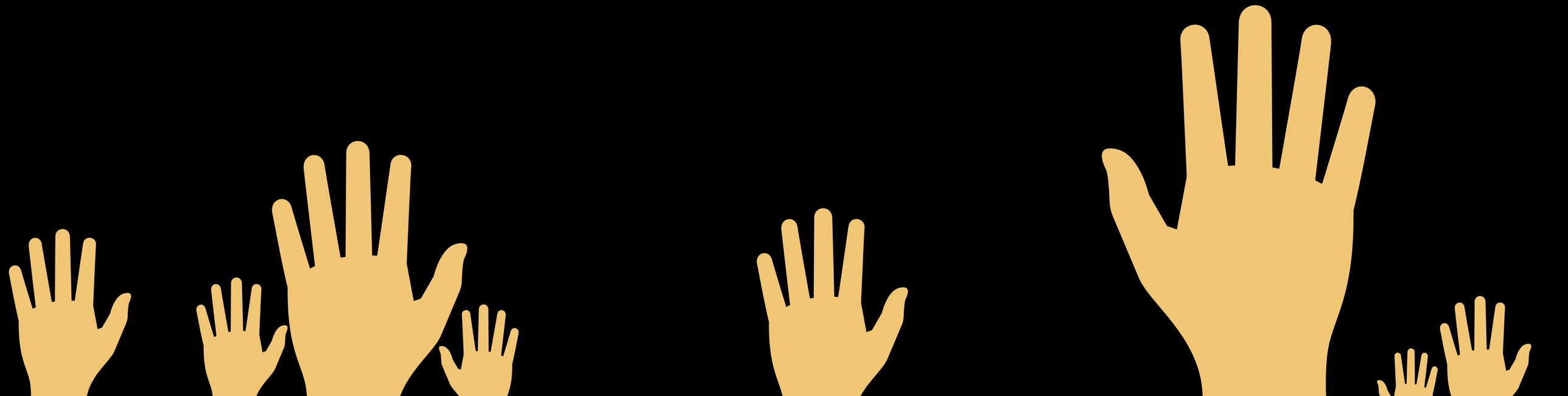


1. More readable
2. (Usually) More overhead
3. We don’t know which one we aspire to more

A clever title

Christina Lee

Two Stones, One Bird

~~A clever title~~

Christina Lee

“Kill two birds with one stone”

“Kill two birds with one stone”

==

Accomplish two objectives with a single action

Accomplish two objectives with a single action

==

GOOD!

Two for one

==

GOOD!

One for two

==

????

Two for one

==

GOOD!

One for two

==

????

ARGUE ON THE INTERNET!

WIN

ARGUEMENTS ON THE INTERNET!

1. Meme effectively

2. (Optional) Know something about the topic

1. Meme effectively ¯_(ツ)_/¯

2. (Optional) Know something about the topic

2. (Optional) Know something about the topic

Nullability

```
data class Pin(
        val id: String,
        val creator: User? = null,
      // ...
)
```

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
    if (pin.creator != null) {
        avatarView.bindUser(pin.creator)
    }
}
```

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
    if (pin.creator != null) {
        avatarView.bindUser(pin.creator)
    }
}
```

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
    if (pin.creator != null) {
        avatarView.bindUser(pin.creator!!)
    }
}
```

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
    val creator = pin.creator
    if (creator != null) {
        avatarView.bindUser(creator)
    }
}
```

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
    pin.creator?.let {
        avatarView.bindUser(it)
    }
}
```

Nullability

???

Style guide

~~Style Guide~~

2

Effective Kt

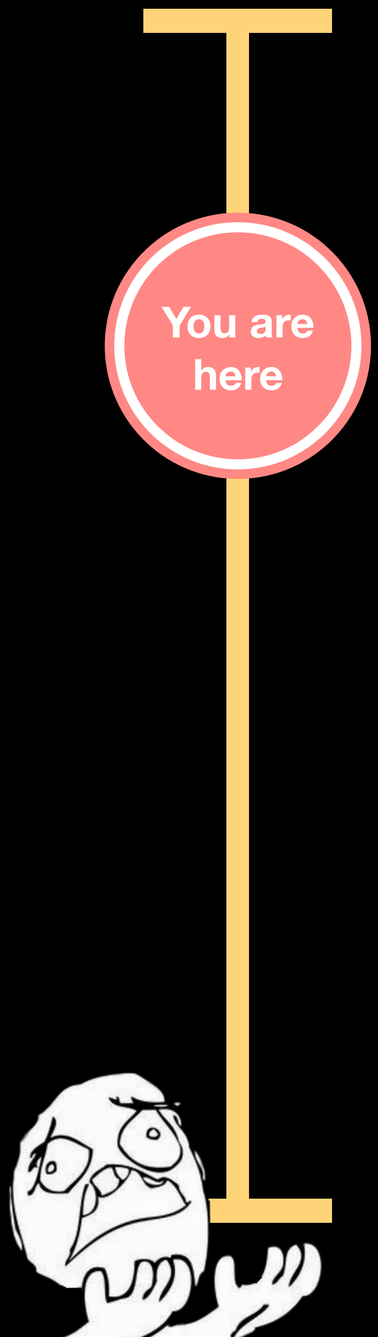
~~Effective Kt~~

Readability

Readability

Readability
+
Performance

Readability



Performance

Nullability

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
    if (pin.creator != null) {
        avatarView.bindUser(pin.creator!!)
    }
}
```

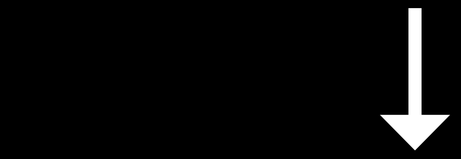
```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
   val creator = pin.creator
   if (creator != null) {
        avatarView.bindUser(creator)
    }
}
```

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
   pin.creator?.let {
        avatarView.bindUser(it)
   }
}
```

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
    val creator = pin.creator
    if (creator != null) {
        avatarView.bindUser(creator)
    }
}
```

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
    pin.creator?.let {
        avatarView.bindUser(it)
    }
}
```

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
    val creator = pin.creator
    if (creator != null) {
        avatarView.bindUser(creator)
    }
}
```



```
public static final void updateAvatar(@NotNull Pin pin, @NotNull AvatarView
avatarView) {
   Intrinsics.checkParameterIsNotNull(pin, "pin");
   Intrinsics.checkParameterIsNotNull(avatarView, "avatarView");
   User creator = pin.getCreator();
   if(creator != null) {
      avatarView.bindUser(creator);
   }
}
```

```
public static final void updateAvatar(@NotNull Pin pin, @NotNull AvatarView
avatarView) {
   Intrinsics.checkParameterIsNotNull(pin, "pin");
   Intrinsics.checkParameterIsNotNull(avatarView, "avatarView");
   User creator = pin.getCreator();
   if(creator != null) {
      avatarView.bindUser(creator);
   }
}
```

