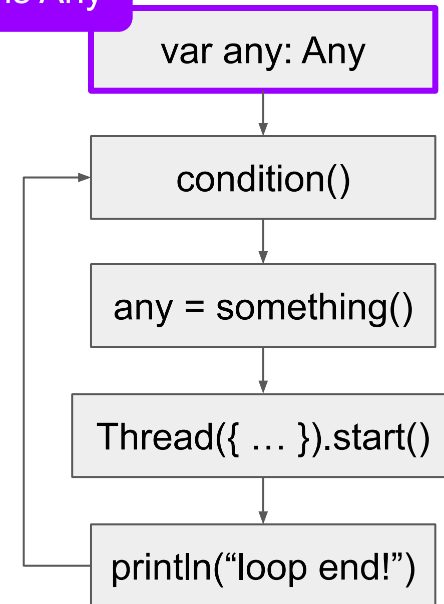
Backwards edges + capturing closures feature interaction

```
fun something(): String {/*...*/}

var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```



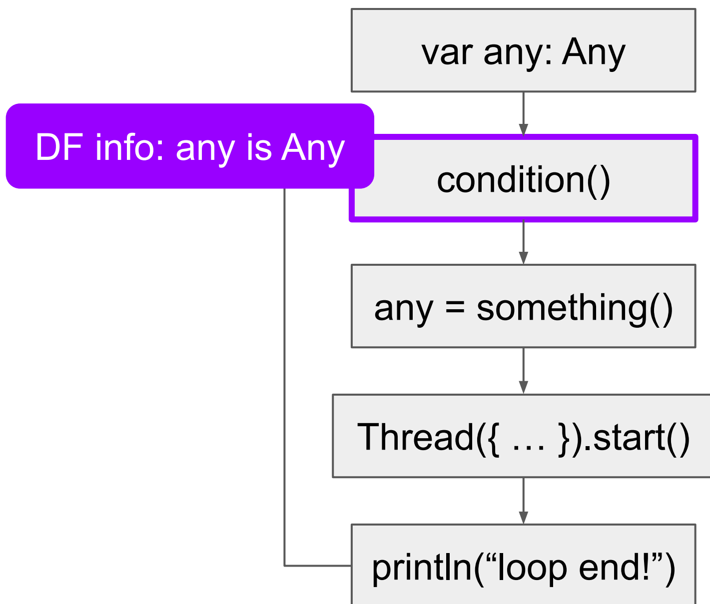
Backwards edges + capturing closures feature interaction

