{ this is Kotlin }

Generics

Tiberiu Tofan

Generics

why

To reuse code in a type safe way

Generics

```
fun Account.withdraw(amount: BigDecimal): Pair<Account, BigDecimal> = when (this) {
    is CurrentAccount -> copy(balance = balance - amount) to amount
    is SavingsAccount -> copy(balance = balance - amount) to amount
    is CreditAccount ->
        if (balance - amount > creditLimit) copy(balance = balance - amount) to amount
        else this to BigDecimal.ZERO
    is TechnicalAccount -> this to BigDecimal.ZERO
}

val account = SavingsAccount("GB00...", 10_000.toBigDecimal(), "0.024".toBigDecimal())
println(account.interest)

val (updatedAccount, _) = account.withdraw(10.toBigDecimal())
println(updatedAccount.interest)
```

compilation error: Unresolved reference: interest

we know that updatedAccount is SavingsAccount, but the compiler doesn't

Generics

to save the day

```kotlin
fun <T : Account> T.withdraw(amount: BigDecimal): Pair<T, BigDecimal> = run {
    val account = this as Account
    when (account) {
        is CurrentAccount -> account.copy(balance = balance - amount) as T to amount
        is SavingsAccount -> account.copy(balance = balance - amount) as T to amount
        /* ommited */
    }
}

val account = SavingsAccount("GB00...", 10_000.toBigDecimal(), "0.024".toBigDecimal())
println(account.interest)

val (updatedAccount, _) = account.withdraw(10.toBigDecimal())
println(updatedAccount.interest) ✓
```

Generics

type erasure

```
public <T> T readValue(String content, Class<T> valueType)
```

```
val json = """
    {
      "name": "Parzival",
      "password": "qwerty123"
    }
“"".trimIndent()

val client: User = mapper.readValue(json, User::class.java)
```

Duplicated type information

Generics

type erasure

```
public <T> T readValue(String content, Class<T> valueType)
```

```
val json = """
    [{
      "name": "Parzival",
      "password": "qwerty123"
    }]
“"".trimIndent()

val clients: List<User> = mapper.readValue(json, List<User>::class.java)
```

There’s no such thing as `List<User>::class` because the type information is lost after compilation

Generics

type erasure

```
public <T> T readValue(String content, TypeReference valueTypeRef)


val json = """
    [{
      "name": "Parzival",
      "password": "qwerty123"
    }]
“"".trimIndent()

val clients: List<User> = mapper.readValue(json,
                                        object : TypeReference<List<User>>() {})
```

The runtime needs a concrete implementation with the generic type fixed

Generics

reified

```kotlin
inline fun <reified T> ObjectMapper.readValue(content: String): T =
    readValue(content, object : TypeReference<T>() {})
```

```kotlin
val client: User = mapper.readValue(json)

val client = mapper.readValue<User>(json)

val clients: List<User> = mapper.readValue(json)

val clients = mapper.readValue<List<User>>(json)
```

Why do functions that used reified types have to be inline?

Generics

invariance

```
public class Array<T>

fun zeroOut(ns: Array<Number>) {
    for (i in ns.indices) {
        ns[i] = 0.0
    }
}

val ns: Array<Int> = arrayOf(1, 2, 3, 4)

zeroOut(ns)
```

[0.0, 0.0, 0.0, 0.0]

The function modifies the Array

Type mismatch.
Required: Array<Number>
Found: Array<Int>

`Array<Int>` is not a subtype of `Array<Number>`
(`Array` is **invariant** in T)

Generics

use site variance / projections

```
public class Array<T>

fun joinIt(a: Array<Any>): String = a.joinToString { it.toString() }
```

The function just reads from the Array

```
val ns: Array<Int> = arrayOf(1, 2, 3, 4)

val concat = joinIt(ns)
```

Type mismatch.
Required: Array<Any>
Found: Array<Int>

Generics

use site variance / projections

```
public class Array<T>

fun joinIt(a: Array<out Any>): String = a.joinToString { it.toString() }
```

The function just reads from the Array

```
val ns: Array<Int> = arrayOf(1, 2, 3, 4)

val concat = joinIt(ns) 1, 2, 3, 4
```

Generics

use site variance / projections

```
public class Array<T>
```

The function just reads from the Array

```
fun joinIt(a: Array<*>): String = a.joinToString { it.toString() }
```

```
*
out Any
in Nothing
```

```
val ns: Array<Int> = arrayOf(1, 2, 3, 4)

val concat = joinIt(ns)
```

