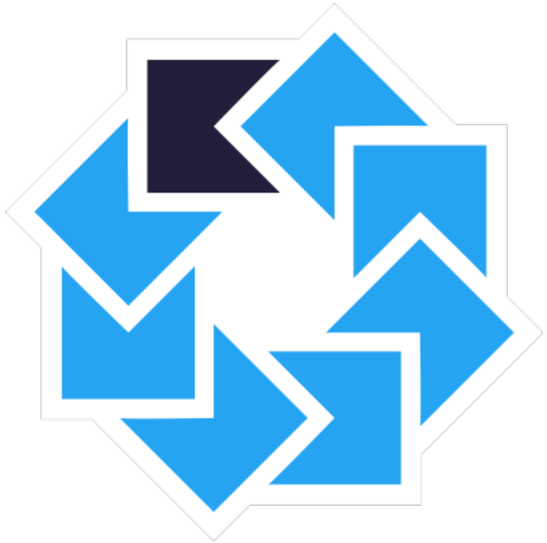
#4: Generics

Pavel Shchors

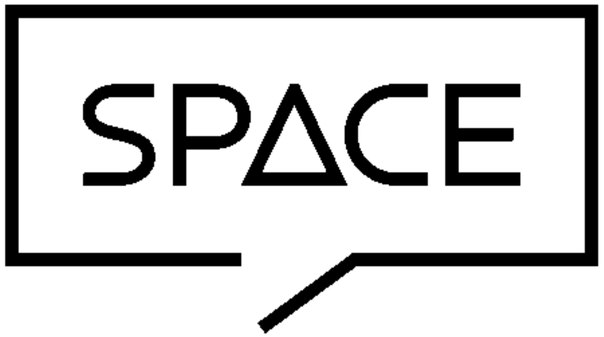
Android Developer@JUNO

pavel.shchors

pavel.shchors@gojuno.com




fitbit


SPACE


juno


JET
BRAINS

“Ordinary classes and methods work with specific types: either primitives or class types. If you are writing code that might be used across more types, this rigidity can be overconstraining."

– Bruce Eckel, "Thinking in Java"


Why generics?

Why generics?

```
class List {
    val elements = ...

    val size = elements.size

    fun add(element: Any?) = elements.add(element)

    fun get(index: Int): Any? = elements.get(index)
}
```

Why generics?

```
class List {
    val elements = ...

    val size = elements.size

    fun add(element: Any?) = elements.add(element)

    fun get(index: Int): Any? = elements.get(index)
}
```

Why generics?

```
class List {
    val elements = ...

    val size = elements.size

    fun add(element: Any?) = elements.add(element)

    fun get(index: Int): Any? = elements.get(index)
}
```

Why generics?

```
val listOfInt = List()
```

Why generics?

```
val listOfInt = List()

listOfInt.add(1)
listOfInt.add(2)
```

Why generics?

```
val listOfInt = List()

listOfInt.add(1)
listOfInt.add(2)

val first: Int = listOfInt.get(0) as Int
```

Why generics?

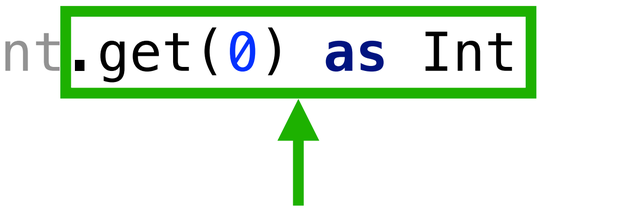
```
val listOfInt = List()

listOfInt.add(1)
listOfInt.add(2)

val first: Int = listOfInt.get(0) as Int
```

Why generics?

```
val listOfInt = List()

listOfInt.add(true)
listOfInt.add(1)
listOfInt.add(2)

val first: Int = listOfInt.get(0) as Int
```

Why generics?

```
val listOfInt = List()

listOfInt.add(true)
listOfInt.add(1)
listOfInt.add(2)

val first: Int = listOfInt.get(0) as Int
```



Why generics?

```
class IntList : List() {

    override fun get(index: Int): Int = elements.get(index)









}
```

Why generics?

```
class IntList : List() {

    override fun get(index: Int): Int = elements.get(index)

    override fun add(element: Any?) {



    }
}
```

Why generics?

```
class IntList : List() {

    override fun get(index: Int): Int = elements.get(index)

    override fun add(element: Any?) {
        if (element is Int) {
            elements.add(element)
        }


    }
}
```

Why generics?

```
class IntList : List() {

    override fun get(index: Int): Int = elements.get(index)

    override fun add(element: Any?) {
        if (element is Int) {
            elements.add(element)
        } else {
            throw IllegalArgumentException()
        }
    }
}
```

Why generics?

```
val listOfInt = IntList()

listOfInt.add(1)
listOfInt.add(2)

val first = listOfInt.get(1)
```

Why generics?

```
val listOfInt = IntList()

listOfInt.add(1)
listOfInt.add(2)

val first = listOfInt.get(1)
```

Why generics?

```
val listOfInt = IntList()

listOfInt.add(true)
listOfInt.add(1)
listOfInt.add(2)

val first = listOfInt.get(1)
```

Why generics?

```
val listOfInt = IntList()

listOfInt.add(true)
listOfInt.add(1)
listOfInt.add(2)

val first = listOfInt.get(1)

IllegalArgumentException
```

Why generics?

```
class IntList : List() {
    ...
}
```

Why generics?

```
class IntList : List() {
    ...
}

class DoubleList : List() {
    ...
}
```

Why generics?

```
class IntList : List() {
    ...
}

class DoubleList : List() {
    ...
}

class StringList : List() {
    ...
}
```

Why generics?

```
class IntList : List() {
    ...
}

class DoubleList : List() {
    ...
}

class StringList : List() {
    ...
}
```

More to come...

