Kotlin - Generics and Sequences

Jacky Xu

RBC One Kotlin Study Group

April 25, 2018

Generics

- Allows a class or function to operate on objects of various types
- Provides compile-time type safety
- Not much different from generics in Java
- Kotlin compiler can usually infer the parameterized type from the values provided

```
val list1 = listOf(1, 2, 3)
val list2 = listOf("1", "2", "3")
```

Generics

- Generic class:

```
class Box<T>(val contents: List<T>)
```

- Generic function:

```
fun <T> take(box: Box<T>) { ... }
```

- Generic interface:

```
interface Formatter<T> { ... }
```

Upper Bounds

- We can specify upper bounds on type parameters

```
// In Java:
class Box<T extends Fruit> { ... }

// In Kotlin:
class Box<T: Fruit>(val contents: List<T>)
```

Upper Bounds

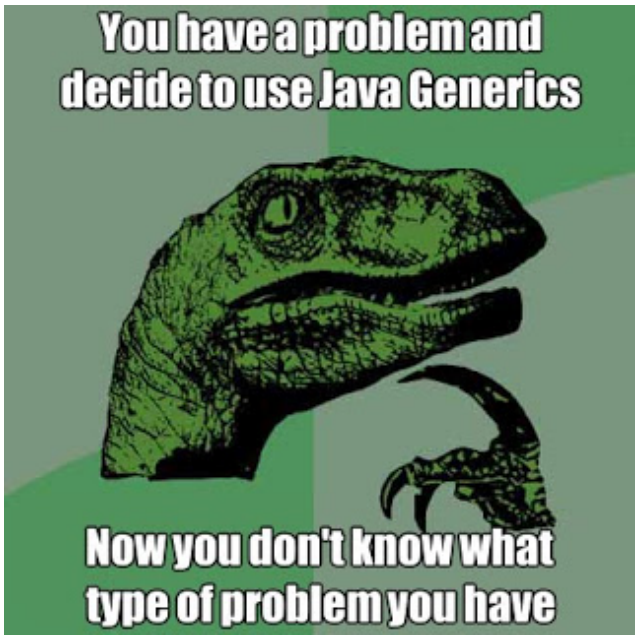
- If there are multiple upper bounds, we must specify them by the `where` keyword:

```
class Box<T>(val contents: List<T>)
    where T: Fruit, T: Round
```

- A type parameter with no upper bound specified will have the upper bound of `Any?`
- Unlike in Java, there are no lower bounds in Kotlin

Type Erasure

- In Java, type information is lost at runtime to maintain backwards compatibility with previous versions of JVM.
- Since Kotlin runs on the JVM, its types are also erased.



Type Erasure

- Kotlin compiler will try to infer type parameter at compile time:

```
val strings = listOf("1", "2", "3")
println(strings is List<String>) // true
```

- If type parameter cannot be inferred at compile time, will throw compile time error:

```
val anys = listOf("1", 2, 3.0)
println(anys is List<String>) // error
```

Type Erasure

- We can use **star projection** if we don't care about the type parameter:

```
val anys = listOf("1", 2, 3.0)
println(anys is List<*>) // true
```

Reified Types

- Kotlin has a way of retaining type information at runtime, called **reification**
- Reification only works:
  - In an **inlined function**
  - If the generic type is preceded by the `reified` keyword

```
inline fun <reified T> hasType(box: Box<*>) =
    box.contents.any { it is T }
```

Reified Types

- How we **can** use reified type parameters:
  - In type checks and casts (is, !is, as, as?)
  - To use the Kotlin reflection APIs (::class)
  - To get the corresponding java.lang.Class (::class.java)
  - As a type argument to call other functions

```
inline fun <reified T> List<T>.printType() {
    println("This list contains elements " +
        "of type ${T::class}")
}
```

