# Type Inference

```
1  fun getNullableInt(): Int? {
2      return 42
3  }
4
5  fun smartCast() {
6      var x = getNullableInt()
7
8      if (x != null)
9          x.inc()
10 }
```

# Type Inference

```
1  fun getNullableInt(): Int? {
2      return 42
3  }
4
5  fun smartCast() {
6      var x = getNullableInt()
7
8      if (x != null)
9          x.inc()
10 }
```

# Type Inference

```
1  fun getNullableInt(): Int? {
2      return 42
3  }
4
5  fun smartCast() {
6      var x = getNullableInt()
7      run { x = null }
8      if (x != null)
9          x.inc()
10 }
```

# Type Inference

```
1  fun getNullableInt(): Int? {
2      return 42
3  }
4
5  fun smartCast() {
6      var x = getNullableInt()
7      run { x = null }
8      if (x != null)
9          x.inc()
10 }
```

# Type Inference

```
1 val a: Any? = TODO()
2 val b = a
3 if (b is Int) {
4     a.inc()
5 }
```

# Type Inference

```
1 val a: Any? = TODO()
2 val b = a
3 if (b is Int) {
4     a.inc()
5 }
```

# Type Inference

- Outline the *Type Inference* of Kotlin in general
- Focus especially on *Smart Casts* in more detail
- Relate type inference in Kotlin to Hindely-Milner type system and others
- Related to *Type Constraint System*

# Subtyping

```
1  fun <T> mk() : T = TODO()
2
3  class Foo<A, B : A?> {
4      val b: B = mk()
5      val bQ: B? = mk()
6      val ab: A = b
7      val abQ: A = bQ
8      val aQb: A? = b
9      val aQbQ: A? = bQ
10 }
```

# Subtyping

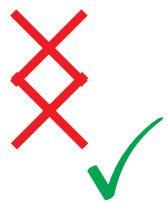
```
1  fun <T> mk() : T = TODO()
2
3  class Foo<A, B : A?> {
4      val b: B = mk()
5      val bQ: B? = mk()
6      val ab: A = b
7      val abQ: A = bQ
8      val aQb: A? = b
9      val aQbQ: A? = bQ
10 }
```

# Subtyping

```
1  fun <T> mk() : T = TODO()
2
3  class Foo<A, B : A?> {
4      val b: B = mk()
5      val bQ: B? = mk()
6      val ab: A = b
7      val abQ: A = bQ
8      val aQb: A? = b
9      val aQbQ: A? = bQ
10 }
```

# Subtyping

```
1  fun <T> mk() : T = TODO()
2
3  class Foo<A, B : A?> {
4      val b: B = mk()
5      val bQ: B? = mk()
6      val ab: A = b
7      val abQ: A = bQ
8      val aQb: A? = b
9      val aQbQ: A? = bQ
10 }
```



# Subtyping

```
1  fun <T> mk() : T = TODO()
2
3  class Foo<A, B : A?> {
4      val b: B = mk()
5      val bQ: B? = mk()
6      val ab: A = b
7      val abQ: A = bQ
8      val aQb: A? = b
9      val aQbQ: A? = bQ
10 }
```

# Subtyping

- Outline the *Subtyping* relation important for the type system
- Describe the relation to *Type Containment*
- Detail the function of *Type Decaying*, *Union-* and *Intersection-Types*
- Present the significance of *Integer Literal Types*
- Related to other type system related topics

# Type Parameters

```
1 interface Consumer<A>
2 interface Producer<A>
3
4 var numConsumer: Consumer<Number>? = TODO()
5 var intConsumer: Consumer<Int>? = numConsumer
6
7 var intProducer: Producer<Int>? = TODO()
8 var numProducer: Producer<Number>? = intProducer
```

# Type Parameters

```
1 interface Consumer<A>
2 interface Producer<A>
3
4 var numConsumer: Consumer<Number>? = TODO()
5 var intConsumer: Consumer<Int>? = numConsumer
6
7 var intProducer: Producer<Int>? = TODO()
8 var numProducer: Producer<Number>? = intProducer
```