Why generics?

```
class List<Type> {
    val elements = ...

    val size = elements.size

    fun add(element: Type) = elements.add(element)

    fun get(index: Int): Type = elements.get(index)
}
```

Why generics?

```
class List<Type> {
    val elements = ...

    val size = elements.size

    fun add(element: Type) = elements.add(element)

    fun get(index: Int): Type = elements.get(index)
}
```

Why generics?

```
class List<Type> {
    val elements = ...

    val size = elements.size

    fun add(element: Type) = elements.add(element)

    fun get(index: Int): Type = elements.get(index)
}
```

Why generics?

```
class List<Type> {
    val elements = ...

    val size = elements.size

    fun add(element: Type) = elements.add(element)

    fun get(index: Int): Type = elements.get(index)
}
```

Why generics?

```
val listOfInt = List<Int>()

listOfInt.add(1)
listOfInt.add(2)

val first = listOfInt.get(1)
```

Why generics?

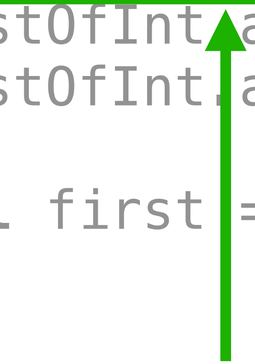
```
val listOfInt = List<Int>()

listOfInt.add(1)
listOfInt.add(2)

val first = listOfInt.get(1)
```

Why generics?

```
val listOfInt = List<Int>()

listOfInt.add(true)
listOfInt.add(1)
listOfInt.add(2)

val first = listOfInt.get(1)
```

Why generics?

```
val listOfInt = List<Int>()

listOfInt.add(true)
listOfInt.add(1)
listOfInt.add(2)

val first = listOfInt.get(1)
```



Type mismatch

Why generics?

```
val listOfInt = List<Int>()
```

Why generics?

```
val listOfInt = List<Int>()

val listOfDouble = List<Double>()
```

Why generics?

```
val listOfInt = List<Int>()

val listOfDouble = List<Double>()

val listOfString = List<String>()
```

Why generics?

```
val listOfInt = List<Int>()

val listOfDouble = List<Double>()

val listOfString = List<String>()
```

More to come...

Why generics?

- Compile-time safety

Why generics?

- Compile-time safety
- Avoid boilerplate code

Why generics?

- Compile-time safety
- Avoid boilerplate code
- Write reusable code

Generic Functions and Properties

Generic function declaration

```
fun <T> Iterable<T?>.filterNotNull(): List<T>
```

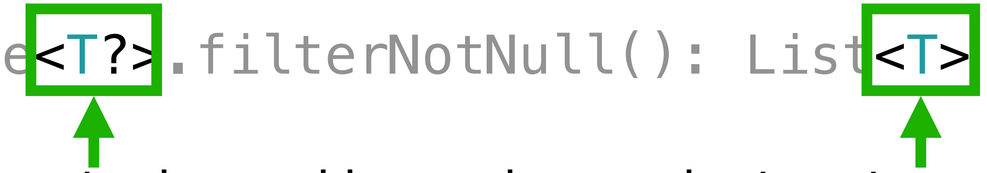
Generic function declaration

```
fun <T> Iterable<T?>.filterNotNull(): List<T>
```

Type parameter declaration

Generic function declaration

```
fun <T> Iterable<T?>.filterNotNull(): List<T>
```



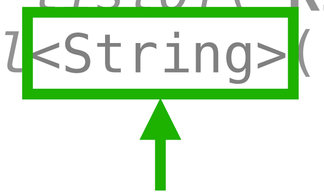
Type parameter is used in receiver and return types

Generic function call

```
val names: List<String?> = listOf("Kirill","Anton", null)
println(names.filterNotNull<String>())
>>>["Kirill","Anton"]
```

Generic function call

```
val names: List<String?> = listOf("Kirill","Anton", null)
println(names.filterNotNull<String>())
>>>["Kirill","Anton"]
```



Specify the type argument explicitly

Generic function call

```
val names: List<String?> = listOf("Kirill","Anton", null)
println(names.filterNotNull())
>>>["Kirill","Anton"]
```

Generic function call

```
val names: List<String?> = listOf("Kirill","Anton", null)
println(names.filterNotNull())
>>>["Kirill","Anton"]
```

Compiler is clever enough to infer that T is String here

Generic extension property

```
val <T> List<T>.isTooBig: Boolean
    get() = this.size > 100
```

Generic extension property

```
var <T> List<T>.isTooBig: Boolean
    get() = this.size > 100
    set(value) = Unit
```

Generic classes

Generic class declaration

```
interface List<T>{
    fun get(index: Int): T
}
```

Generic class declaration

```
interface List<T>{
    fun get(index: Int): T
}
```

Generic class declaration

```
interface List<T>{
    fun get(index: Int): T
}
```

Extending generic class

```
class SchoolsList: List<School> {
    override fun get(index: Int): School = ...
}
```