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
    val creator = pin.creator
    if (creator != null) {
         avatarView.bindUser(creator)
     }
}
```

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
    pin.creator?.let {
        avatarView.bindUser(it)
    }
}
```

```
fun updateAvatar(pin: Pin, avatarView: AvatarView) {
    pin.creator?.let {
        avatarView.bindUser(it)
    }
}
```

```
public static final void updateAvatar(@NotNull Pin pin, @NotNull AvatarView
avatarView) {
   Intrinsics.checkParameterIsNotNull(pin, "pin");
   Intrinsics.checkParameterIsNotNull(avatarView, "avatarView");
   User var10000 = pin.getCreator();
   if(var10000 != null) {
      User var2 = var10000;
      avatarView.bindUser(var2);
   }

}
```

```
public static final void updateAvatar(@NotNull Pin pin, @NotNull AvatarView
avatarView) {
   Intrinsics.checkParameterIsNotNull(pin, "pin");
   Intrinsics.checkParameterIsNotNull(avatarView, "avatarView");
   User var10000 = pin.getCreator();
   if(var10000 != null) {
      User var2 = var10000;
      avatarView.bindUser(var2);
   }

}
```

if (creator == null)

```
L1
    LINENUMBER 13 L1
    ALOAD 0  // load pin
    INVOKEVIRTUAL Pin.getCreator ()  //invoke getter
    ASTORE 2 // store in field 2
L2
    LINENUMBER 14 L2
    ALOAD 2 // load field 2 (user)
    IFNULL L3 // if user is null, jump to return
L4
    LINENUMBER 15 L4
    ALOAD 1 // load view
    ALOAD 2 // load user
    INVOKEVIRTUAL AvatarView.bindUser() //invoke
L3
    LINENUMBER 22 L3
    RETURN
```

pin.creator?.let { }

```
L1
    LINENUMBER 13 L1
    ALOAD 0      // load pin from ref 0
    INVOKEVIRTUAL Pin.getCreator()  // invoke pin.getCreator()
    DUP   // duplicate returned value
    IFNULL L2   // if null pop
    ASTORE 2  // store creator value in field 2
L3
    ALOAD 2   // load ref 2 onto stack
    ASTORE 3  // store in field 3
L4
    LINENUMBER 14 L4
    ALOAD 1 // load “avatarView”
    ALOAD 3 // load value of pin.getCreator()
    INVOKEVIRTUAL AvatarView.bindUser
L2
    POP
```

val creator = pin.creator; creator?.let { }

```
L1
    LINENUMBER 13 L1
    ALOAD 0  // load pin
    INVOKEVIRTUAL Pin.getCreator() // invoke getter
    ASTORE 2 // store result in field 2
L2
    LINENUMBER 14 L2
    ALOAD 2 // load creator
    DUP // duplicate
    IFNULL L3 // if null??
    ASTORE 3 // store in field 3
L4
    ALOAD 3 // load field 3
    ASTORE 4 // store in field 4
L5
    LINENUMBER 15 L5
    ALOAD 1 // load view
    ALOAD 4 // load user
    INVOKEVIRTUAL AvatarView.bindUser(v)  //invoke view with user
```

TL;DR:

Idk

TL;DR:
It doesn’t matter*

```
val creator = pin.creator
if (creator != null) {
    avatarView.bindUser(creator)
}
```

vs.

```
pin.creator?.let {
    avatarView.bindUser(it)
}
```

```
val creator = pin.creator
if (creator != null) {
    val firstName = creator.firstName
    if (firstName != null) {
        avatarView.setFirstName(firstName)
    }
}
```

vs.

```
pin.creator?.firstName?.let {
    avatarView.setFirstName(it)
}
```

```
val creator = pin.creator
if (creator != null) {
    val firstName = creator.firstName
    if (firstName != null) {
        avatarView.setFirstName(firstName)
    }
} else {

}
```

vs.

```
pin.creator?.firstName?.let {
    avatarView.setFirstName(it)
}.orElse {  }
```

```
val creator = pin.creator
if (creator != null && creator.firstName != null) {
    avatarView.setFirstName(firstName)
} else {

}
```

vs.

```
pin.creator?.firstName?.let {
    avatarView.setFirstName(it)
}.orElse {  }
```

Lean towards let

Lean towards let*

* Brought to you by Twitter

Lean towards let*

let

with / apply
let / also

with

with

```
public inline fun <T, R> with(
   receiver: T,
   block: T.() -> R
): R = receiver.block()
```

with

```
public inline fun <T, R> with(
   receiver: T,
   block: T.() -> R
): R = receiver.block()
```

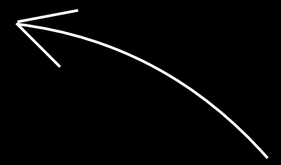
with

```
public inline fun <T, R> with(
    receiver: T,
    block: T.() -> R
): R = receiver.block()
```

```
with(foo) {
    foo.bar()
    foo.baz()
}
```

with

```
public inline fun <T, R> with(
    receiver: T,
    block: T.() -> R
): R = receiver.block()
```



```
with(foo) {
    bar()
    baz()
}
```

with

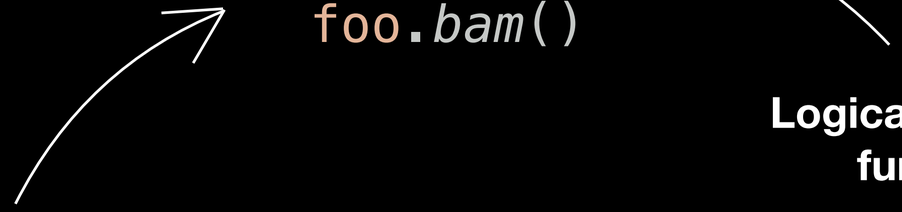
```
public inline fun <T, R> with(
   receiver: T,
   block: T.() -> R
): R = receiver.block()
```

with

```
public inline fun <T, R> with(
   receiver: T,
   block: T.() -> R
): R = receiver.block()
```

with

```
foo.bar()
foo.baz()
foo.bam()
```



Same receiver
multiple times

Logically grouped
functions

with

```
with(db) {
    open()
    commit(obj)
    close()
}
```

with

apply

apply

```
public inline fun <T> T.apply(
  block: T.() -> Unit
): T {
  block();
  return this
}
```

apply

```
public inline fun <T> T.apply(
  block: T.() -> Unit
): T {
  block();
  return this
}
```

Extension function

apply

```
public inline fun <T> T.apply(
  block: T.() -> Unit
): T {
  block();
  return this
}
```

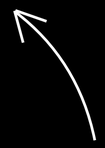
Lambda +
Receiver

apply

```
public inline fun <T> T.apply(
  block: T.() -> Unit
): T {
  block();
  return this
}
```

apply

```
public inline fun <T> T.apply(
  block: T.() -> Unit
): T {
  block();
  return this
}
```



