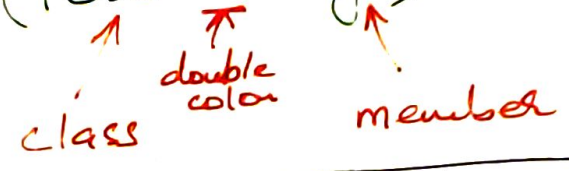
Member References:

→ Like Java, Kotlin has member references which can replace simple lambdas that just call the member function or return a member property.

Example:

```
class Person(val name: String, val age: Int)
people.maxBy { it.age }
people.maxBy (Person :: age)
```



→ In the example above, First goes the class name, then double colon, then the member which we refer.

→ IDE can even convert lambda to reference automatically when its possible.

Storing Lambda in a Variable.

→ You can store a lambda in a variable

→ However you cannot store a function in a variable.

- If you try to assign a function to a variable, youll get a compiler error

- Using a function reference syntax however fixes this issue.
  - Function References allow you to store a reference to any defined function inside a variable.

e.g: Lambda in a variable

```
val isEven = (Int) -> Boolean = {
        i : Int -> i %2 == 0
}
```

But you cant store a function in a variable.

e.g:

```
fun isEven(i: Int) : Boolean = i %2 == 0.
```

```
val predicate = isEven
```

Compiler Error.

However,

```
val predicate = :: isEven
```

double colon. function

val predicate = :: isEven

internally equivalent to the following,

val predicate = { i:Int -> isEven(i) }

A lambda just calling the isEven function.

→ Another example

```
val action = { person: Person, message: String
                -> sendEmail (person, message)
             } // This is a bit of a verbose syntax.
```

Concised syntax given below,

```
val action = :: sendEmail.
// Member reference allows you to hide
// all implementation because the compiler
// infers the type for you.
```

Passing Function Reference as an Argument:

```
fun isEven i:Int): Boolean = i % 2 == 0
```

→ You can pass the above function reference as an argument.

e.g:

```
val list = listOf(1, 2, 3, 4)
list.any(::isEven)          // true
list.filter(::isEven)       // returns [2, 4]
```

Bound and Non Bound References:

```
class Person(val name: String, val age: Int) {
    fun isOlder(ageLimit: Int) = age > ageLimit
}

val agePredicate = Person::isOlder   // Regular Nonbound reference
val alice = Person("Alice", 29)
agePredicate(alice, 21)              // true
```

→ Passed the reference of Person::isOlder to agePredicate variable.

→ Reusing it by now passing an object of class Person and an age limit.

→ If we look at the function type this member reference has,

```
val agePredicate = Person::isOlder.
```

is equivalent to,

```
val agePredicate: (Person, Int) -> Boolean =
                                    Person::isOlder
// i.e First argument of this function type
// is person.
// When ever we want to call this variable
// of function type, we to parse this
// Person instance explicitly need
```

→ Internally implementation of,

```
Person::isOlder
```

is equivalent to,

```
{ person, ageLimit ->
        person.isOlder(ageLimit) }
// Internally the member function is being
// called inside.
```

How to call Non Bound Reference Variable?

```
val alice = Person("Alice", 29)
agePredicate(alice, 21)   //true
```

alice → Object of type Person

21 → Age

//true → Returns a Boolean.

Bound Reference:

→ To make a Bound Reference, you need to call a member of specific instance of a class.

Example:

```
class Person (val name: String, val age: Int) {
    fun isOlder(ageLimit: Int) = age > ageLimit
}
val alice = Person("Alice", 29)
val agePredicate = alice :: isOlder
agePredicate(21)  // true.
```

alice is an object of a Person class.

When this agePredicate is called, it will be called on this specific instance (alice).

Internal Implementation Equivalency:

val agePredicate = alice :: isOlder

internally has a functin type,

```
val agePredicate: (Int) -> Boolean = alice :: isOlder
```

(Int) → Argument.

Boolean → Return Type

Another example of Bound Reference.

```
class Person (val name: String, val age: Int) {
    fun isOlder(ageLimit: Int) = age > ageLimit
    fun getAgePredicate() = ::isOlder
}
```

↳ Equivalent to,

```
this::isOlder
```

// Making it a Bound Reference.

Function Type of this function is,

(Int) → Boolean

↓

Function Type.

Example of Non Bound Reference:

```
fun isEven(i: Int): Boolean = i % 2 == 0
val list = listOf(1, 2, 3, 4)
list.any(::isEven)
list.filter(::isEven)
```

↳ This is just a reference to a top level function. Making it a Non Bound Reference.

Return from Lambda :

→ What does return expression inside a lambda does? Why it works like that? And how to safely use it for your own purposes.?

→ Functions can be nested in Kotlin.

→ Qualified returns allow us to return from an outer function.

↳ Most important use case is returning from a Lambda expression

Example :

```
fun foo() {
    listOf(1,2,3,4,5).forEach {
        if (it == 3) return // non-local return
        print(it)           // directly to the caller
    }                       // of foo().
    println("This point is unreachable")
}
```

→ return-expression returns from the nearest enclosing function i.e foo.

Note:

Such non local returns are only supported for Lambda expressions passed to inline functions.

→ If we need to return from a Lambda expression, we have to label it and qualify the return:

Example:

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach lit@ {
        if (it == 3) return@lit // local return
        print(it)               // to the caller of the
    }                           // lambda, i.e forEach loop.
    print("done with explicit label")
}
```

You can provide a custom Label

→ Now it returns only from the Lambda expression.

Use of Implicit Labels:

→ Its more convinient to use implicit labels. Such as the function to which the lambda is passed.

Example:

```
listOf() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return@forEach // local return
        print(it)                   // to the caller of lambda,
    }                               // i.e forEach loop.
    print("done with implicit label")
}
```

Replacing Lambda Expression with Anonymous Function:

-> return statement in an anonymous function will return from anonymous function itself.

Example:

```
fun foo() {
 listOf(1,2,3,4,5).forEach(fun(value:Int) {
    if (value == 3) return // local return
                           // to the caller of anonymous
    print(value)           // function i.e. forEach loop.
 })
 print("done with anonymous function")
}
```

> **Note**
>
> Previous three examples is similar to the use of continue in regular loops.