Extending generic class

```
class ArrayList<K> : List<K> {
    override fun get(index: Int): K = ...
}
```

Extending generic class

```
class ArrayList<K> : List<K> {
    override fun get(index: Int): K = ...
}
```

Extending generic class

```
class ArrayList<K> : List<K> {
    override fun get(index: Int): K = ...
}
```

Type parameter constraints

Type parameter constraints

```
fun <T> List<T>.multiplyAll(): T
```

Type parameter constraints

```
fun <T: Number> List<T>.multiplyAll(): T
```

Type parameter constraints

```
fun <T: Number> List<T>.multiplyAll(): T
```

Type parameter with upper bound

Multiple type parameter constraints

```
fun <T> ensureTrailingPeriod(seq: T)
        where T : CharSequence, T : Appendable {
    if (!seq.endsWith('.')) {
        seq.append('.')
    }
}
```

Multiple type parameter constraints

```
fun <T> ensureTrailingPeriod(seq: T)
        where T : CharSequence, T : Appendable {
    if (!seq.endsWith('.')) {
        seq.append('.')
    }
}
```

Nullability constraints

```
class ClassNamePrinter<T> {
    fun printClassName(value: T) {
        println(value::class)
    }
}
```

Nullability constraints

```
class ClassNamePrinter<T> {
    fun printClassName(value: T) {
        println(value::class)
    }
}
```

Expression in a type literal has a nullable type 'T'

Nullability constraints

```
class ClassNamePrinter<T: Any?> {
    fun printClassName(value: T) {
        println(value::class)
    }
}
```

Nullability constraints

```
class ClassNamePrinter<T: Any> {
    fun printClassName(value: T) {
        println(value::class)
    }
}
```

What about Java?

- Generic classes work the same way

What about Java?

- Generic classes work the same way

- No extension functions and properties

What about Java?

- Generic classes work the same way
- No extension functions and properties
- You can create static generic methods

What about Java?

- Generic classes work the same way
- No extension functions and properties
- You can create static generic methods
- Java classes support type parameter constraints

Generics at runtime

Type Erasure

```
val listA: List<Int> = listOf(1, 2, 3)
val listB: List<String> = listOf("a", "b", "c")

println("listB class = ${listA::class}")
println("listB class = ${listB::class}")
```

Type Erasure

```
val listA: List<Int> = listOf(1, 2, 3)
val listB: List<String> = listOf("a", "b", "c")

println("listB class = ${listA::class}")
println("listB class = ${listB::class}")

>>> listA class = java.util.Arrays$ArrayList
>>> listB class = java.util.Arrays$ArrayList
```

Type Erasure

```
if(value is List<String>) {
    //code
}
```

Type Erasure

```
if(value is List<String>) {
    //code
}
```

Cannot check for instance of erased type: List<String>

Type Erasure

```
value as List<String>
```

Type Erasure

```
value as List<String>
```

Unchecked cast: List<*> to List<String>

Type Erasure

```
if(value is List<*>){
    //code
}
```

Type Erasure

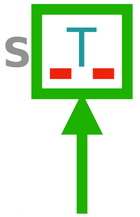
```
fun printStrings(value: Collection<String>) {
    if(value is List<String>) {
        value.joinToString()
    }
}
```

Reified type

```
fun <T> isInstanceOf(value: Any) = value is T
```

Reified type

```
fun <T> isInstanceOf(value: Any) = value is T
```



Cannot check for instance of erased type: T

Reified type

```
fun <T> isInstanceOf(value: Any) = value is T
```

```
inline fun <reified T> isInstanceOf(value: Any) = value is T
```

Reified type

```
inline fun <reified T> isInstanceOf(value: Any) = value is T
```

Reified type

```
inline fun <reified T> isInstanceOf(value: Any) = value is T
```

Reified type

```
inline fun <reified T> isInstanceOf(value: Any) = value is T

val result = isInstanceOf<String>(value)
```

Reified type

```
inline fun <reified T> isInstanceOf(value: Any) = value is T

val result = isInstanceOf<String>(value)
```

```
val result =  value is String
```

With reified type you can

- Do type checks and casts (is, !is, as, as?)

With reified type you can

- Do type checks and casts (is, !is, as, as?)
- Use the Kotlin reflection APIs

With reified type you can

- Do type checks and casts (is, !is, as, as?)
- Use the Kotlin reflection APIs
- Get the corresponding java.lang.Class (::class.java)

With reified type you can

- Do type checks and casts (is, !is, as, as?)
- Use the Kotlin reflection APIs
- Get the corresponding java.lang.Class (::class.java)
- Call other generic functions and pass them type parameter

With reified type you can not

- Create new instances of the class specified as a type parameter

With reified type you can not

- Create new instances of the class specified as a type parameter
- Call methods on the companion object of the type parameter class

With reified type you can not

- Create new instances of the class specified as a type parameter
- Call methods on the companion object of the type parameter class
- Use a non-reified type parameter as a type argument when calling a function with a reified type parameter