**“I need this object, but
I need to run a bit of
code on it before I use it”**

apply

```
public inline fun <T> T.apply(
  block: T.() -> Unit
): T {
  block();
  return this
}
```

apply

```
val myTextView = TextView(context).apply {
    layoutParams = LinearLayout.LayoutParams(...)
}
```

apply

```
return foo.apply { baz() }
```

apply

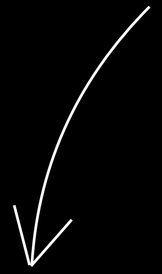
let

```kotlin
public inline fun <T, R> T.let(
  block: (T) -> R
): R = block(this)
```

let

Spoiler Alert:

Everything but
with()



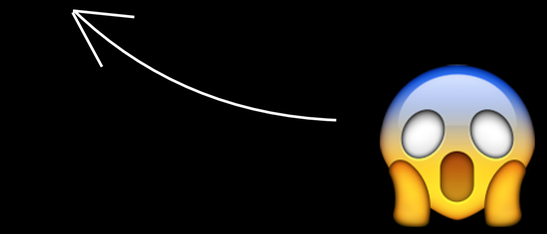
```
public inline fun <T, R> T.let(
  block: (T) -> R
): R = block(this)
```

let

```kotlin
public inline fun <T, R> T.let(
  block: (T) -> R
): R = block(this)
```

let

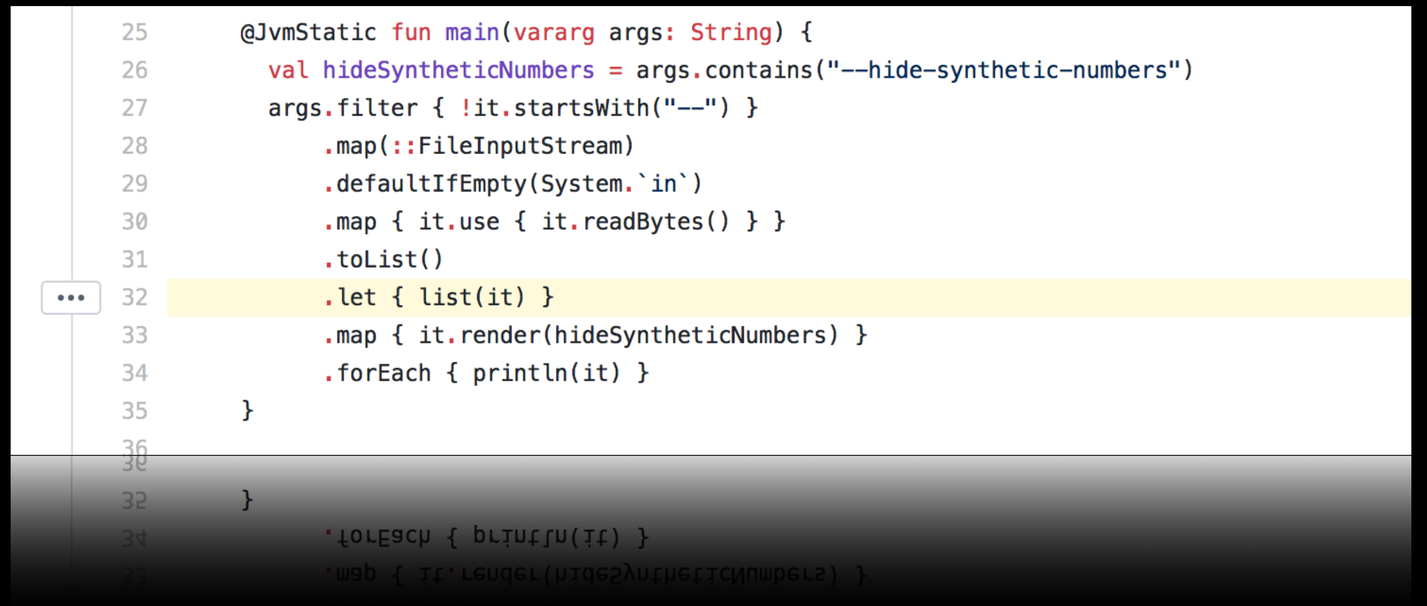
```kotlin
public inline fun <T, R> T.let(
  block: (T) -> R
): R = block(this)
```



let

```
val usernameView: View? = user.name?.let { name ->
    ViewWithText(context, name)
}
```

let



let

DexMethods.kt: http://bit.ly/2xFN3Bd

🎉 Nullablity 🎉

let

let

also

```kotlin
public inline fun <T> T.also(
  block: (T) -> Unit
): T {
  block(this);
  return this
}
```

also

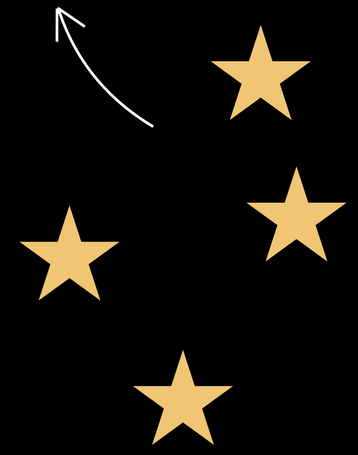
```
public inline fun <T> T.also(
  block: (T) -> Unit
): T {
  block(this);
  return this
}
```

also

```
public inline fun <T> T.also(
  block: (T) -> Unit
): T {
  block(this);
  return this
}
```

also

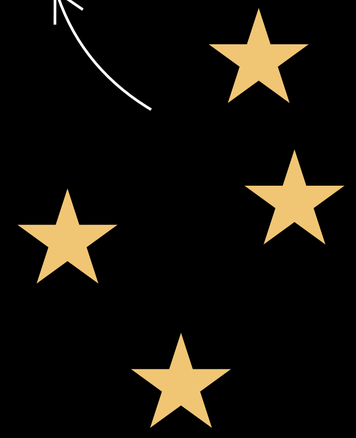
```
public inline fun <T> T.also(
  block: (T) -> Unit
): T {
  block(this);
  return this
}
```



also

```
public inline fun <T> T.also(
  block: (T) -> Unit
): T {
  block(this);
  return this
}
```

Plain
lambda



also

“Oh and also…”

```
public inline fun <T> T.also(
  block: (T) -> Unit
): T {
  block(this);
  return this
}
```

