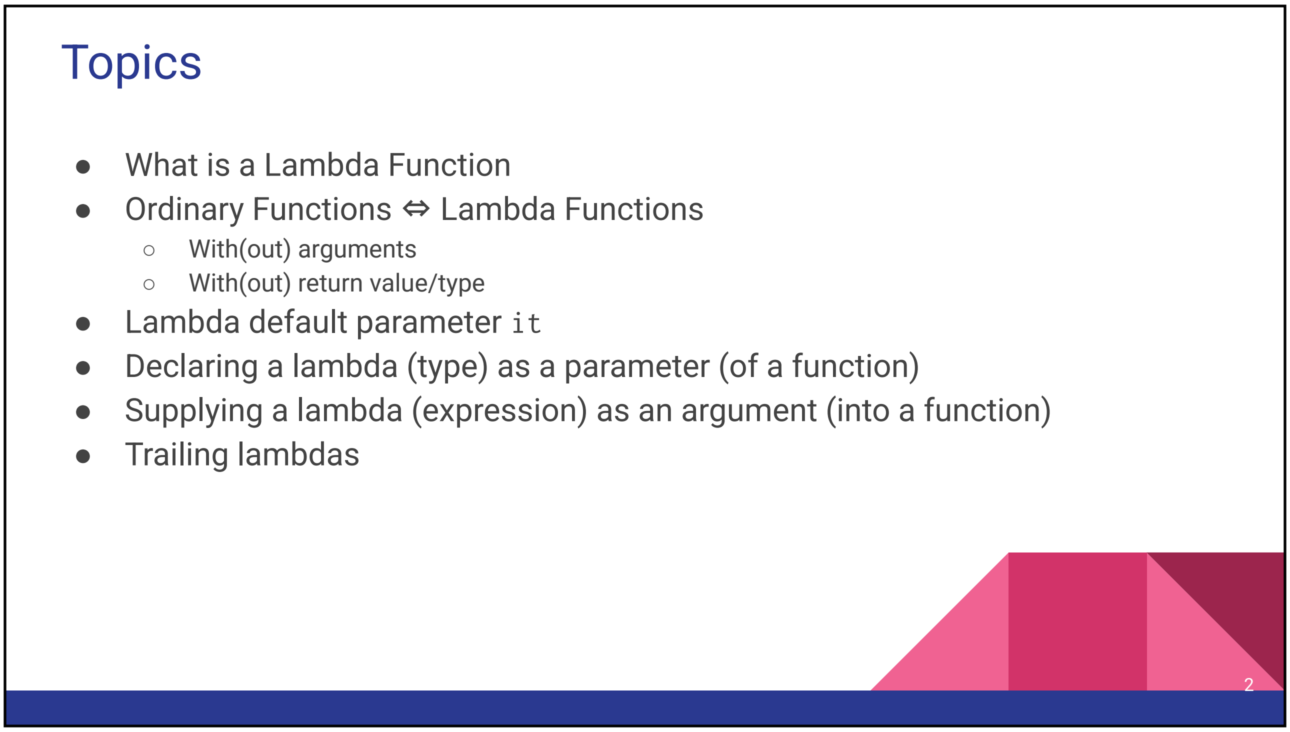
Kotlin Lambda Functions

Kotlin



Topics

- What is a Lambda Function
- Ordinary Functions ⇔ Lambda Functions
  - With(out) arguments
  - With(out) return value/type
- Lambda default parameter `it`
- Declaring a lambda (type) as a parameter (of a function)
- Supplying a lambda (expression) as an argument (into a function)
- Trailing lambdas

Lambda Functions

- Short definition: lambda functions are anonymous functions
  - Function-like expressions with no function name
  - The expression defines argument name and types, body of code
- Practical use
  - Extensive use by Kotlin standard library
  - Manipulate collections using functional approach
  - Jetpack Compose

Expressions: Arithmetic vs. Lambda

	Arithmetic	Lambda
Has a type	✔	✔
Can be saved to a variable	✔	✔
Can be invoked	❌	✔
Typical Operators	+, -, *, /, %, ...	"Arrow" and curly braces
Evaluates to	a value ("data")	The expression itself evaluates to a function ("code") When invoked, the function evaluates to a value ("data")

Name, Type & Value of a Variable

Declaration	Name	Type	Value
`val a = “50”`	a	String	Five Zero
`val b = 50.0`	b	Double	Fifty
`var c: Float? = null`	c	Maybe Float	No data

Name, Type & Value of a Function?