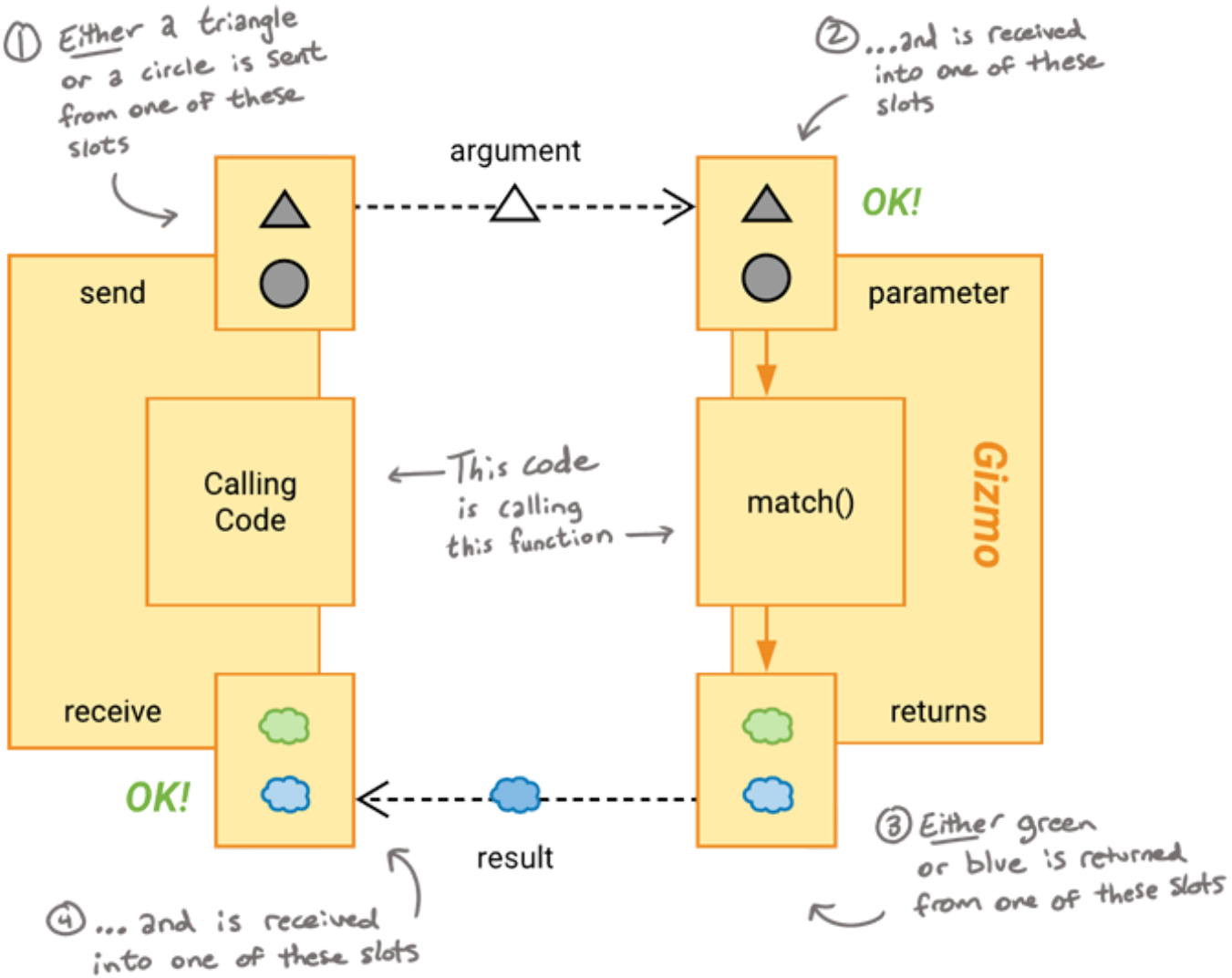
Reified Types

- How we **can't** use reified type parameters:
  - 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