```
fun something(): String {/*...*/}

var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```



Backwards edges + capturing closures feature interaction

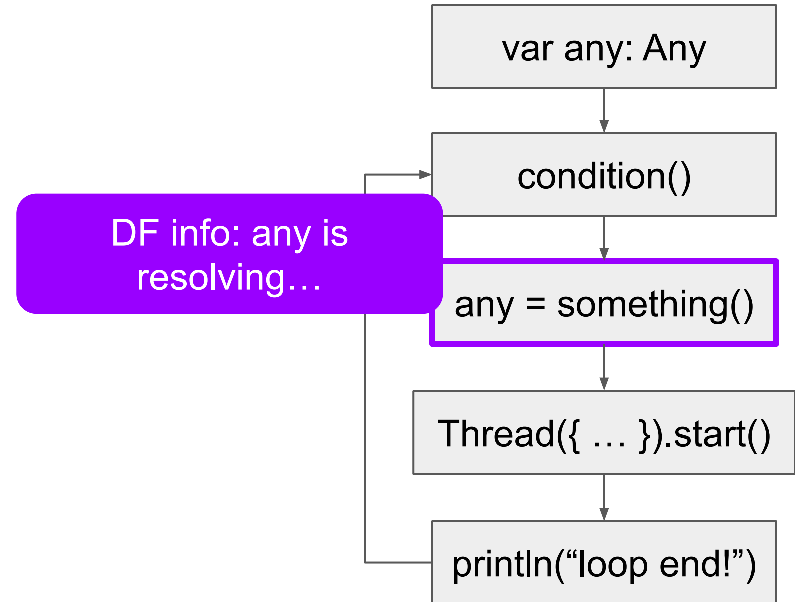
```
fun something(): String {/*...*/}


var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```



Backwards edges + capturing closures feature interaction

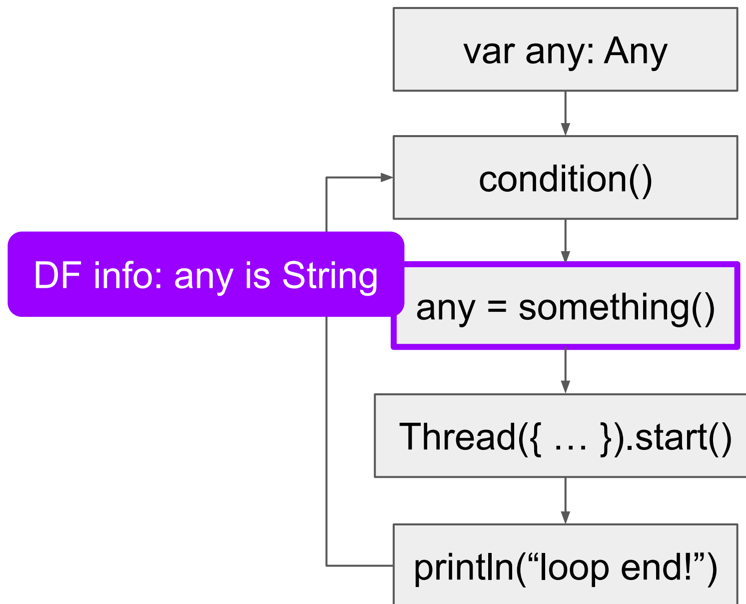
```
fun something(): String {/*...*/}

var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```

“Resolves to”



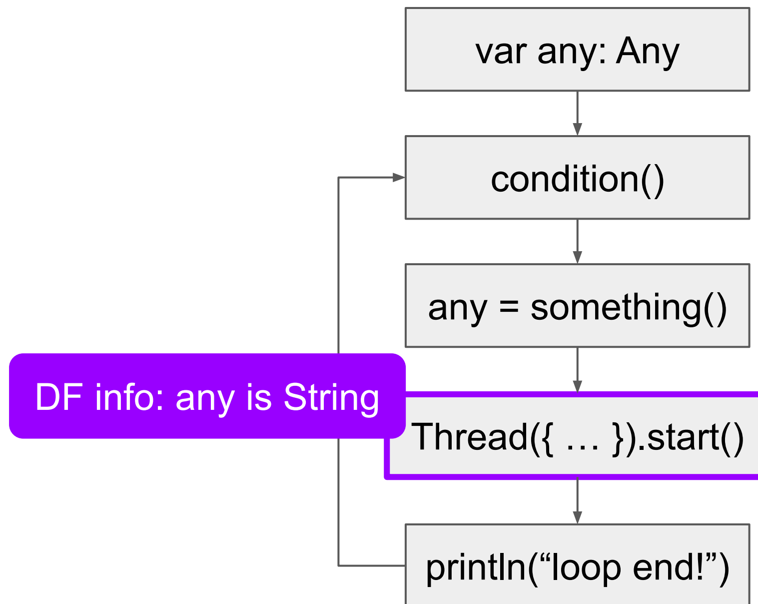
Backwards edges + capturing closures feature interaction

```
fun something(): String {/*...*/}

var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```



Backwards edges + capturing closures feature interaction

```
fun something(): String {/*...*/}

var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```



Bac   on

fun
var
whil
a
T
p
}

Compilation error. How does Kotlin compiler understand?

```
var any: Any = ""
if (any is String) {
    any.length // Green code
    Thread({
        any = 1
    }).start()
    any.length // error: SMARTCAST_IMPOSSIBLE
}
// etc
```

(1)

(2)