Function Signature	**Value**	**Type**
`fun go() { }`		`() -> Unit`
`fun hello(n: String?) { }`		
`fun isPrime(n: Int): Boolean { }`	`Boolean`	
`fun add(a:Int, b:Int): Int { }`		
`fun mult(a:Int, b:Int): Int { }`		
`fun commonLetters(a: String, b: String): Int`		
`fun longestPrefix(a: String, b: String): Int`		

Ordinary Function ⇒ Lambda (Case 1: No params)

```
fun hello1() {
  println(“Hello World”)
}
```

```
val hello1: () -> Unit = {
  println(“Hello World”)
}
```

```
val hello1 = { -> Unit
  println(“Hello World”)
}
```

```
val hello1: = { ->
  println(“Hello World”)
}
```

```
val hello1: = {
  println(“Hello World”)
}
```

Ordinary Function ⇒ Lambda (Case 2: With param)

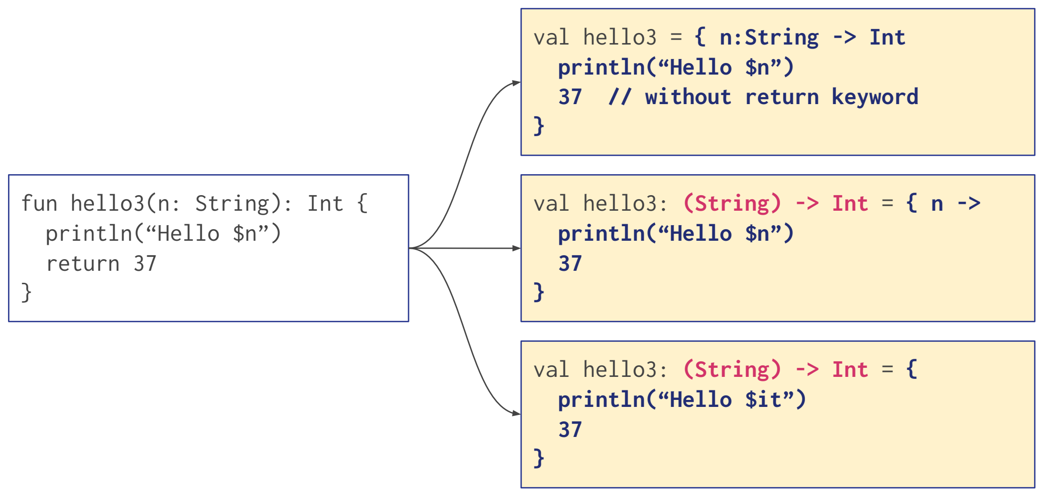
```
fun hello2(n: String) {
  println(“Hello $n”)
}
```

```
val hello2 = { n:String -> Unit
  println(“Hello $n”)
}
```

```
val hello2: (String) -> Unit = { n ->
  println(“Hello $n”)
}
```

```
// With implicit name of a SINGLE param
val hello2: (String) -> Unit = {
  println(“Hello $it”)
}
```

Ordinary Function ⇒ Lambda (Case 3: With Return)



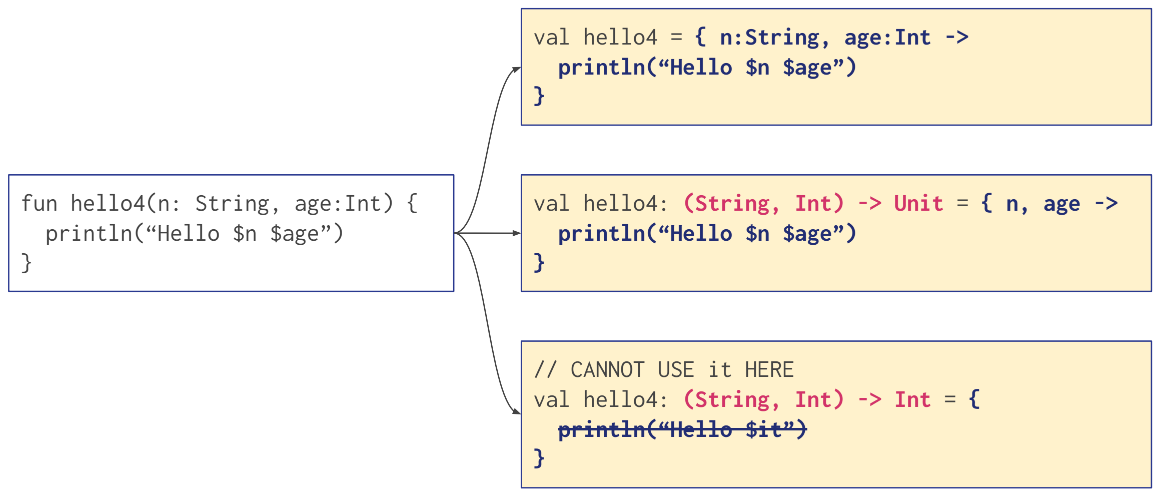
```
fun hello3(n: String): Int {
  println(“Hello $n”)
  return 37
}
```

```
val hello3 = { n:String -> Int
  println(“Hello $n”)
  37  // without return keyword
}
```

```
val hello3: (String) -> Int = { n ->
  println(“Hello $n”)
  37
}
```

```
val hello3: (String) -> Int = {
  println(“Hello $it”)
  37
}
```