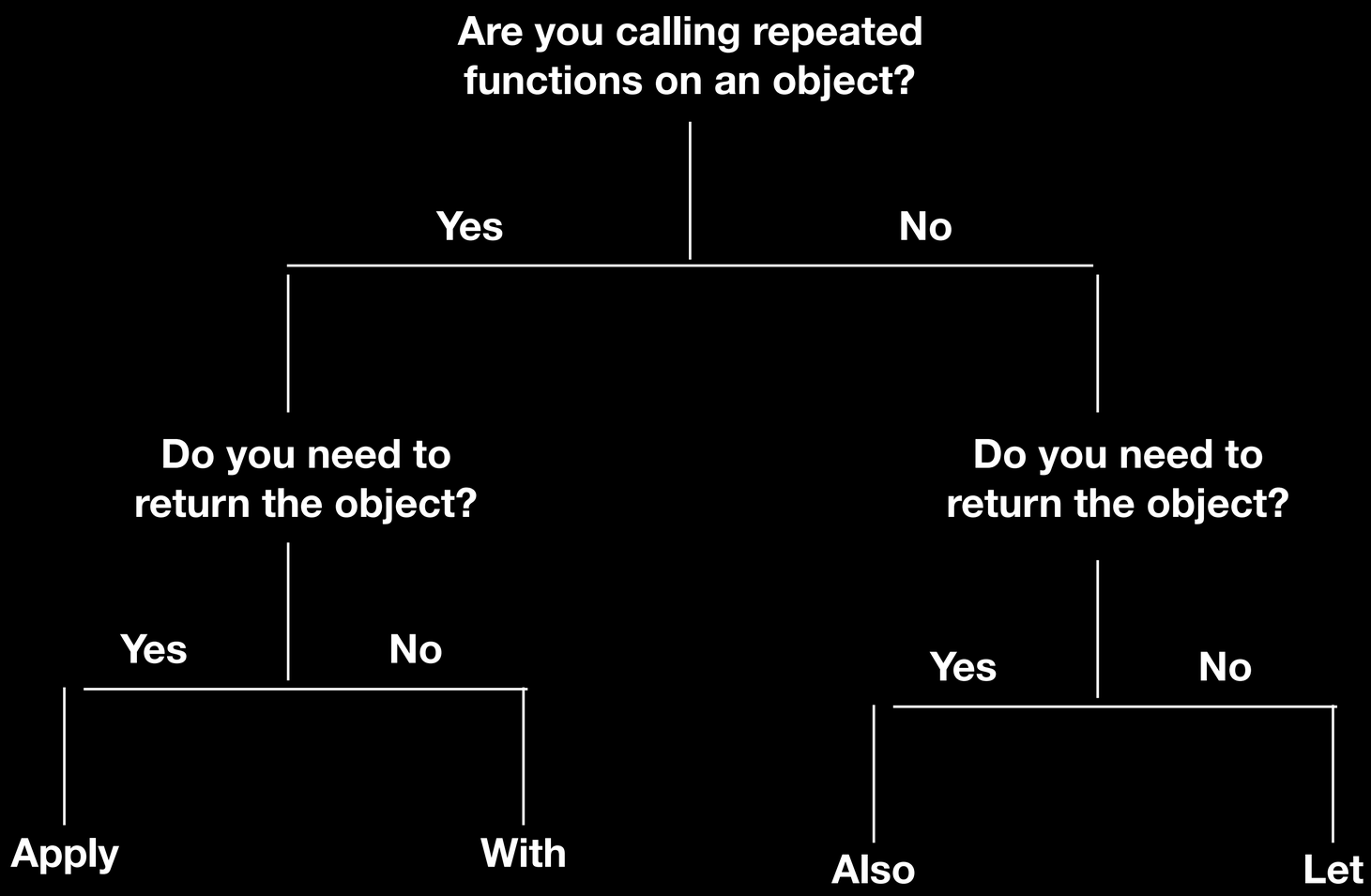
also

“Oh and also…”

```
foo.bar()
   .also { }
   .baz()
```

also

???


Are you calling repeated functions on an object?
Yes
No
Do you need to return the object?
Do you need to return the object?
Yes
No
Yes
No
Apply
With
Also
Let

	need caller to be returned	don’t need caller returned
need receiver	apply	with
don’t need receiver	also	let

1. Things return values you don’t
always need to use

2. You can call non-receiver functions
inside lambdas with receivers

```
val idk = with(foo) {
    bar()
    baz()
}
```

```
val idk: Baz = with(foo) {
    bar()
    baz()
}
```

```
val idk: Baz = with(foo) {
    bar()
    createBaz()
}
```

With: Logically grouped calls on an object

Apply: Initialization or configuration

Let: Nulls + conversions of single objects

Also: Ancillary/Side effect-y code in chains

With: Logically grouped calls on an object

Apply: Initialization or configuration

Let: Nulls + conversions of single objects

Also: Ancillary/Side effect-y code in chains

Run: ???

run

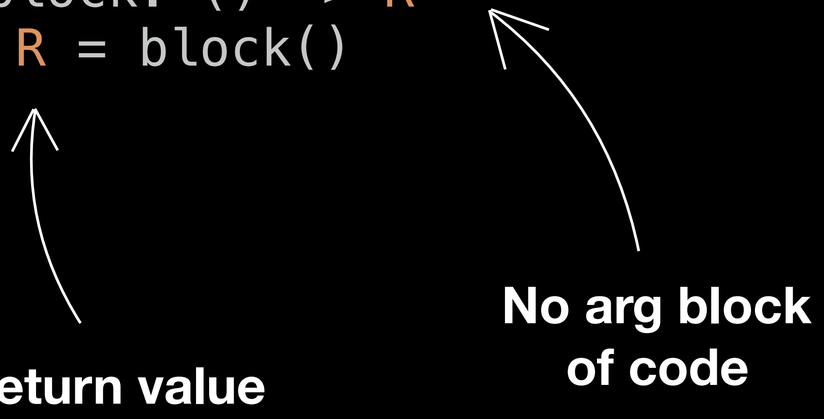
```
public inline fun <R> run(
  block: () -> R
): R = block()


public inline fun <T, R> T.run(
  block: T.() -> R
): R = block()
```

run

```
public inline fun <R> run(
  block: () -> R
): R = block()
```

run

```
public inline fun <R> run(
  block: () -> R
): R = block()
```



run

```
public inline fun <R> run(
  block: () -> R
): R = block()
```

```
fun myFun(user: User) {
  val foo = Foo()
  run {
    foo.baz()
    foo.bar()
  }
}
```

run

```
public inline fun <R> run(
  block: () -> R
): R = block()
```

```
fun myFun(user: User) {
  val foo = Foo()
  {
    foo.baz()
    foo.bar()
  }()
}
```

run

```
public inline fun <R> run(
  block: () -> R
): R = block()
```

```
fun myFun(user: User) {
  val foo = Foo()
  {
    foo.baz()
    foo.bar()
  }()
}
```

run

```
public inline fun <R> run(
  block: () -> R
): R = block()
```

```
fun myFun(user: User) {
  val foo = Foo()
  foo.baz()
  foo.bar()
}
```

run

```
public inline fun <R> run(
  block: () -> R
): R = block()
```

```
fun myFun(user: User) {
  val foo = Foo()
  val result = run {
    foo.baz()
    foo.bar()
  }
}
```

run

```
public inline fun <R> run(
  block: () -> R
): R = block()
```

```
fun myFun(user: User) {
  val foo = Foo()
  val result = {
    foo.baz()
    foo.bar()
  }()
}
```

run

```
public inline fun <R> run(
  block: () -> R
): R = block()
```

```
val repo = currentRepo ?: run { initializeRepo() }
```

run

```
public inline fun <R> run(
  block: () -> R
): R = block()


public inline fun <T, R> T.run(
  block: T.() -> R
): R = block()
```

run

```
public inline fun <T, R> T.run(
  block: T.() -> R
): R = block()
```

run

```
public inline fun <T, R> with(
   receiver: T,
   block: T.() -> R
): R = receiver.block()
```

```
public inline fun <T, R> T.run(
   block: T.() -> R
): R = block()
```

run

```
with(db) {
    open()
    commit()
    close()
}
```

Don’t want return value

vs.

```
db?.run {
    open()
    commit()
    close()
}
```

Can insert ?.

run

```
with(db) {
    open()
    commit()
    close()
}
```

Don’t want return value

vs.