Before analyzing (1) and (2) CFG subgraphs, Kotlin forbids smart-casts for **symbols with the same names** that are mutated in (1) and (2)

Important! (1) and (2) mark all CFG nodes reachable from the beginning of (1) and (2)

62

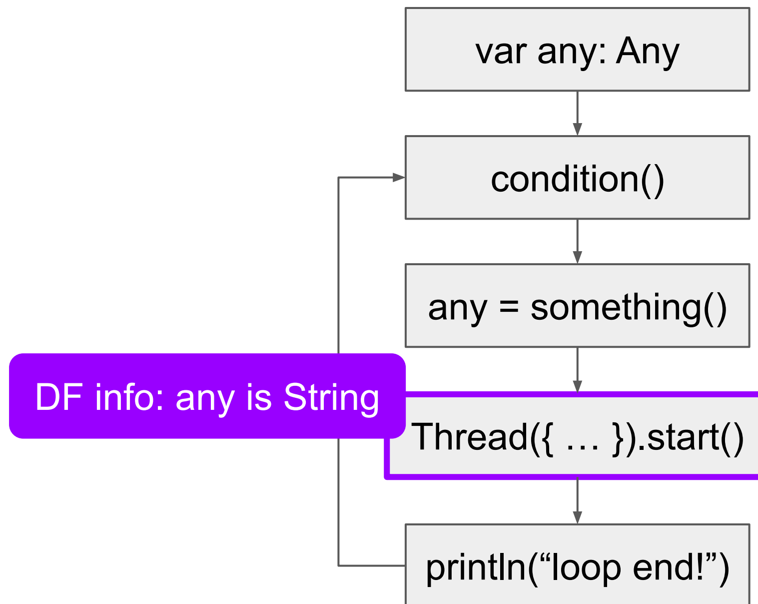
g()

art()

d!")

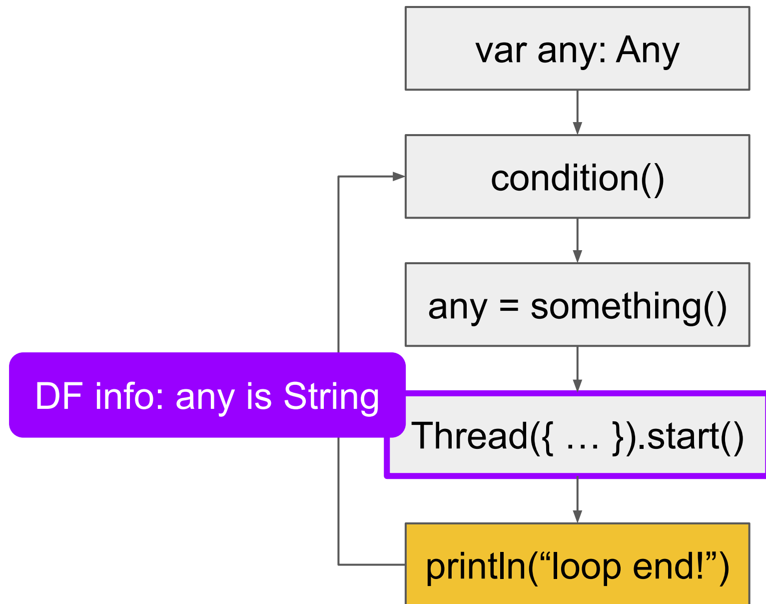
Backwards edges + capturing closures feature interaction

```
fun something(): String {/*...*/}

var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```



Backwards edges + capturing closures feature interaction

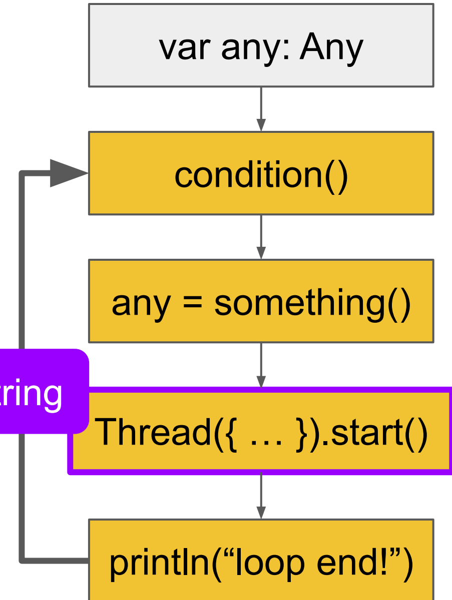
```
fun something(): String {/*...*/}