# Type Parameters

```
1 interface Consumer<in A>
2 interface Producer<A>
3
4 var numConsumer: Consumer<Number>? = TODO()
5 var intConsumer: Consumer<Int>? = numConsumer
6
7 var intProducer: Producer<Int>? = TODO()
8 var numProducer: Producer<Number>? = intProducer
```

# Type Parameters

```
1 interface Consumer<in A>
2 interface Producer<out A>
3
4 var numConsumer: Consumer<Number>? = TODO()
5 var intConsumer: Consumer<Int>? = numConsumer
6
7 var intProducer: Producer<Int>? = TODO()
8 var numProducer: Producer<Number>? = intProducer
```

# Type Parameters

- Outline the relation of *in-/co-/contravariant types* to subtyping
- Describe *Mixed-Site-Variance* in this context
- Detail *Type Capturing*
- Related to *Type Inference* and *Subtyping*

# Null Safety

```
1  fun main() {
2      val x: Int
3      var y: Int
4      if (cond) {
5          x = 40
6          y = 4
7      } else {
8          x = 20
9      }
10     y = 5
11     val z = x + y
12 }
```

# Null Safety

```
1  fun main() {
2      val x: Int
3      var y: Int
4      if (cond) {
5          x = 40
6          y = 4
7      } else {
8          x = 20
9      }
10     y = 5
11     val z = x + y
12 }
```

# Null Safety

```
1  fun main() {
2      val x: Int
3      var y: Int
4      while (cond) {
5          x = 40
6          y = 4
7      }
8
9
10
11     val z = x + y
12 }
```

# Null Safety

```
1  fun main() {
2      val x: Int
3      var y: Int
4      while (cond) {
5          x = 40
6          y = 4
7      }
8
9
10
11     val z = x + y
12 }
```

# Null Safety

```
1  fun main() {
2      val x: Int
3      var y: Int
4      while (cond) {
5          x = 40
6          y = 4
7      }
8
9
10
11     val z = x + y
12 }
```

# Null Safety

- Outline the function of *Variable Initialisation Analysis*
- Explain the relation between *Nullable* and *Non-nullable Types*
- Describe approaches to handle nullable Types, e. g. *Elvis Operator*
- Related to *Type Inference*, esp. *Smart-Casts*

# Contracts

```
1  fun Any?.isValidString(): Boolean {
2
3
4
5      return this != null && this is String && this.length > 0
6  }
7
8  fun getString() : String? {
9      // Somehow get the string, which might be null.
10 }
11
12 fun testString() {
13     val test = getString()
14
15     if (test.isValidString()) {
16         // Does not compile:
17         // Type mismatch. Required: String. Found: String?.
18         val result: String = test
19     }
20 }
```

In the example:

- nullability checks and cast checks performed in `isValidString`
- however, this is not propagated to call site

# Contracts

```
1  @ExperimentalContracts
2  fun Any?.isValidString(): Boolean {
3      contract {
4          returns(true) implies (this@getString is String)
5      }
6      return this != null && this is String && this.length > 0
7  }
8
9  fun getString() : String? {
10     // Somehow get the string, which might be null.
11 }
12
13 fun testString() {
14     val test = getString()
15
16     if (test.isValidString()) {
17
18         // compiles now
19         val result: String = test
20     }
21 }
```

## In the example:

- nullability checks and cast checks performed in `isValidString`
- however, this is not propagated to call site

Contracts

- give additional information to e.g. function and lambda calls
- help with smart casts

$\rightarrow$ verification of contracts at declaration site?

## Topic

- assess contract capabilities
- how about verification?
- details on how are they used?
- details on how are they implemented, esp. on JVM?