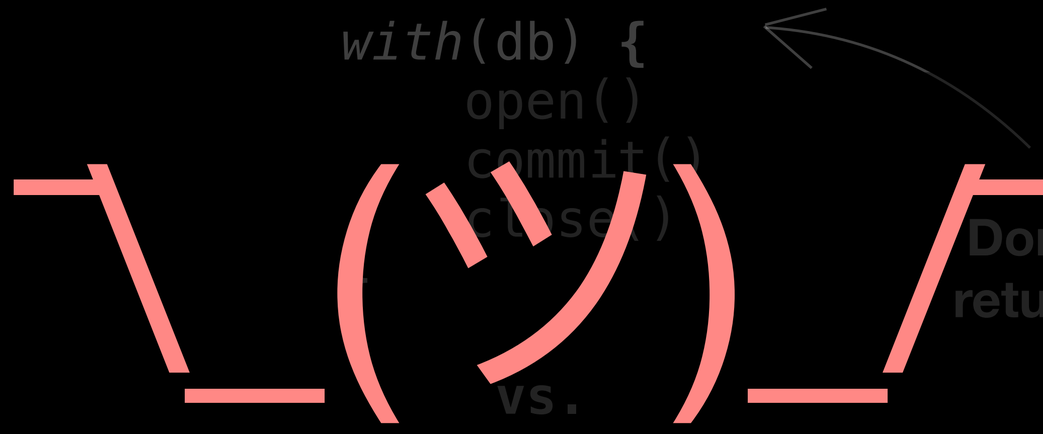
```
db?.run {
    open()
    commit()
    close()
}
```

Can insert ?.

```
if (db != null) {
    //lambda with receiver
}
```

run

```
with(db) {
    open()
    commit()
    close()
}
```



Don’t want return value

vs.

```
db.run {
    open()
    commit()
    close()
}
```

Can insert ?.

```
if (db != null) {
    //lambda with receiver
}
```

With: Logically grouped calls on an object

Apply: Initialization or configuration

Let: Nulls + conversions of single objects

Also: Ancillary/Side effect-y code in chains

Run: With + nullability (*?:* and *?.*)

With: Logically grouped calls on an object

Apply: Initialization or configuration

Let: Nulls (and maybe conversions of single objects)

Also: Ancillary/Side effect-y code in chains

Run: With + nullability (*?:* and *?.*) (Maybe…?)

Array<Int>

Array<Int>

IntArray

Array<Int>

IntArray

¯_(ツ)_/¯

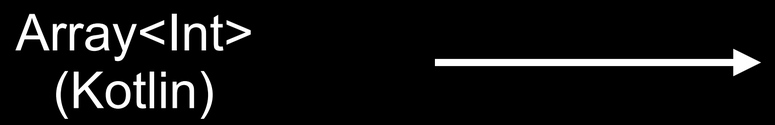
Non null types in Kotlin tend to
be compiled to Java’s primitives

List<Int>
(Kotlin)

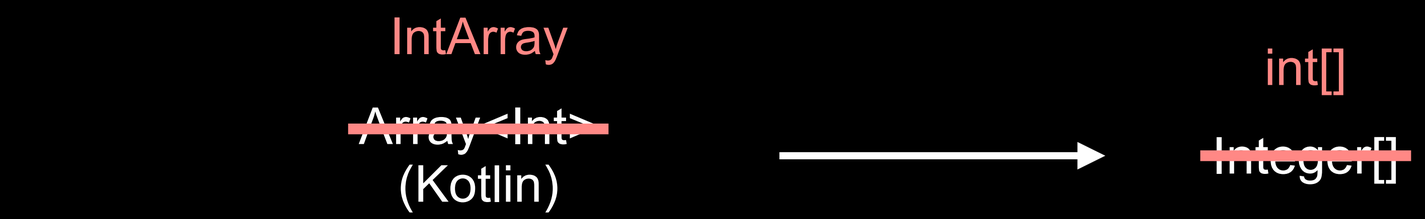
List<Integer>
(Java)


Array<Int>
(Kotlin)
Integer[]

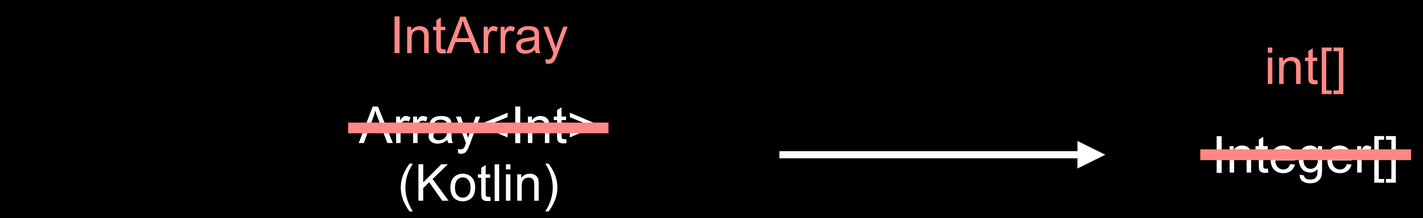
```
Object[] result$iv = new Integer[size$iv];
```




Array<Int>
(Kotlin)
int[] ?
Integer[]


IntArray
~~Array<Int>~~
(Kotlin)
int[]
~~Integer[]~~

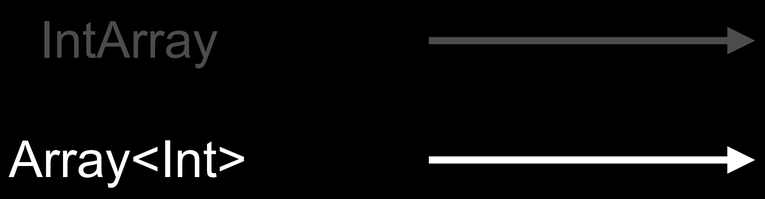
```
int[] result$iv = new int[size$iv];
```




IntArray
int[]
Array<Int>
Integer[]

Performance

Performance

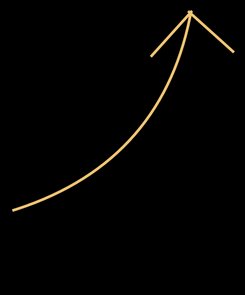


Performance

Delegates.notNull
vs
lateinit

Lateinit	Delegates.notNull
must be a variable -> can be reassigned	can be val or var
can’t be a “primitive”	can be a “primitive”