Ordinary Function ⇒ Lambda (Case 4: More Params)



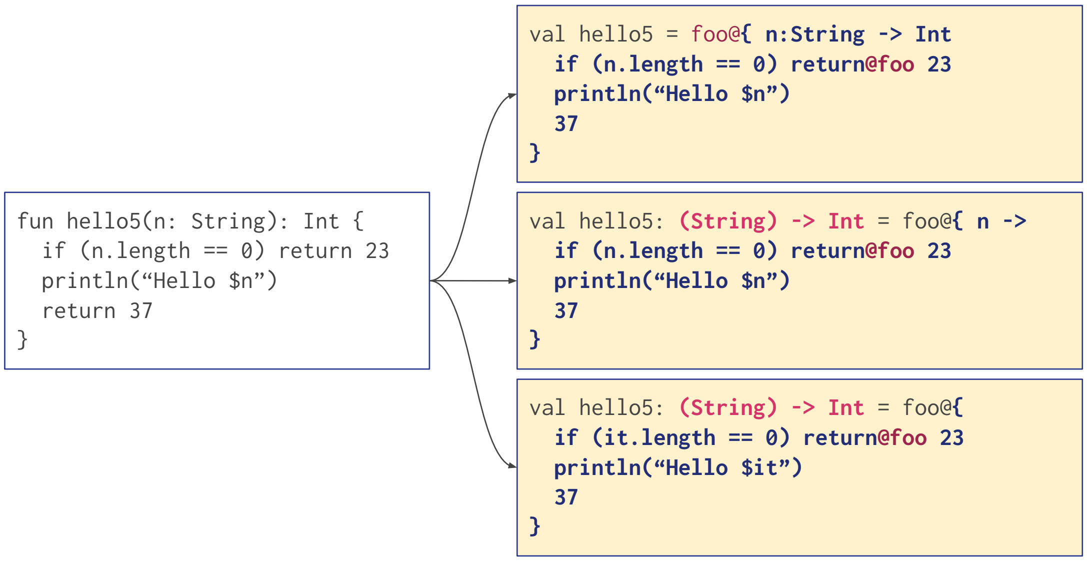
```
fun hello4(n: String, age:Int) {
  println(“Hello $n $age”)
}
```

```
val hello4 = { n:String, age:Int ->
  println(“Hello $n $age”)
}
```

```
val hello4: (String, Int) -> Unit = { n, age ->
  println(“Hello $n $age”)
}
```

```
// CANNOT USE it HERE
val hello4: (String, Int) -> Int = {
  ~~println(“Hello $it”)~~
}
```

Ordinary Function ⇒ Lambda (Case 5: Early Return)



```
fun hello5(n: String): Int {
  if (n.length == 0) return 23
  println("Hello $n")
  return 37
}
```

```
val hello5 = foo@{ n:String -> Int
  if (n.length == 0) return@foo 23
  println("Hello $n")
  37
}
```

```
val hello5: (String) -> Int = foo@{ n ->
  if (n.length == 0) return@foo 23
  println("Hello $n")
  37
}
```

```
val hello5: (String) -> Int = foo@{
  if (it.length == 0) return@foo 23
  println("Hello $it")
  37
}
```

Passing/Supplying Arguments to a Function

- All programming languages allow passing data as an argument to function
- NOT all programming languages allow passing code/function to a function

Passing Data As Arguments (into a Function)

```
fun payTax (acctBalance: Float, taxRate: Float) { }
```

Supply data via a variable	Supply data “in place” (**immediate value**)