# Coroutines

```
1  var c: Continuation<Unit>? = null
2
3  suspend fun suspendMe() = suspendCoroutine<Unit> { continuation ->
4      println("Suspended")
5      c = continuation
6  }
7
8
9
10
11
12
13
14 fun main() {
15   val lambda: suspend () -> Unit = {
16       suspendMe()
17       println(1)
18       suspendMe()
19       println(2)
20   }
21
22   lambda()
23
24
25
26 }
```

In the example:

- first compiled to CPS
- then implemented via one-shot continuations
- Continuation is exposed via API

# Coroutines

```
1  var c: Continuation<Unit>? = null
2
3  suspend fun suspendMe() = suspendCoroutine<Unit> { continuation ->
4      println("Suspended")
5      c = continuation
6  }
7  fun builder(c: suspend () -> Unit) {
8      c.startCoroutine(object: Continuation<Unit> {
9          override val context = EmptyCoroutineContext
10         override fun resumeWith(result: Result<Unit>) {
11             result.getOrThrow()
12         }
13 })}
14 fun main() {
15   val lambda: suspend () -> Unit = {
16       suspendMe()
17       println(1)
18       suspendMe()
19       println(2)
20   }
21   builder {
22       lambda()
23   }
24   c?.resume(Unit)
25   c?.resume(Unit)
26 }
```

## In the example:

- first compiled to CPS
- then implemented via one-shot continuations
- Continuation is exposed via API

## Topic

- assess capabilities of Coroutines
- shed light on the implemenatation concept
- highlight implementation consequences on JVM

# Overload Resolution

```
1  interface Y
2
3  class X : Y {
4    fun Y.foo() {} // `foo` is an extension for Y,
5    // needs extension receiver to be called
6    fun bar() {
7      foo() // `this` reference is both
8      // the extension and the dispatch receiver
9    }
10 }
11
12 fun main() {
13   val x: X = mk()
14   val y: Y = mk()
15   // y.foo()
16   // Error, as there is no implicit receiver
17   // of type X available
18   with (x) {
19     y.foo() // OK!
20   }
21 }
```

In the example:

- E.g. Extension Methods complicate *Overload Candidate Set*

# Overload Resolution

```
1 fun foo(a: Foo, b: Bar) {
2   (a + b)(42)
3   // Such a call is handled as if it is
4   // (a + b).invoke(42)
5 }
```

In the example:

- E.g. Extension Methods complicate *Overload Candidate Set*
- Operators / Infix Receivers

# Overload Resolution

```
1  fun f(arg: Int, arg2: String) {} // (1)
2  fun f(arg: Any?, arg2: CharSequence) {} // (2)
3  ...
4  f(2, "Hello")
5
6  fun f1(arg: Int, arg2: String) {
7    f2(arg, arg2) // VALID: can forward both arguments
8  }
9  fun f2(arg: Any?, arg2: CharSequence) {
10   f1(arg, arg2) // INVALID: function f1 is not applicable
11 }
```

In the example:

- E.g. Extension Methods complicate *Overload Candidate Set*
- Operators / Infix Receivers
- Concept of *Most Specific Candidate* refined

# Overload Resolution

```kotlin
@OverloadResolutionByLambdaReturnType
fun foo(cb: (Unit) -> String) = Unit // (1)

@OverloadResolutionByLambdaReturnType
fun foo(cb: (Unit) -> Int) = Unit // (2)

fun testOk01() {
  foo { 42 }
  // Both (1) and (2) are applicable
  // (2) is preferred by the lambda return type
}
```

## In the example:

- E.g. Extension Methods complicate *Overload Candidate Set*
- Operators / Infix Receivers
- Concept of *Most Specific Candidate* refined
- *Lambda return types* also affect the MRO

## Topic

- explore Kotlin extension Methods
- elaborate on *Overload Candidate Set* Determination
- elaborate how to pick the *Most Specific Candidate*

# Type Constraints

For each call, we determine function applicability via one ofthe following constraint systems:

- For every non-lambda argument inferred to have type $T_i$, corresponding to the function parameter of type $U_j$ , a constraint $T_i \leq U_j$ is constructed
- All declaration-site type constraints for the function are also added to the constraint system
- For every lambda argument with the number of lambda arguments known to be K, corresponding to the function parameter of type $U_m$, a special constraint of the form $(FT(L_1, \ldots, L_K) \rightarrow R \ \& \ FTR(RT, L_1, \ldots, L_n) \rightarrow R) \leq U_m$ is added to the constraint system, where $R$, $RT$, $L_1, \ldots, L_K$ are fresh type variables
- For each lambda argument with an unknown number of lambda arguments (that is, being equal to 0 or 1), corresponding to the function parameter of type $U_n$, a special constraint of the form $(FT() \rightarrow R \ \& \ FT(L) \rightarrow R \ \& \ FTR(RT) \rightarrow R \ \& \ FTR(RT, L) \rightarrow R) \leq U_m$ is added to the constraint system, where $R$, $RT$, $L$ are fresh type variables

## In the example:

Type constraints stem from several applications:

- Applicability of a function during collection of *Overload Candidate Sets*

# Type Constraints

During MSC selection, for every two distinct members of the candidate set F1 and F2, the following constraint system is constructed and solved:

- For every non-default argument of the call and their corresponding declaration-site parameter types $X_1, \ldots, X_N$ of F1 and $Y_1, \ldots, Y_N$ of F2, a type constraint $X_K \leq Y_K$ is built unless both $X_K$ and $Y_K$ are built-in integer types. If both XK and YK are built-in integer types, a type constraint $\mathit{Widen}(X_K) \leq \mathit{Widen}(Y_K)$ is built instead, where Widen is the integer type widening operator. During construction of these constraints, all declaration-site type parameters $T_1, \ldots, T_M$ of F1 are considered bound to fresh type variables $T_{\tilde{1}}, \ldots, T_{\tilde{M}}$, and all type parameters of F2 are considered free;
- If F1 and F2 are extension callables, their extension receivers are also considered non-default arguments of the call, even if implicit, and the corresponding constraints are added to the constraint system as stated above. For non-extension callables, only declaration-site parameters are considered;
- All declaration-site type constraints of $X_1, \ldots, X_N$ and $Y_1, \ldots, Y_N$ are also added to the constraint system

## In the example:

Type constraints stem from several applications:

- Applicability of a function during collection of *Overload Candidate Sets*
- comparing two candidate function signatures during determination of *Most Specific Candidate*

# Type Constraints

During MSC selection, for every two distinct members of the candidate set F1 and F2, the following constraint system is constructed and solved:

- For every non-default argument of the call and their corresponding declaration-site parameter types $X_1, \ldots, X_N$ of F1 and $Y_1, \ldots, Y_N$ of F2, a type constraint $X_K \leq Y_K$ is built unless both $X_K$ and $Y_K$ are built-in integer types. If both XK and YK are built-in integer types, a type constraint $\mathit{Widen}(X_K) \leq \mathit{Widen}(Y_K)$ is built instead, where Widen is the integer type widening operator. During construction of these constraints, all declaration-site type parameters $T_1, \ldots, T_M$ of F1 are considered bound to fresh type variables $T_{\tilde{1}}, \ldots, T_{\tilde{M}}$, and all type parameters of F2 are considered free;
- If F1 and F2 are extension callables, their extension receivers are also considered non-default arguments of the call, even if implicit, and the corresponding constraints are added to the constraint system as stated above. For non-extension callables, only declaration-site parameters are considered;
- All declaration-site type constraints of $X_1, \ldots, X_N$ and $Y_1, \ldots, Y_N$ are also added to the constraint system

## In the example:

Type constraints stem from several applications:

- Applicability of a function during collection of *Overload Candidate Sets*
- comparing two candidate function signatures during determination of *Most Specific Candidate*

## Topic

- give examples for noteworthy applications of constraint solving
- assess the expressivity of constraints, as well as solvable classes
- point out how constraint solving is implemented in Kotlin

# Topic Selection

1. Type Inference
2. Subtyping
3. Type Parameters
4. Null Safety
5. Contracts
6. Coroutines
7. Overload Resolution
8. Type Constraints

## Proceeding

In order to vote for your preferred topics:

1. Sort the topics in the order that you prefer them,
2. assign them numbers, with your most prefered topic #1 and the least preferred topic #8
3. Send an email with your ranking to petter@in.tum.de until Tue April 11th 23:59