Lateinit	Delegates.notNull
must be a variable -> can be reassigned	can be val or var
can’t be a “primitive”	can be a “primitive”



Basically no restrictions

Lateinit

Delegates.notNull

Forcing functions: val(?) or primitive

must be a variable -> can be reassigned

can be val or var

can’t be a “primitive”

can be a “primitive”

Delegates.notNull

Delegates.notNull

Delegates.notNull

```
private class NotNullVar<T: Any>() : ReadWriteProperty<Any?, T> {
    private var value: T? = null

    public override fun getValue(...): T {
        return value ?: throw IllegalStateException(
            "Property ${property.name} should be initialized before get.")
    }

    public override fun setValue(..., value: T) {
        this.value = value
    }
}
```

Delegates.notNull

```
private class NotNullVar<T: Any>() : ReadWriteProperty<Any?, T> {
    private var value: T? = null

    public override fun getValue(...): T {
        return value ?: throw IllegalStateException(
            "Property ${property.name} should be initialized before get.")
    }

    public override fun setValue(..., value: T) {
        this.value = value
    }
}
```

Delegates.notNull

```
private class NotNullVar<T: Any>() : ReadWriteProperty<Any?, T> {
    private var value: T? = null

    public override fun getValue(...): T {
        return value ?: throw IllegalStateException(
            “Property ${property.name} should be initialized before get.")
    }

    public override fun setValue(..., value: T) {
        this.value = value
    }
}
```

Delegates.notNull

```
private class NotNullVar<T: Any>() : ReadWriteProperty<Any?, T> {
    private var value: T? = null

    public override fun getValue(...): T {
        return value ?: throw IllegalStateException(
            Property ${property.name} should be initialized before get.")
    }

    public override fun setValue(..., value: T) {
        this.value = value
    }
}
```

Delegates.notNull

Delegates.notNull?

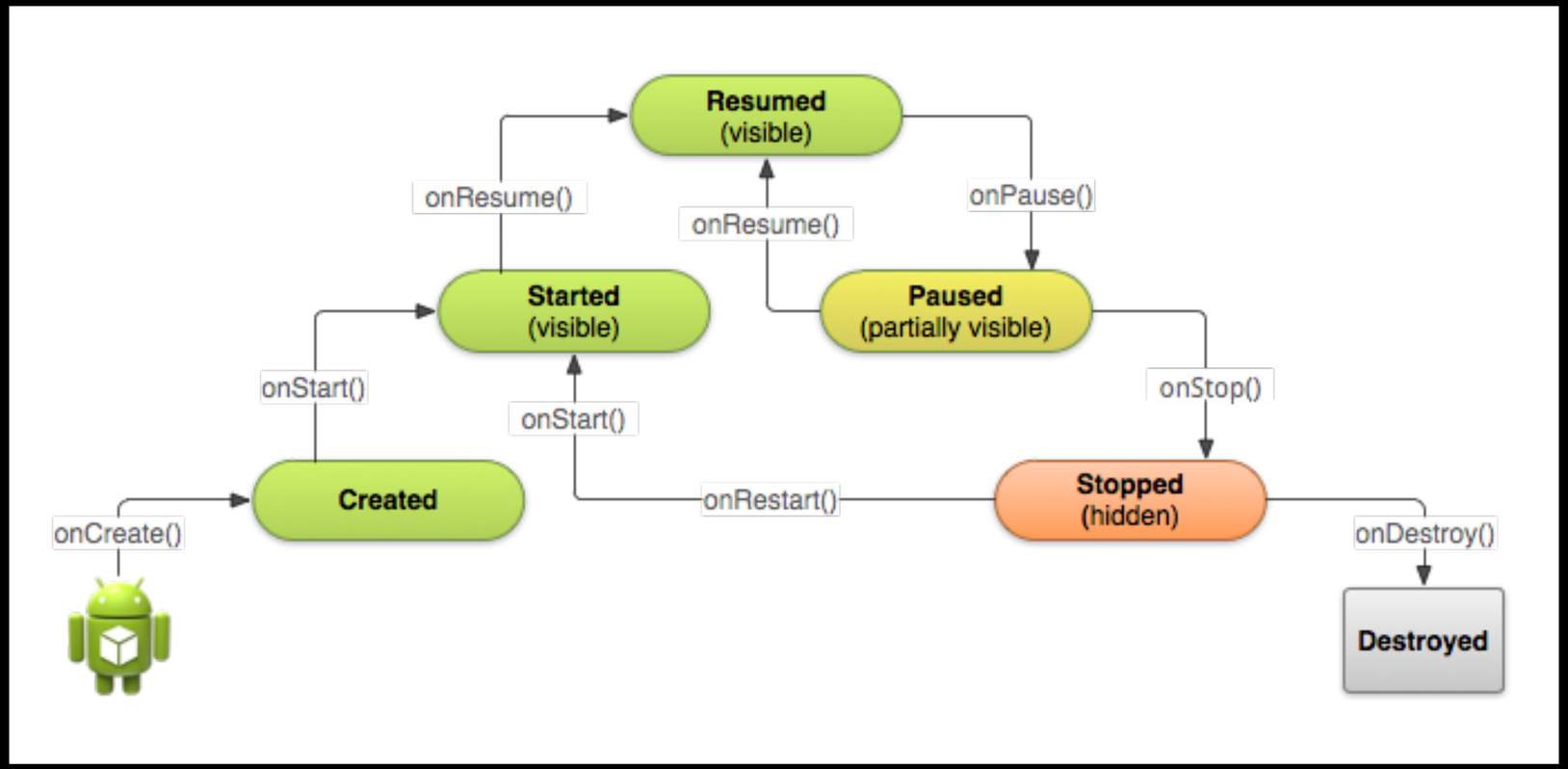
lateinit

late initialized

late initialized

```
class User {
    val firstName: String = "Christina"
    val lastName: String

    init {
        lastName = "Lee"
    }
}
```

late initialized



late initialized

```
class TestActivity : AppCompatActivity() {
    lateinit var button: Button

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_test)
        button = findViewById(R.id.button) as Button
    }
}
```

???

late initialized:

when you can

late initialized:

when you can*

Inline fun
vs
Custom getter

```
class User(val firstName: String, val lastName: String,
           var height: Inch) {

    val isTall
        get() = height > SixFeet

}
```

```
class User(val firstName: String, val lastName: String,
           var height: Inch) {

    fun isTall(): Boolean = height > SixFeet

}
```

```
val me = User(firstName = "Christina",
              lastName = "Lee",
              height = 68)
```

```
val me = User(firstName = "Christina",
              lastName = "Lee",
              height = 68)
```

```
me.isTall          vs          me.isTall()
```

```
val me = User(firstName = "Christina",
              lastName = "Lee",
              height = 68)
```

```
me.isTall
```