1, 2, 3, 4

Generics

declaration site variance

```
interface List<out E>
```

declaration site covariance

```
fun joinIt(ns: List<Number>): String = ns.joinToString { it.toString() }
```

we didn't specify use site variance

```
val ns: List<Int> = listOf(1, 2, 3, 4)

val concat = joinIt(ns)
```

1, 2, 3, 4

Generics

declaration site variance

```kotlin
interface Comparable<in T>
```

declaration site contravariance

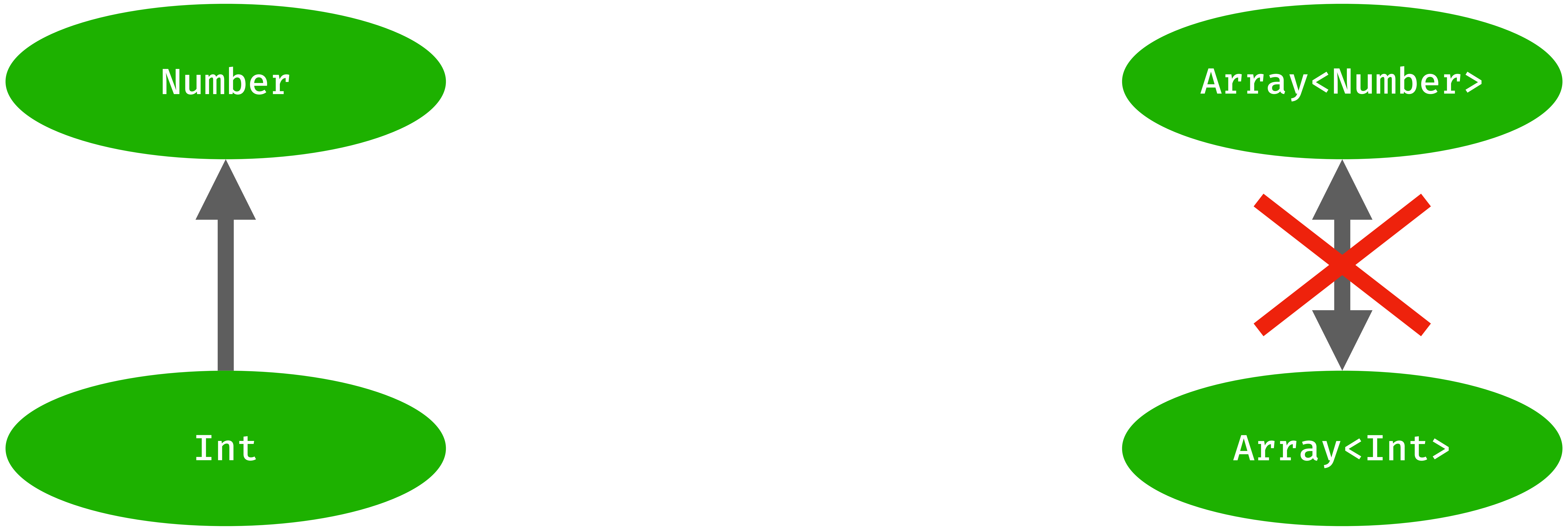
```kotlin
val comparableOfNumber: Comparable<Number> = TODO()
val comparableOfDouble: Comparable<Double> = comparableOfNumber
```