var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```



Backwards edges + capturing closures feature interaction

```
fun something(): String {/*...*/}


var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```

Nodes are reachable via the backwards edge!

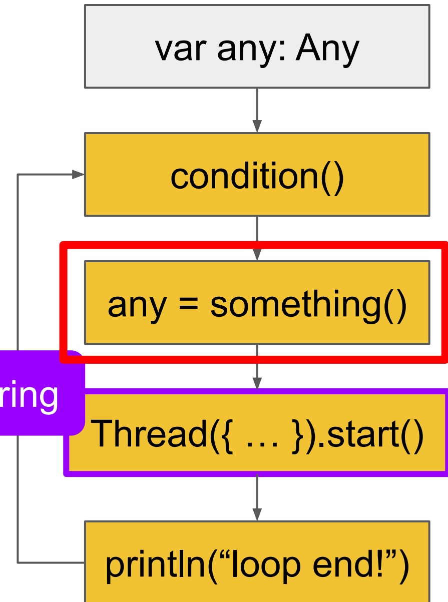


Backwards edges + capturing closures feature interaction

```
fun something(): String {/*...*/}

var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```

Mutation! :(

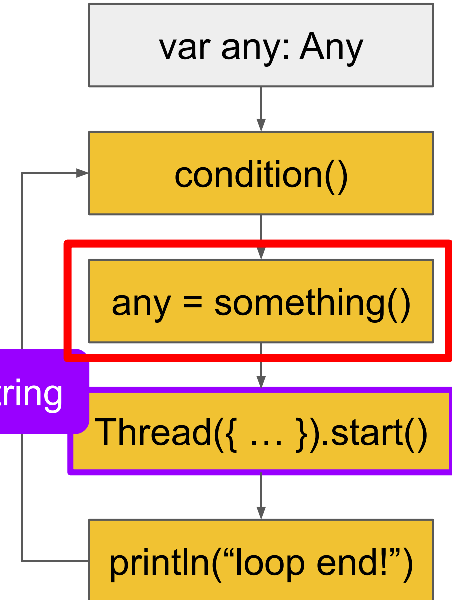
DF info: any is String



Backwards edges + capturing closures feature interaction

Mutation! :(

DF info: any is String

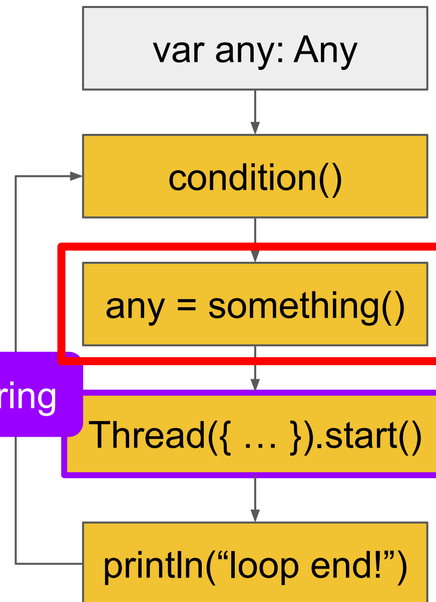