With reified type you can not

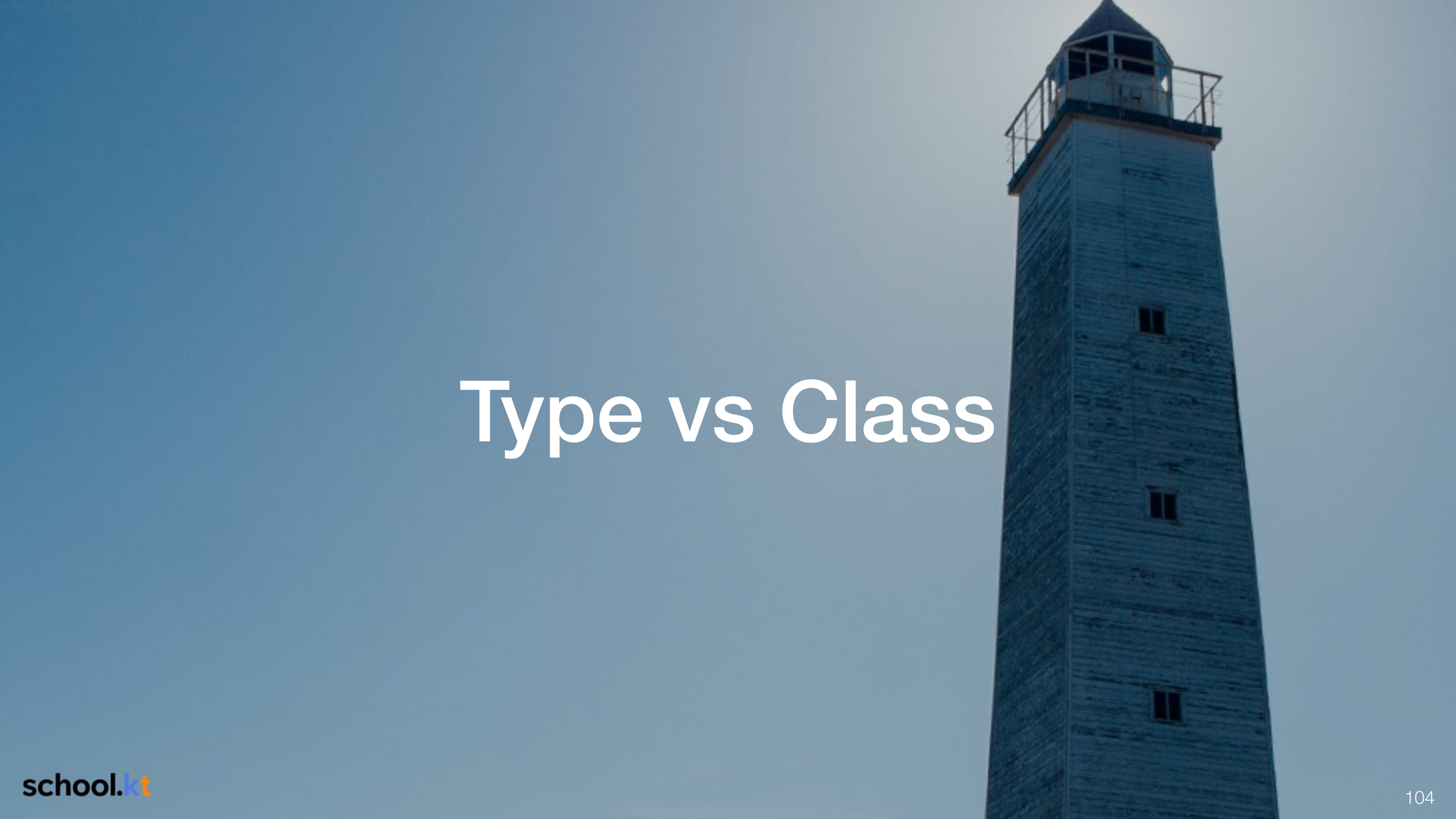
- Create new instances of the class specified as a type parameter
- Call methods on the companion object of the type parameter class
- Use a non-reified type parameter as a type argument when calling a function with a reified type parameter
- Mark type parameters of classes, properties, or non-inline functions as reified

What about Java?

- Types are erased anyway

- Functions with reified types are not accessible from Java code

Type vs Class



Type vs Class

```
val name: String
```

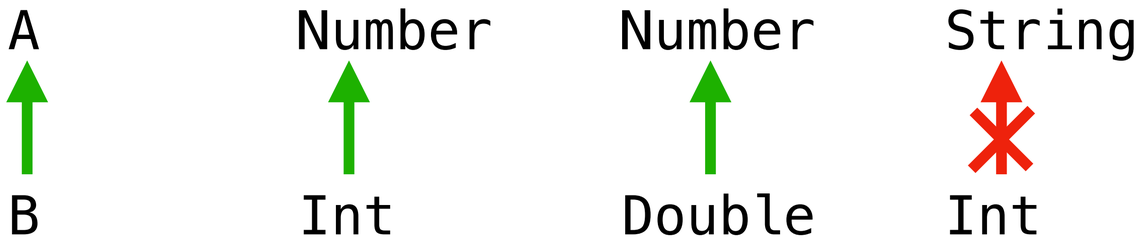
Type vs Class

```
val name: String

val name: String?
```

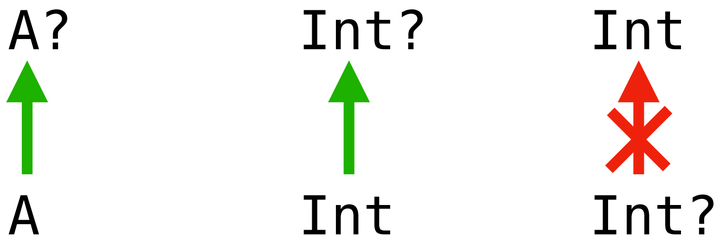
Type vs Class

```
List<String>

List<String?>

List<List<Int>>

List<List<Int>>?
```