`val myMoney = 20_000`
`val cityTax = 0.05`
`payTax(myMoney, cityTax)`
	`payTax(20_000, 0.05)`

Passing Code As Arguments (into a Function)

```
fun payDividend (acctBalance: Float, bonus: (Int, Float) -> Float) {   }
```

```
fun newYearBonus(yrsOfSrvc: Int, totSales: Float): Float {
  val percentage = if (yrsOfSrvc > 10) 0.08 else 0.03
  return percentage * totSales
}

val myMoney = 20_000
payDividend (myMoney, ::newYearBonus) // Double :: and no parentheses
```

```
// Supply args via immediate values
payDividend (20_000, { yrsOfSrvc, totSales -> Float
  val percentage = if (yrsOfSrvc > 10) 0.08 else 0.03
  percentage * totSales
})
```

Trailing Lambda

Supplying Trailing Lambdas

```
fun payDividend (balance: Float, bonus: (Int) -> Float) {
  val extraPay = bonus(yearsOfService)
  println (“Dividend this year ${bonus + extraPay}”)
}
```

```
// Lambda expression (inside parentheses)
payDividend (20_000, { years ->
  if (years > 10) 500 * years else 200 * years
})
```

```
// Lambda expression (outside parentheses)
payDividend (20_000) { years ->
  if (years > 10) 500 * years else 200 * years
}
```

Supplying Trailing Lambdas

```
fun payDividend (percentage: Float = 0.05, bonus: (Int) -> Float) {
  val extraPay = bonus(yearsOfService)
  println ("Dividend this year ${bonus + extraPay}")
}
```

```
// Default 0.05 on first parameter, Lambda (outside parentheses)
payDividend () { years ->
  if (years > 10) 500 * years else 200 * years
}
```

```
// Empty parentheses removed
payDividend { years ->
  if (years > 10) 500 * years else 200 * years
}
```

Jetpack Compose Example(s)

Example #1: Using Button (@Composable)

```
@Composable
fun Button(
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    shape: Shape = ButtonDefaults.shape,
    border: BorderStroke? = null,
    content: @Composable RowScope.() -> Unit
) {
  /* Code Not Shown */
}
```

trailing lambda

```
Button(
   onClick = { println("Hi") },
   enabled = false,
   content = { Text("Try me") }
)
```

```
Button(onClick = { println("Hi") }) {
     Text("Try me")
}
```

Example #2: Deeper Nesting of @Composable

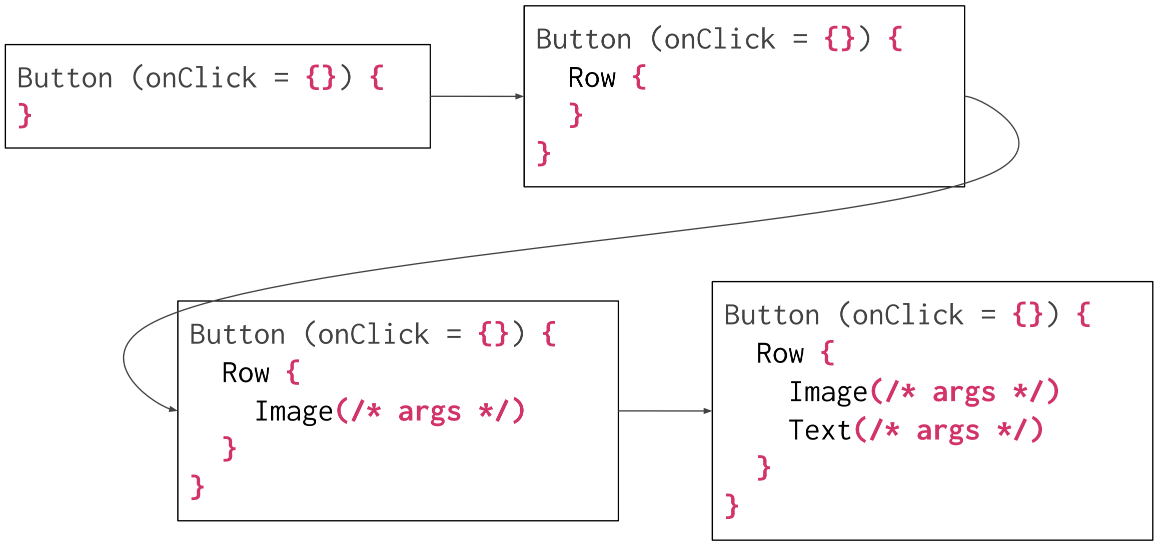
```
@Composable
fun Button(
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    shape: Shape = ButtonDefaults.shape,
    border: BorderStroke? = null,
    content: @Composable RowScope.() -> Unit
) {
  /* Code Not Shown */
}
```

```
@Composable
fun Row(
    modifier: Modifier = Modifier,
    // more parameters hidden here
    content: @Composable RowScope.() -> Unit
) {
  /* code not shown */
}
```

```
Button(onClick = { println("Hi") }) {
  Row (content = {
    Icon(/* some args */)
    Text("Try me")
  })
}
```

```
Button(onClick = { println("Hi") }) {
  Row {
    Icon(/* some args */)
    Text("Try me")
  }
}
```