Backwards edges + capturing closures feature interaction

```
fun something(): String {/*...*/}

var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```

Mutation! :(

DF info: any is String

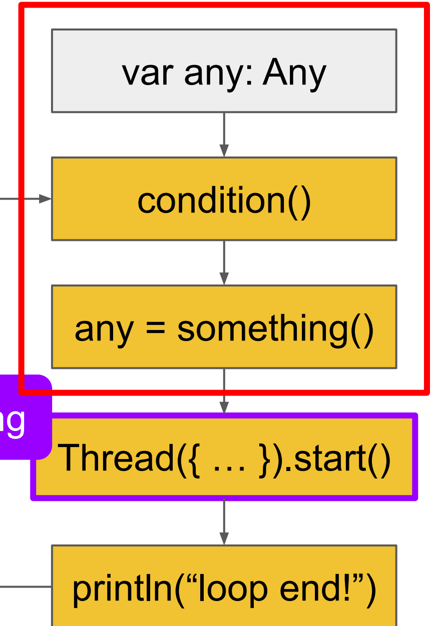


Backwards edges + capturing closures feature interaction

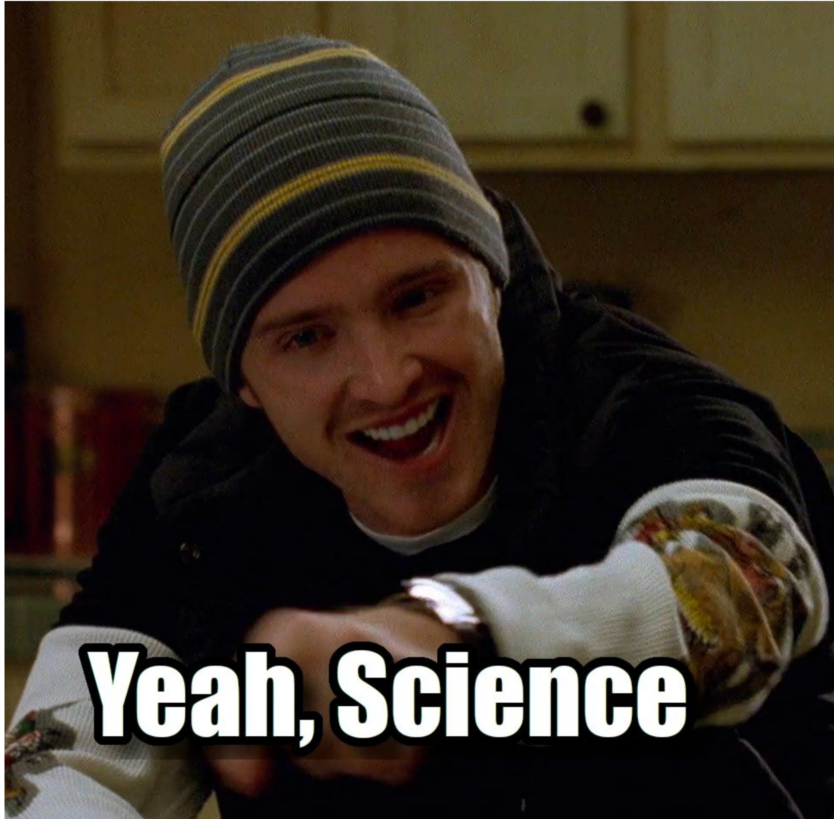
```
fun something(): String {/*...*/}


var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```

Symbols in the red subgraph are already resolved!

We already know something() return type



Backwards edges + capturing closures feature interaction

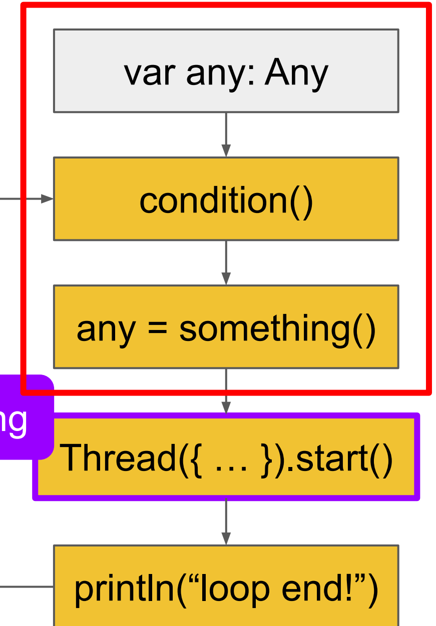


/*...*/}

Symbols in the red subgraph are already resolved!