Type vs Class



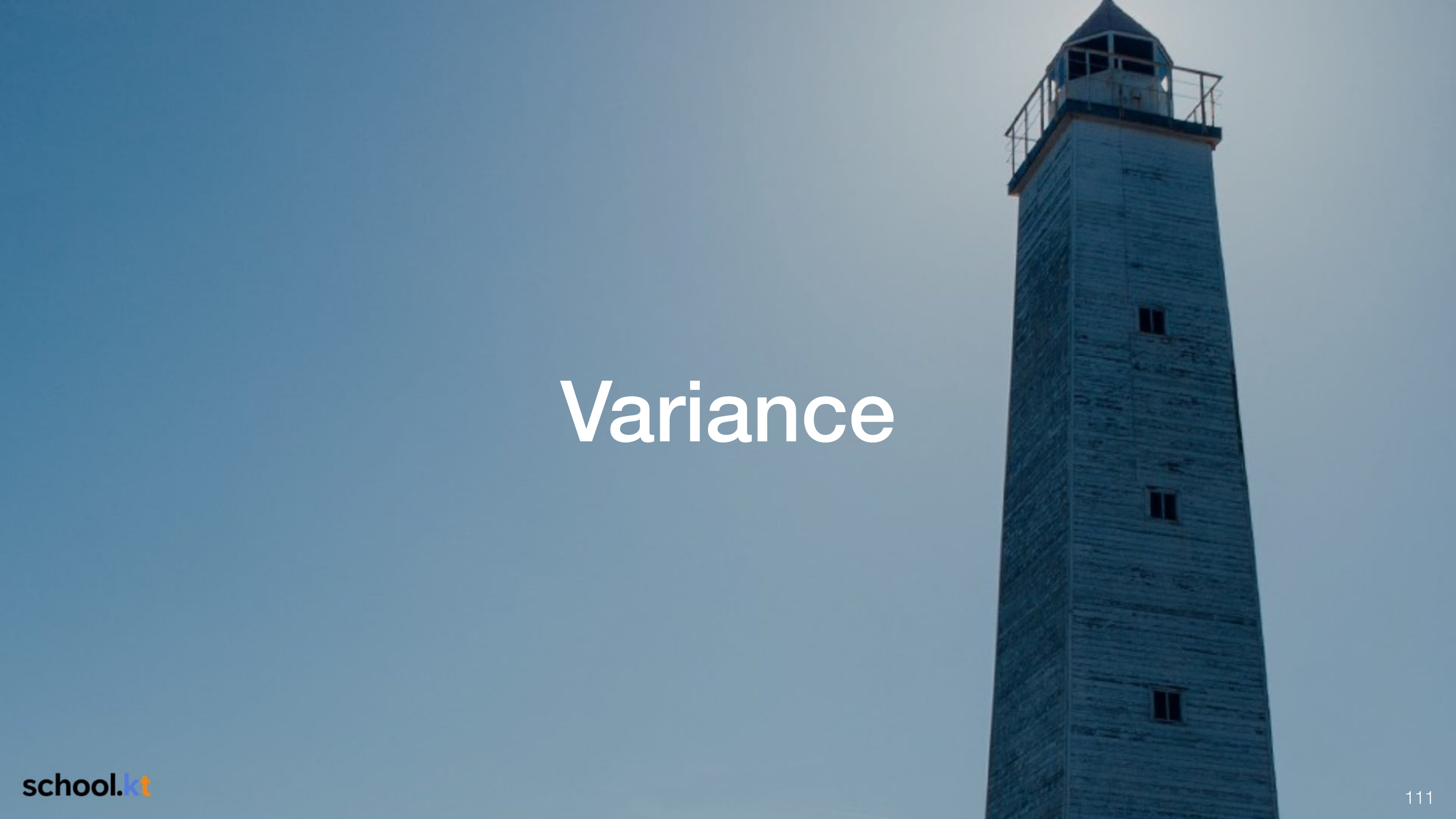
Type vs Class



Type vs Class

`List<A>`	`MutableList<A>`	`A`
?	?	
`List<B>`	`MutableList<B>`	`B`

Variance



Variance

```
fun printElements(list: List<Any>) {
    println(list.joinToString())
}
```

Variance

```
fun printElements(list: List<Any>) {
    println(list.joinToString())
}

printElements(listOf("a", "b", "c"))

>>> a,b,c
```

Variance

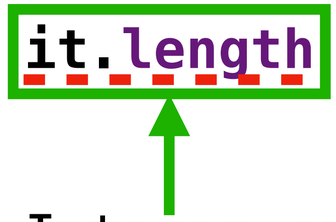
```
fun addValue(list: MutableList<Any>) {
    list.add(42)
}
```

Variance

```
fun addValue(list: MutableList<Any>) {
    list.add(42)
}

val strings = mutableListOf("a", "b", "c")

addValue(strings)

println(strings.maxBy { it.length })
```