Build Your Code Incrementally by pairs of () or {}



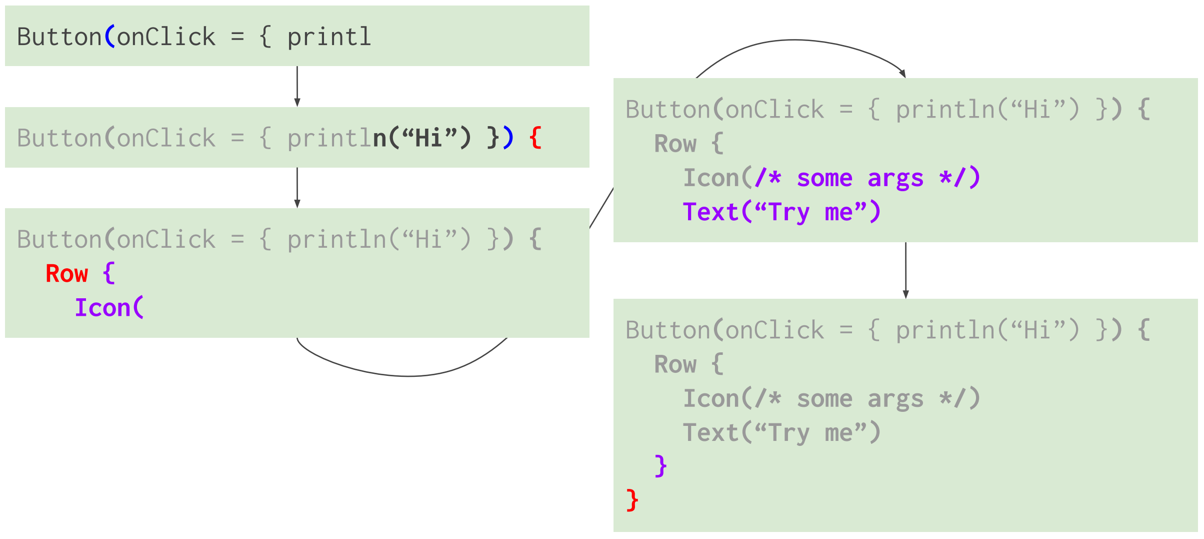
```
Button (onClick = {}) {
}
```

```
Button (onClick = {}) {
  Row {
  }
}
```

```
Button (onClick = {}) {
  Row {
    Image(/* args */)
  }
}
```

```
Button (onClick = {}) {
  Row {
    Image(/* args */)
    Text(/* args */)
  }
}
```

DO NOT develop your code *sequentially*!!!



```
Button(onClick = { printl
```

```
Button(onClick = { println("Hi") }) {
```

```
Button(onClick = { println("Hi") }) {
  Row {
    Icon(
```

```
Button(onClick = { println("Hi") }) {
  Row {
    Icon(/* some args */)
    Text("Try me")
```

```
Button(onClick = { println("Hi") }) {
  Row {
    Icon(/* some args */)
    Text("Try me")
  }
}
```

Lambda & Collections (Optional)

```
data class Atom(val name: String,
                val weight: Float)

fun main(): Unit {
  val atoms = listOf(
    Atom(name = "Carbon", weight=12.011f),
    Atom(name = "Oxygen", weight=15.999f),
    Atom(name = "Hydrogen", weight=1.008f),
    Atom(name = "Sodium", weight= 22.990f),
    Atom(name = "Copper", weight= 63.546f)
  )

  val byWeight = atoms.sortedBy { x -> x.weight }
  val byWeight = atoms.sortedBy { it.weight }

  for (a in byWeight) {
    println(a)
  }
}
```

Lambda expression