?

```
me.isTall()
```

`me.isTall` ? `me.isTall()`

```
public final isTall1()Z
L0
    LINENUMBER 16 L0
    ALOAD 0
    GETFIELD User.height : I
    BIPUSH 72
    IF_ICMPLE L1
    ICONST_1
    GOTO L2
L1
    ICONST_0
L2
    IRETURN
L3
    LOCALVARIABLE this User; L0 L3 0
    MAXSTACK = 2
    MAXLOCALS = 1
```

```
public final isTall2()Z
L0
    LINENUMBER 18 L0
    ALOAD 0
    GETFIELD User.height : I
    BIPUSH 72
    IF_ICMPLE L1
    ICONST_1
    GOTO L2
L1
    ICONST_0
L2
    IRETURN
L3
    LOCALVARIABLE this User; L0 L3 0
    MAXSTACK = 2
    MAXLOCALS = 1
```

`me.isTall` ? `me.isTall()`

```
public final isTall2()Z
L0
    LINENUMBER 16 L0
    ALOAD 0
    GETFIELD User.height : I
    BIPUSH 72
    IF_ICMPLE L1
    ICONST_1
    GOTO L2
L1
    ICONST_0
L2
    IRETURN
L3
    LOCALVARIABLE this User; L0 L3 0
    MAXSTACK = 2
    MAXLOCALS = 1
```

`me.isTall` **?** `me.isTall()`

```
public final isTall2()Z
L0
    LINENUMBER 16 L0
    ALOAD 0
    GETFIELD User.height : I
    BIPUSH 72
    IF_ICMPLE L1
    ICONST_1
    GOTO L2
L1
    ICONST_0
L2
    IRETURN
L3
    LOCALVARIABLE this User; L0 L3 0
    MAXSTACK = 2
    MAXLOCALS = 1
```

There is no difference in implementation or performance; they only differ in readability.
- Kotlin in Action, pg 26

`me.isTall` ? `me.isTall()`

```
me.firstName
me.lastName
me.age
me.isTall
```

?

```
me.isTall()
```

```
me.firstName
me.lastName
me.age
me.isTall
```

?

```
me.getFriends()
me.clear()
me.clone()
me.isTall()
```

```
me.firstName
me.lastName
me.age
me.isTall
```

?

```
me.getFriends()
me.clear()
me.clone()
me.isTall()
```

data

```
me.firstName
me.lastName
me.age
me.isTall
```

?

```
me.getFriends()
me.clear()
me.clone()
me.isTall()
```

data

actions

```
me.isTall
```

data

attribute of the type

on the order of a field access

no overt side effects

?

```
me.isTall()
```

actions

computationally expensive

conversions or copies

return value changes each time

`me.isTall` ? `me.isTall()`

```
me.isTall
```

```
me.isTall()
```

Map/Filter
+
Sequence

```kotlin
 * Elements in the set returned are sorted according to the given [comparator].
 */
@kotlin.jvm.JvmVersion
public fun <T> Array<out T>.toSortedSet(comparator: Comparator<in T>): SortedSet<T> {
    return toCollection(TreeSet<T>(comparator))
}

/**
 * Returns a single list of all elements yielded from results of [transform] function being invoked on each element of original array.
 */
public inline fun <T, R> Array<out T>.flatMap(transform: (T) -> Iterable<R>): List<R> {
    return flatMapTo(ArrayList<R>(), transform)
}

/**
 * Appends all elements yielded from results of [transform] function being invoked on each element of original array, to the given [destination].
 */
public inline fun <T, R, C : MutableCollection<in R>> Array<out T>.flatMapTo(destination: C, transform: (T) -> Iterable<R>): C {
    for (element in this) {
        val list = transform(element)
        destination.addAll(list)
    }
    return destination
}

/**
 * Groups elements of the original array by the key returned by the given [keySelector] function
 * applied to each element and returns a map where each group key is associated with a list of corresponding elements.
 *
 * The returned map preserves the entry iteration order of the keys produced from the original array.
 *
 * @sample samples.collections.Collections.Transformations.groupBy
 */
public inline fun <T, K> Array<out T>.groupBy(keySelector: (T) -> K): Map<K, List<T>> {
    return groupByTo(LinkedHashMap<K, MutableList<T>>(), keySelector)
}

/**
 * Groups values returned by the [valueTransform] function applied to each element of the original array
 * by the key returned by the given [keySelector] function applied to the element
 * and puts to the [destination] map each group key associated with a list of corresponding values.
 *
 * @return The [destination] map.
 *
 * @sample samples.collections.Collections.Transformations.groupByKeysAndValues
 */
public inline fun <T, K, V, M : MutableMap<in K, MutableList<V>>> Array<out T>.groupByTo(destination: M, keySelector: (T) -> K, valueTransform: (T) -> V): M {
    for (element in this) {
        val key = keySelector(element)
        val list = destination.getOrPut(key) { ArrayList<V>() }
        list.add(valueTransform(element))
    }
```

```
val arr = arrayOf("1", "2", "3", “4”)

val numGreaterThanTwo = arr.map { Integer.valueOf(it) }
                           .filter { it > 2 }
                           .count()
```

```
Object[] elements$iv = new String[]{"1", "2", "3", “4"};
Object[] $receiver$iv = (Object[])elements$iv;
Object[] $receiver$iv$iv = $receiver$iv;
Collection destination$iv$iv = (Collection)(new ArrayList($receiver$iv.length));

Object element$iv$iv;
for(int var6 = 0; var6 < $receiver$iv$iv.length; ++var6) {
   element$iv$iv = $receiver$iv$iv[var6];
   String it = (String)element$iv$iv;
   Integer var13 = Integer.valueOf(it);
   destination$iv$iv.add(var13);
}

Iterable $receiver$iv = (Iterable)((List)destination$iv$iv);
destination$iv$iv = (Collection)(new ArrayList());
Iterator var17 = $receiver$iv.iterator();

while(var17.hasNext()) {
   element$iv$iv = var17.next();
   Integer it = (Integer)element$iv$iv;
   if(Intrinsics.compare(it.intValue(), 2) > 0) {
      destination$iv$iv.add(element$iv$iv);
   }
}

Collection var16 = (Collection)((List)destination$iv$iv);
int numbersGreaterThanTwo = var16.size();
```

```
Object[] elements$iv = new String[]{"1", "2", "3", “4"};
Object[] $receiver$iv = (Object[])elements$iv;
Object[] $receiver$iv$iv = $receiver$iv;
Collection destination$iv$iv = (Collection)(new ArrayList($receiver$iv.length));

Object element$iv$iv;
for(int var6 = 0; var6 < $receiver$iv$iv.length; ++var6) {
   element$iv$iv = $receiver$iv$iv[var6];
   String it = (String)element$iv$iv;
   Integer var13 = Integer.valueOf(it);
   destination$iv$iv.add(var13);
}

Iterable $receiver$iv = (Iterable)((List)destination$iv$iv);
destination$iv$iv = (Collection)(new ArrayList());
Iterator var17 = $receiver$iv.iterator();

while(var17.hasNext()) {
   element$iv$iv = var17.next();
   Integer it = (Integer)element$iv$iv;
   if(Intrinsics.compare(it.intValue(), 2) > 0) {
      destination$iv$iv.add(element$iv$iv);
   }
}

Collection var16 = (Collection)((List)destination$iv$iv);
int numbersGreaterThanTwo = var16.size();
```

```
Object[] elements$iv = new String[]{“1", "2", "3", “4"};


Collection destination$iv$iv = (Collection)(new ArrayList($receiver$iv.length));
Object element$iv$iv;
for(int var6 = 0; var6 < $receiver$iv$iv.length; ++var6) {
   Integer var13 = Integer.valueOf(it);
}


destination$iv$iv = (Collection)(new ArrayList());
while(var17.hasNext()) {
   if(Intrinsics.compare(it.intValue(), 2) > 0) {
      destination$iv$iv.add(element$iv$iv);
   }
}


int numbersGreaterThanTwo = var16.size();
```

```
Collection destination$iv$iv = (Collection)(new ArrayList($receiver$iv.length));
Object element$iv$iv;
for(int var6 = 0; var6 < $receiver$iv$iv.length; ++var6) {
   Integer var13 = Integer.valueOf(it);
}
```

```
destination$iv$iv = (Collection)(new ArrayList());
while(var17.hasNext()) {
   if(Intrinsics.compare(it.intValue(), 2) > 0) {
      destination$iv$iv.add(element$iv$iv);
   }
}
```