Variance

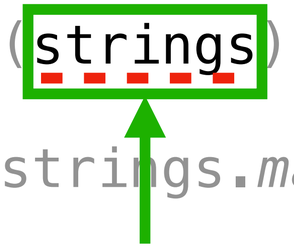
```
fun addValue(list: MutableList<Any>) {
    list.add(42)
}

val strings = mutableListOf("a", "b", "c")

addValue(strings)

println(strings.maxBy { it.length })
```



```
    ClassCastException: Integer cannot be cast to String
```

Variance

```
fun addValue(list: MutableList<Any>) {
    list.add(42)
}

val strings = mutableListOf("a", "b", "c")

addValue(strings)

println(strings.maxBy { it.length })
```



```
Type mismatch.
Required: MutableList<Any>
Found:    MutableList<String>
```

Variance

`List<Any>` `MutableList<Any>`

`List<String>` `MutableList<String>`

Variance

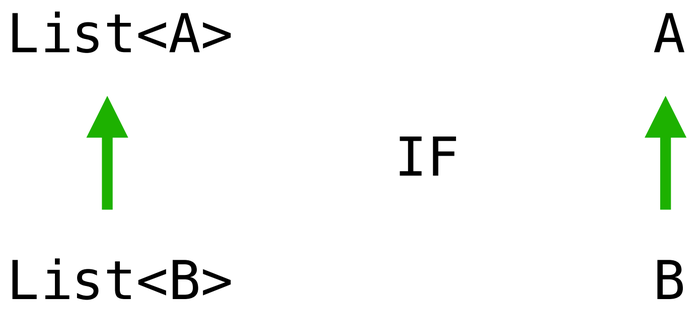
- Invariance
- Covariance
- Contravariance

Invariance

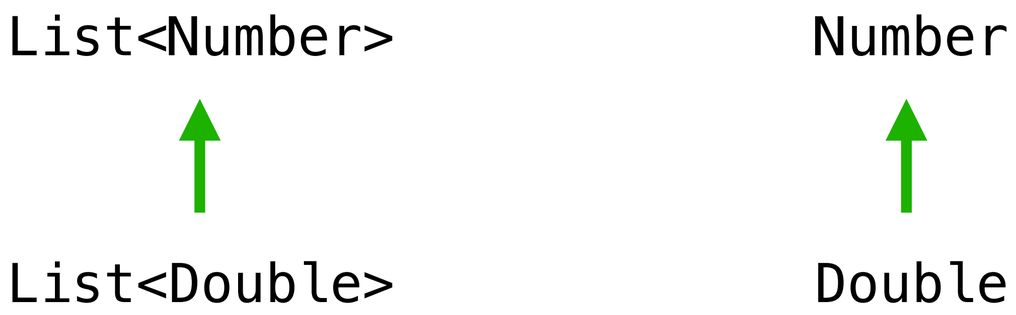
`MutableList<A>`

`MutableList<B>`

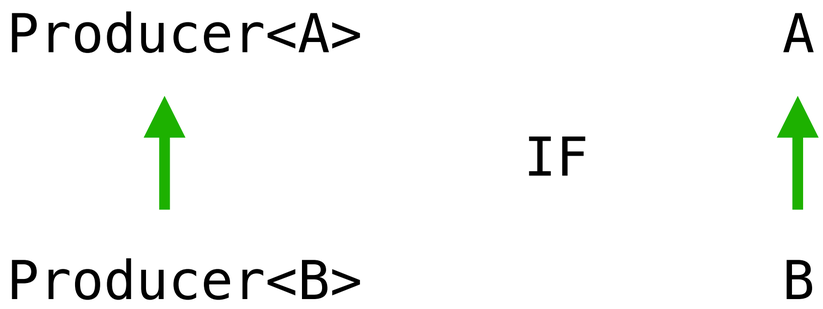
Covariance



Covariance



Covariance



Covariance

```
interface Producer<out T> {
    fun produce(): T
}
```

Covariance

```
interface Producer<out T> {
    fun produce(): T
}
```

Covariance

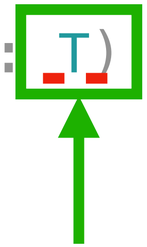
```
interface Producer<out T> {
    fun produce(): T
}
```

Covariance

```
interface Producer<out T> {
    fun produce(): T
    fun consume(value: T)
}
```

Covariance

```
interface Producer<out T> {
    fun produce(): T
    fun consume(value: T)
}
```



```
Type parameter T is declared as 'out' but occurs in 'in' position
```

Covariance

```
interface Developer {
    fun doWork()
}
```

Covariance

```
interface Developer {
    fun doWork()
}

class Team<T : Developer> {
    val size: Int
    fun getDeveloper(index: Int): T
}
```

Covariance

```
interface Developer {
    fun doWork()
}

class Team<T : Developer> {
    val size: Int
    fun getDeveloper(index: Int): T
}

fun startProject(team: Team<Developer>) {
    for (i in 0 until team.size) {
        team.getDeveloper(i).doWork()
    }
}
```

Covariance

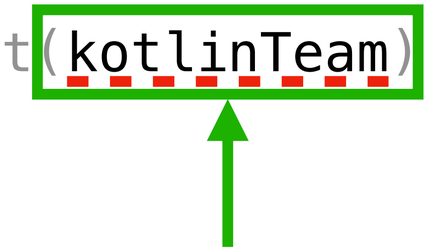
```
fun startProject(team: Team<Developer>) {
    ...
}

class KotlinDev: Developer {
    override fun doWork() = ...
}

val kotlinTeam = Team<KotlinDev>()
```