✓

Generics

invariance

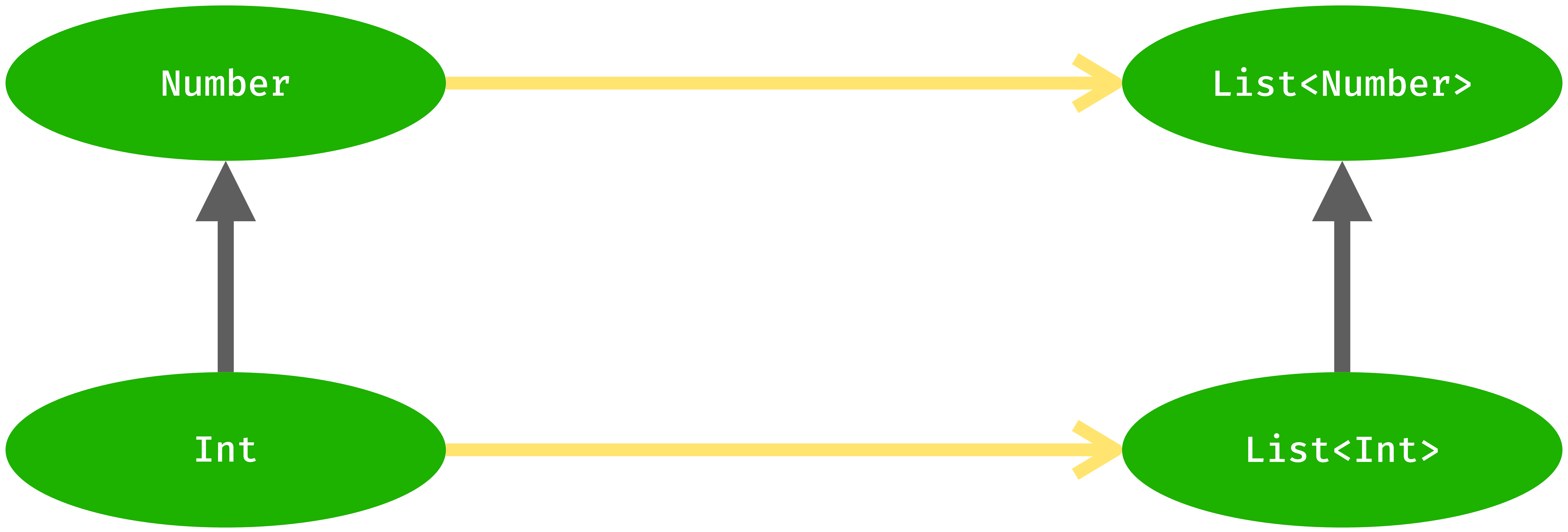
```
interface Array<T>
```



Generics

covariance: `out`

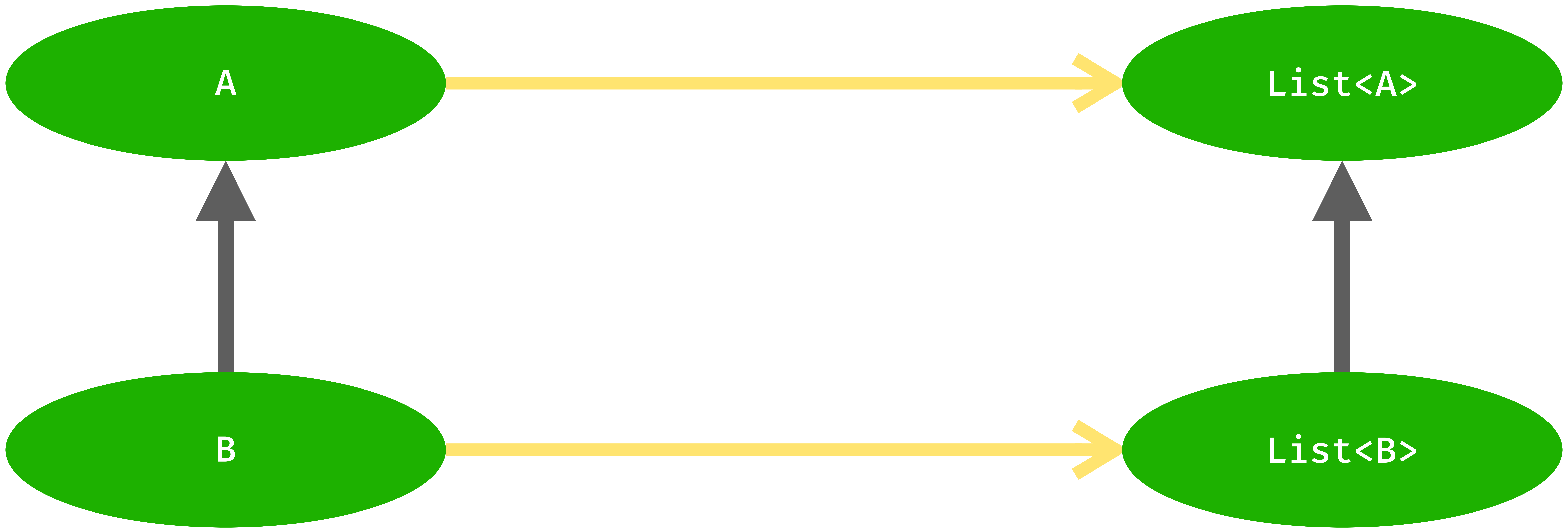
```
interface List<out E>
```



Generics

covariance: `out`

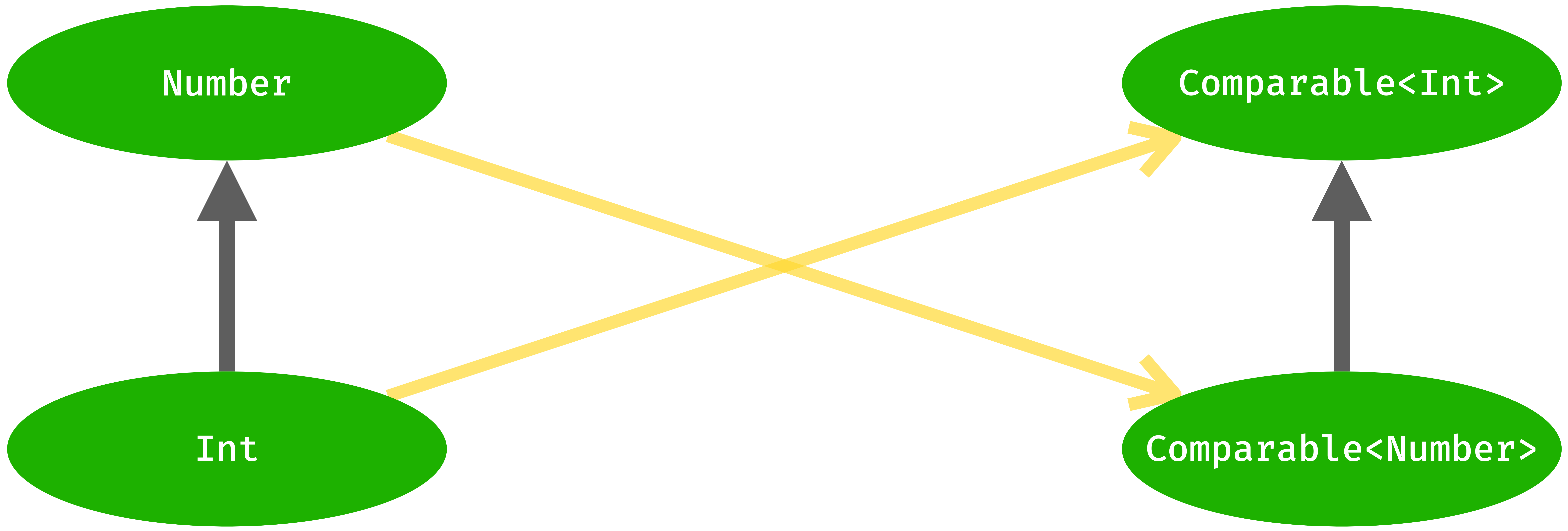
```
interface List<out E>
```



Generics