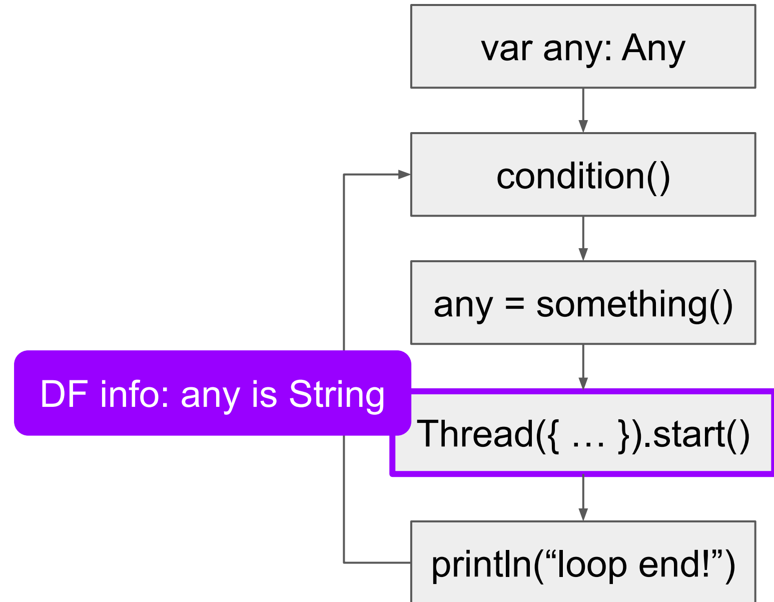
We already know something() return type

DF info: any is String



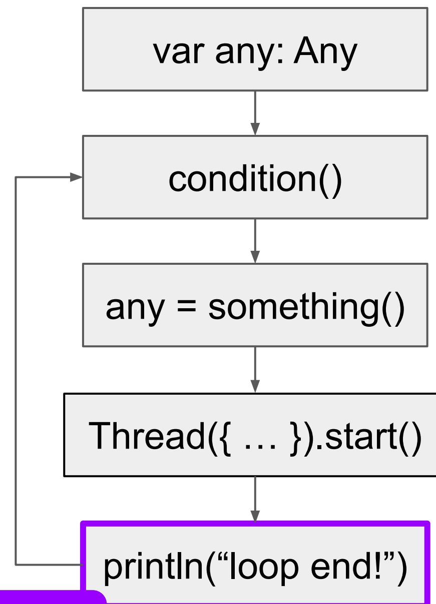
Backwards edges + capturing closures feature interaction

```
fun something(): String {/*...*/}

var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```



Backwards edges + capturing closures feature interaction

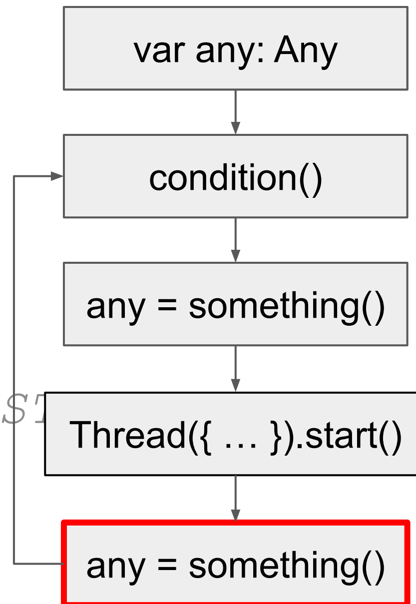
```
fun something(): String {/*...*/}