Covariance

```
fun startProject(team: Team<Developer>) {
    ...
}

class KotlinDev: Developer {
    override fun doWork() = Unit
}

val kotlinTeam = Team<KotlinDev>()

startProject(kotlinTeam)
```

Covariance

```
fun startProject(team: Team<Developer>) {
    ...
}

class KotlinDev: Developer {
    override fun doWork() = Unit
}

val kotlinTeam = Team<KotlinDev>()

startProject(kotlinTeam)
```



Type mismatch

Covariance

```
class Team<T : Developer> {
    val size: Int
    fun getDeveloper(index: Int): T
}
```

Covariance

```
class Team<out T : Developer> {
    val size: Int
    fun getDeveloper(index: Int): T
}
```

Covariance

```
class Team<out T : Developer> {
    val size: Int
    fun getDeveloper(index: Int): T
}

class KotlinDev: Developer {
    override fun doWork() = Unit
}

val kotlinTeam = Team<KotlinDev>()

startProject(kotlinTeam)
```

Covariance

```
class Team<out T : Developer>(
        vararg developers: T
) {
    val size: Int
    fun getDeveloper(index: Int): T
}
```

Covariance

```
class Team<out T : Developer>(
        var developer: T
) {
    val size: Int
    fun getDeveloper(index: Int): T
}
```

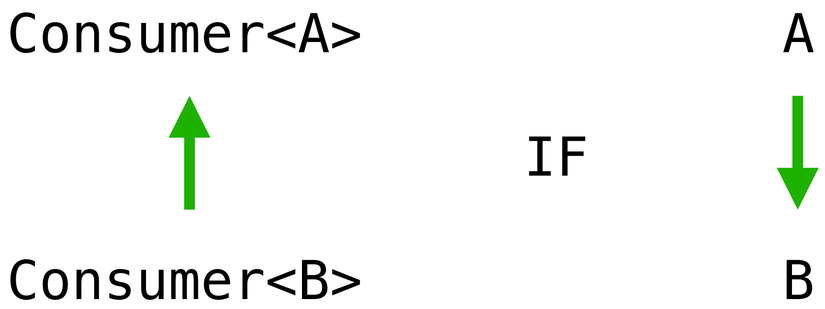
Covariance summary

- The subtyping is preserved (Team<KotlinDeveloper> is a subtype of Team<Developer>).

Covariance summary

- The subtyping is preserved (Team<KotlinDeveloper> is a subtype of Team<Developer>).
- T can be used only in out positions.

Contravariance



Contravariance

```
interface Comparator<in T> {
    fun compare(e1: T, e2: T): Int {}
}
```

Contravariance

```
interface Comparator<in T> {
    fun compare(e1: T, e2: T): Int {}
}
```

Contravariance

```
interface Comparator<in T> {
    fun compare(e1: T, e2: T): Int {}
}
```

Contravariance

```
val anyComparator = Comparator<Any> {
    e1, e2 -> e1.hashCode() - e2.hashCode()
}
```

Contravariance

```
val anyComparator = Comparator<Any> {
    e1, e2 -> e1.hashCode() - e2.hashCode()
}

val strings: List<String> = ...

strings.sortedWith(anyComparator)
```

Covariance vs Contravariance

Declaration site variance

```
interface Producer<out T> {
    fun produce(): T
}

interface Comparator<in T> {
    fun compare(e1: T, e2: T): Int {}
}
```

Use-site variance

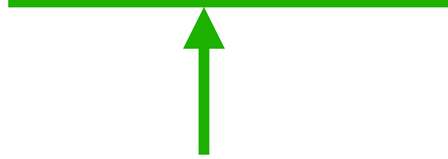
```
fun printElements(list: MutableList<Any>) {
    list.forEach {
        println(it)
    }
}
```

Use-site variance

```
fun printElements(list: MutableList<Any>) {
    list.forEach {
        println(it)
    }
}

printElements(mutableListOf<String>("aaa"))
```

Use-site variance

```
fun printElements(list: MutableList<Any>) {
    list.forEach {
        println(it)
    }
}

printElements(mutableListOf<String>("aaa"))
```



Type mismatch

Use-site variance

```
fun printElements(list: MutableList<Any>) {
    list.forEach {
        println(it)
    }
}

printElements(mutableListOf<String>("aaa"))
```

Use-site variance

```
fun printElements(list: MutableList<out Any>) {
    list.forEach {
        println(it)
    }
}

printElements(mutableListOf<String>("aaa"))
```

Use-site variance

```
fun printElements(list: MutableList<out Any>) {
    list.forEach {
        println(it)
    }
    list.add("aaa")
}

printElements(mutableListOf<String>("aaa"))
```



```
Out-projected type 'MutableList<out Any>' prohibits the use of
'public abstract fun add(element: E): Boolean defined in
kotlin.collections.MutableList'
```

Star projection

```
fun printElements(list: MutableList<*>) {
    list.forEach { value: Any? ->
        println(value)
    }
}
```

Star projection

```
fun printElements(list: MutableList<out Any?>) {
    list.forEach { value: Any? ->
        println(value)
    }
}
```

What about Java?