contravariance: `in`

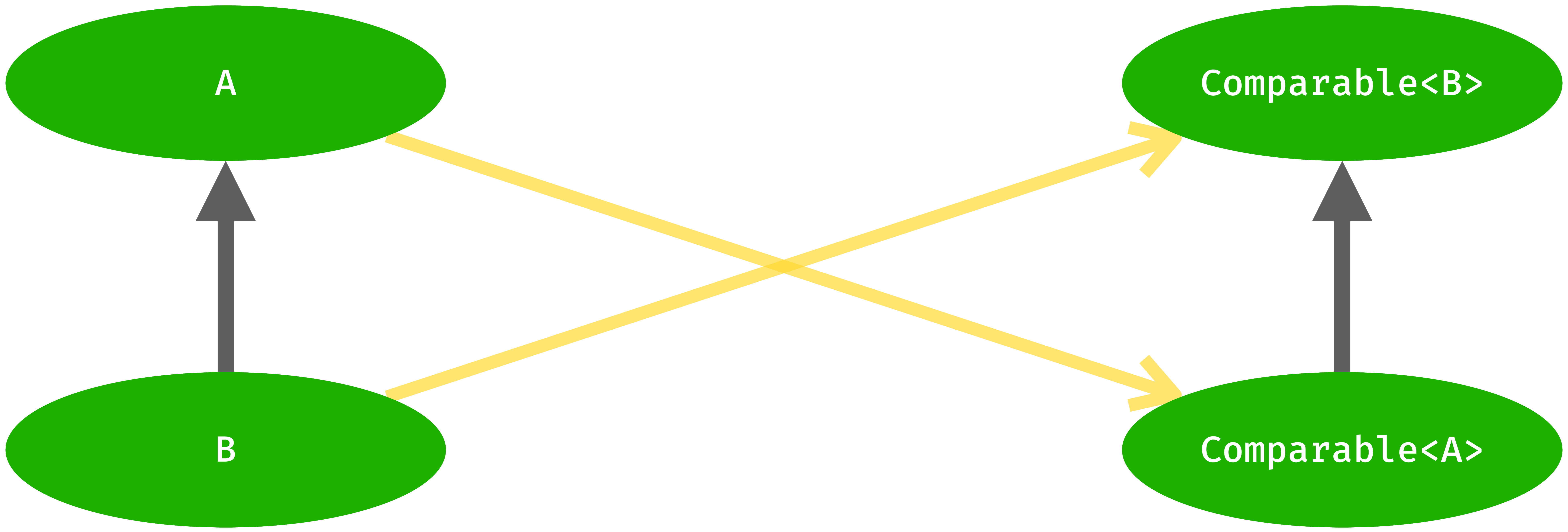
```
interface Comparable<in T>
```



Generics

contravariance: `in`

```
interface Comparable<in T>
```

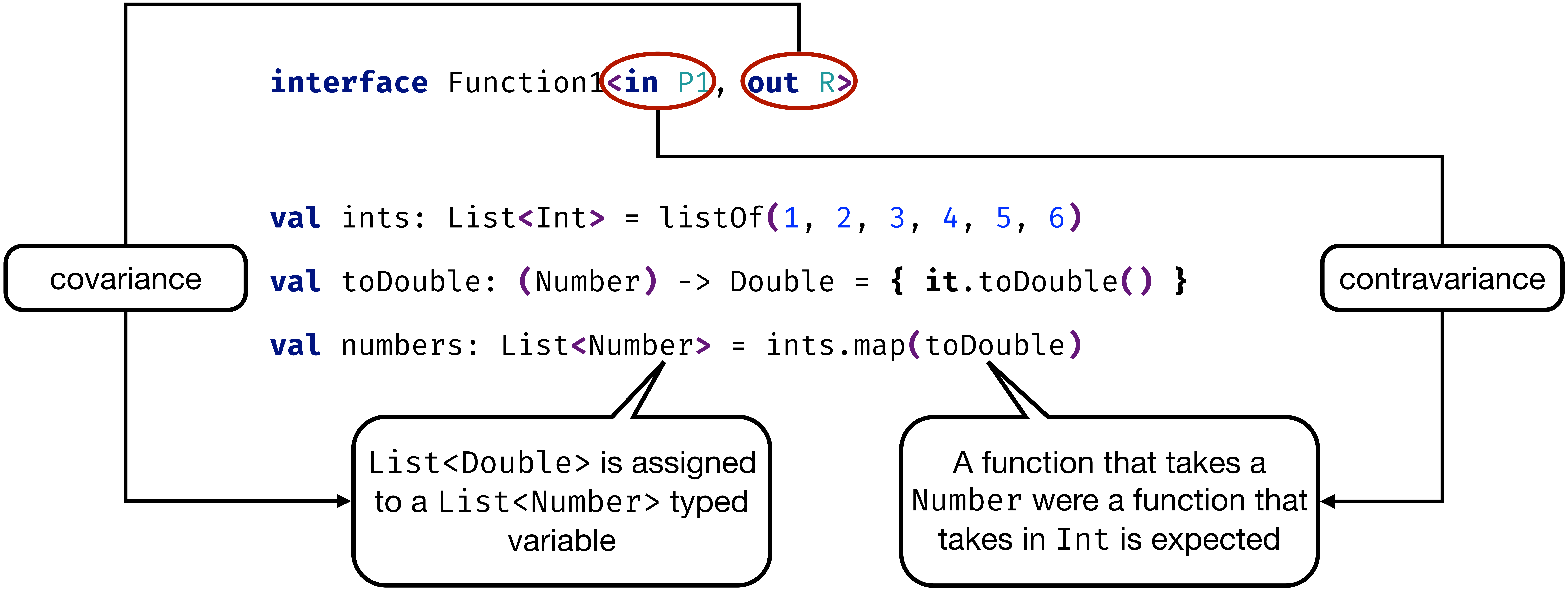


Could a **class/interface** be both **covariant and contravariant?**

Generics

```
interface Function1<in P1, out R>

val ints: List<Int> = listOf(1, 2, 3, 4, 5, 6)
val toDouble: (Number) -> Double = { it.toDouble() }
val numbers: List<Number> = ints.map(toDouble)
```


covariance
contravariance
List<Double> is assigned to a List<Number> typed variable
A function that takes a Number were a function that takes in Int is expected