var any: Any
while (condition()) {
    any = something()
    Thread({ any.length })
        .start()
    println("loop end!")
}
```



Backwards edges + capturing closures feature interaction

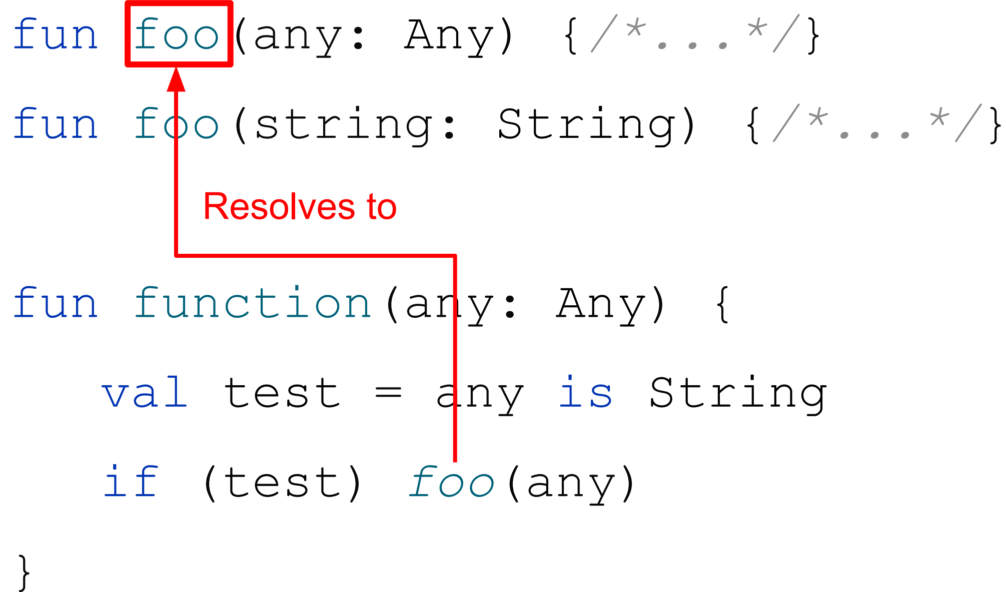
```
fun something(): String {/*...*/}

var any: Any
while (condition()) {
    any = something()
    Thread({ any.length }) // error: SMARTCAST
        .start()
    any = something()
}
```

Mutation :(



Why isn’t flow-sensitive typing (FST) the norm?

Technically, inferring a more precise type is a breaking change



Kotlin 1.9

Technically, inferring a more precise type is a breaking change

```
fun foo(any: Any) {/*...*/}
fun foo(string: String) {/*...*/}

fun function(any: Any) {
    val test = any is String
    if (test) foo(any) // Smart-cast
}
```

Resolves to

Kotlin 2.0

Technically, inferring a more precise type is a breaking change

```
fun foo(any: Any) {/*...*/}
fun foo(string: String) {/*...*/}

fun function(any: Any) {
    val test = any is String
    if (test) foo(any) // Smart-cast
}
```

Resolves to

Kotlin 2.0

Two considerations:

1. Overloads do essentially the same thing
2. FST algorithm can be frozen in time, in the language specification (not Kotlin way)

Programming languages break Liskov Substitution Principle (LSP)!

```
class Consumer<T>(val t: T) {
    fun consume(t: T) {/*...*/}
}
fun function(any: Any) {
    Consumer(any).consume(1)
    if (any is String)
        // error: incompatible types
        Consumer(any).consume(1)
}
```

Programming languages break Liskov Substitution Principle (LSP)! Java too :)

```
class Consumer<T> {
    Consumer(T t) {}
    void consume(T t) {/*...*/}
    static void function(Object any) {
        new Consumer<>(any).consume(1);
        // error: incompatible types
        new Consumer<>((String)any).consume(1);
    }
}
```

That’s it. Compilers are fun!

- How does CFG for try-catch-finally look like?
  - (consider cases when symbol types are changed in try, and exceptions and thrown)
- Kotlin specification:
  - https://kotlinlang.org/spec/type-system.html
  - https://kotlinlang.org/spec/control--and-data-flow-analysis.html
  - https://kotlinlang.org/spec/type-inference.html
- Kotlin contracts (Inter functional Control-Flow Analysis)