- Java does not support declaration-site variance at all

- Java supports use-site variance with meta-symbols (? extends, ? super)

- When you use Kotlin types in Java you can easily make mistakes

Resume

- Generics in Kotlin work mostly like in Java

Resume

- Generics in Kotlin work mostly like in Java
- Types are still erased

Resume

- Generics in Kotlin work mostly like in Java
- Types are still erased
- Sometimes you can avoid it with reified types

Resume

- Generics in Kotlin work mostly like in Java
- Types are still erased
- Sometimes you can avoid it with reified types
- Variance can help with definition of type-subtype relationship

Resume

- Generics in Kotlin work mostly like in Java
- Types are still erased
- Sometimes you can avoid it with reified types
- Variance can help with definition of type-subtype relationship
- Invariance means no relationship

Resume

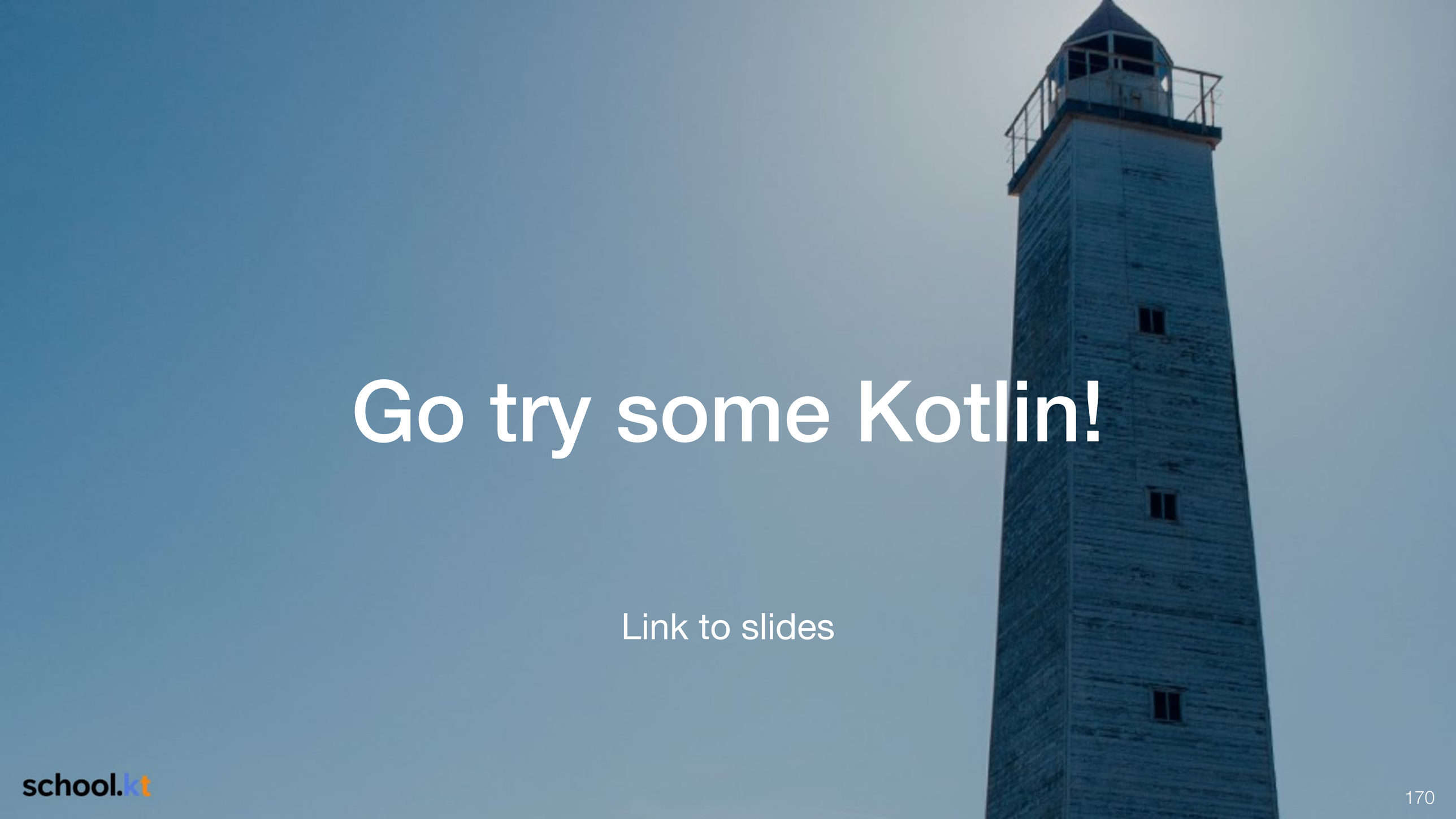
- Generics in Kotlin work mostly like in Java
- Types are still erased
- Sometimes you can avoid it with reified types
- Variance can help with definition of type-subtype relationship
- Invariance means no relationship
- Covariance means straightforward type-subtype relationship

Resume

- Generics in Kotlin work mostly like in Java
- Types are still erased
- Sometimes you can avoid it with reified types
- Variance can help with definition of type-subtype relationship
- Invariance means no relationship
- Covariance means straightforward type-subtype relationship
- Contravariance means inverse order

Resume

- Generics in Kotlin work mostly like in Java
- Types are still erased
- Sometimes you can avoid it with reified types
- Variance can help with definition of type-subtype relationship
- Invariance means no relationship
- Covariance means straightforward type-subtype relationship
- Contravariance means reverse order
- We can use declaration-site variance and use-site variance



Go try some Kotlin!

Link to slides