Intermediate Collections

Intermediate Collections

```
val arr = arrayOf(“1", "2", "3", “4")

val numGreaterThanTwo = arr.asSequence()
        .map { Integer.valueOf(it) }
        .filter { it > 2 }
        .count()
```

```
arr.map { }
   .filter { }
   .count()
```

```
Mapping 1
Mapping 2
Mapping 3
Mapping 4

Filtering 1
Filtering 2
Filtering 3
Filtering 4
```

```
arr.asSequence()
   .map { }
   .filter { }
   .count()
```

```
Mapping 1
Filtering 1

Mapping 2
Filtering 2

Mapping 3
Filtering 3

Mapping 4
Filtering 4
```

```
arr.map { }
   .filter { }
```

```
arr.asSequence()
   .map { }
   .filter { }
```

```
Mapping 1
Mapping 2
Mapping 3
Mapping 4

Filtering 1
Filtering 2
Filtering 3
Filtering 4
```

```
arr.map { }
   .filter { }
```

```
Mapping 1
Mapping 2
Mapping 3
Mapping 4

Filtering 1
Filtering 2
Filtering 3
Filtering 4
```

```
arr.asSequence()
   .map { }
   .filter { }
```

Requires a terminal operation

```
internal class TransformingSequence<T, R>
constructor(private val sequence: Sequence<T>,
   private val transformer: (T) -> R) : Sequence<R> {
    override fun iterator(): Iterator<R> = object : Iterator<R> {
        val iterator = sequence.iterator()
        override fun next(): R {
            return transformer(iterator.next())
        }

        override fun hasNext(): Boolean {
            return iterator.hasNext()
        }
    }

    // . . .
}
```

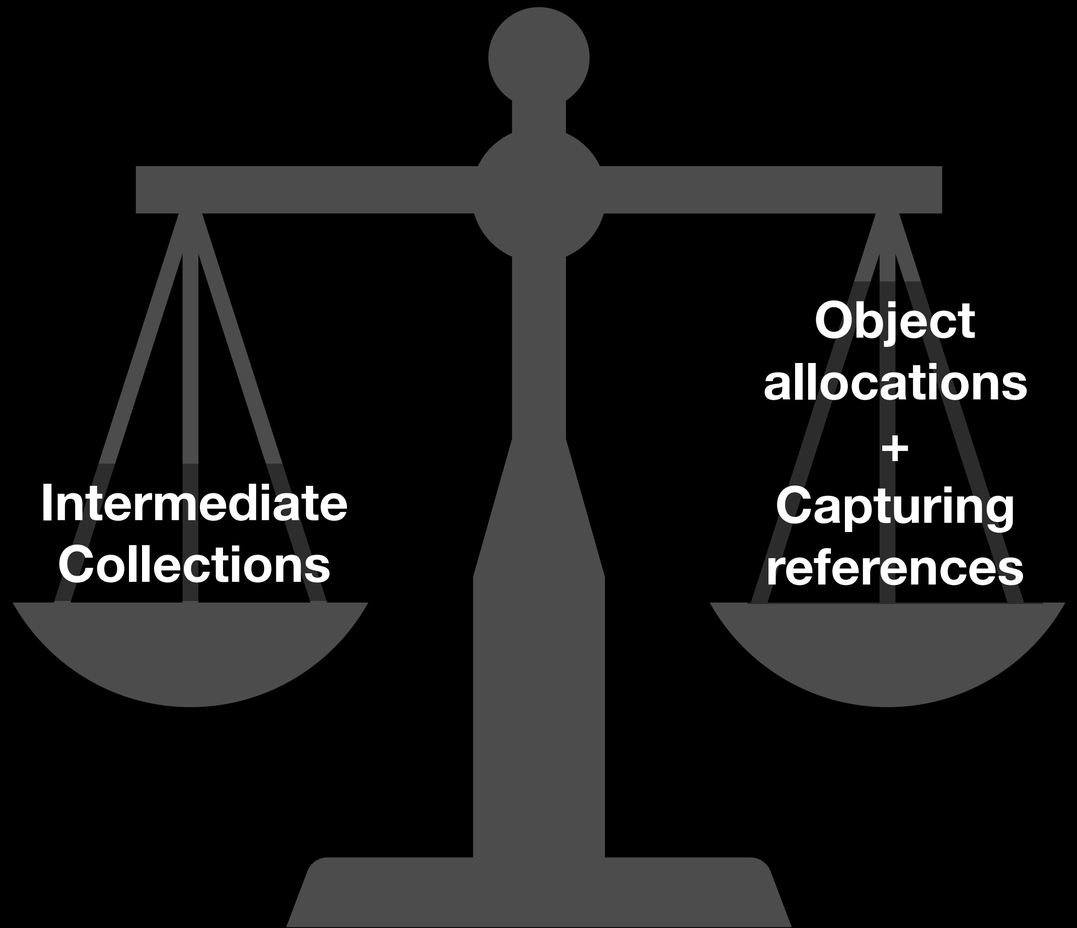
```
override fun next(): R {
  return transformer(iterator.next())
}
```

```
for (item in this)
    destination.add(transform(item))
return destination
```

Performance

Wholistic Performance

Performance



Rule of thumb:

Use asSequence if you have
a compelling large data set
or have profiled perf issues.

The End

Christina Lee

@RunChristinaRun