Variance

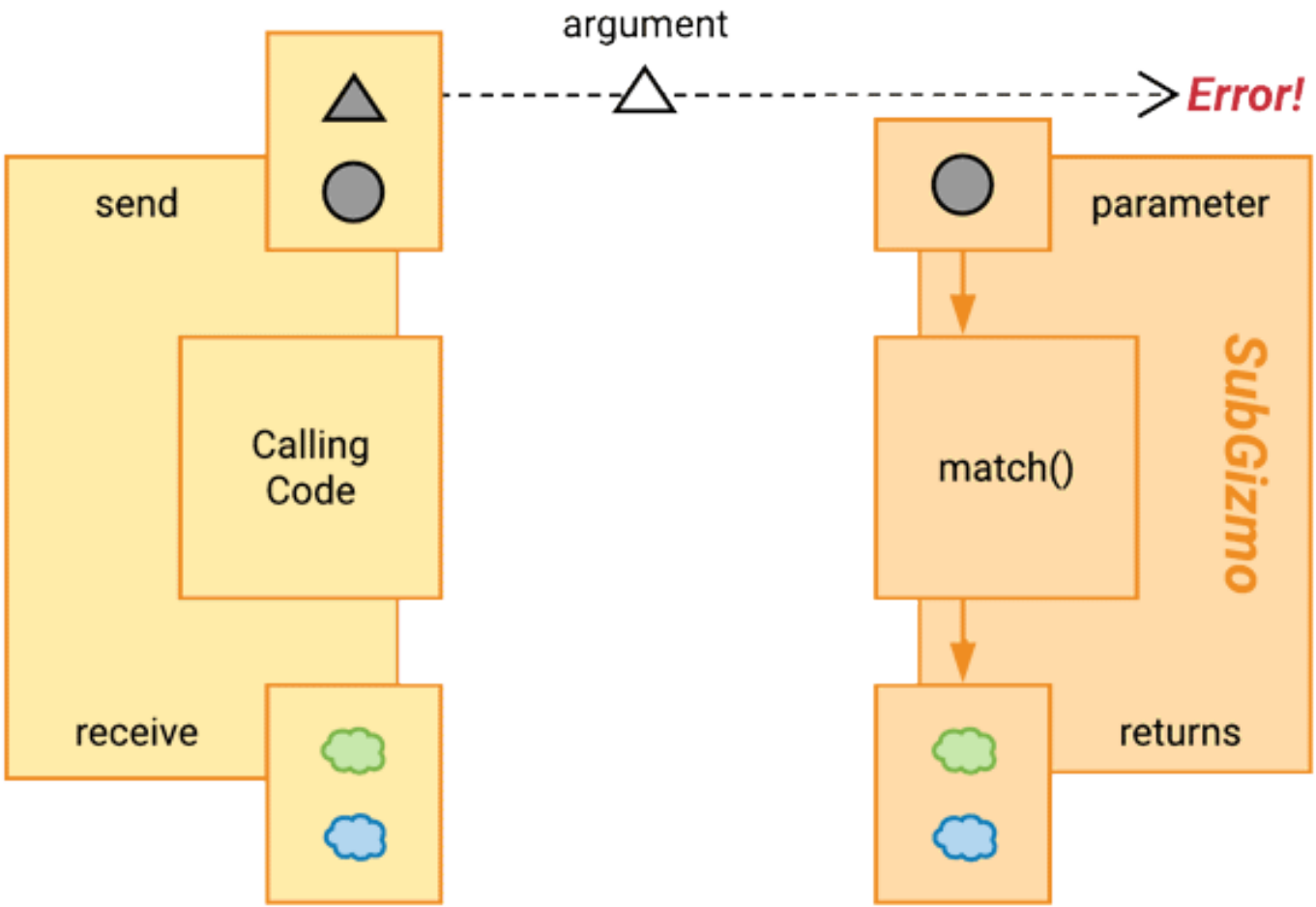
- Liskov Substitution Principle (LSP): A subtype can be substituted for its supertype without altering expected behavior.

Variance

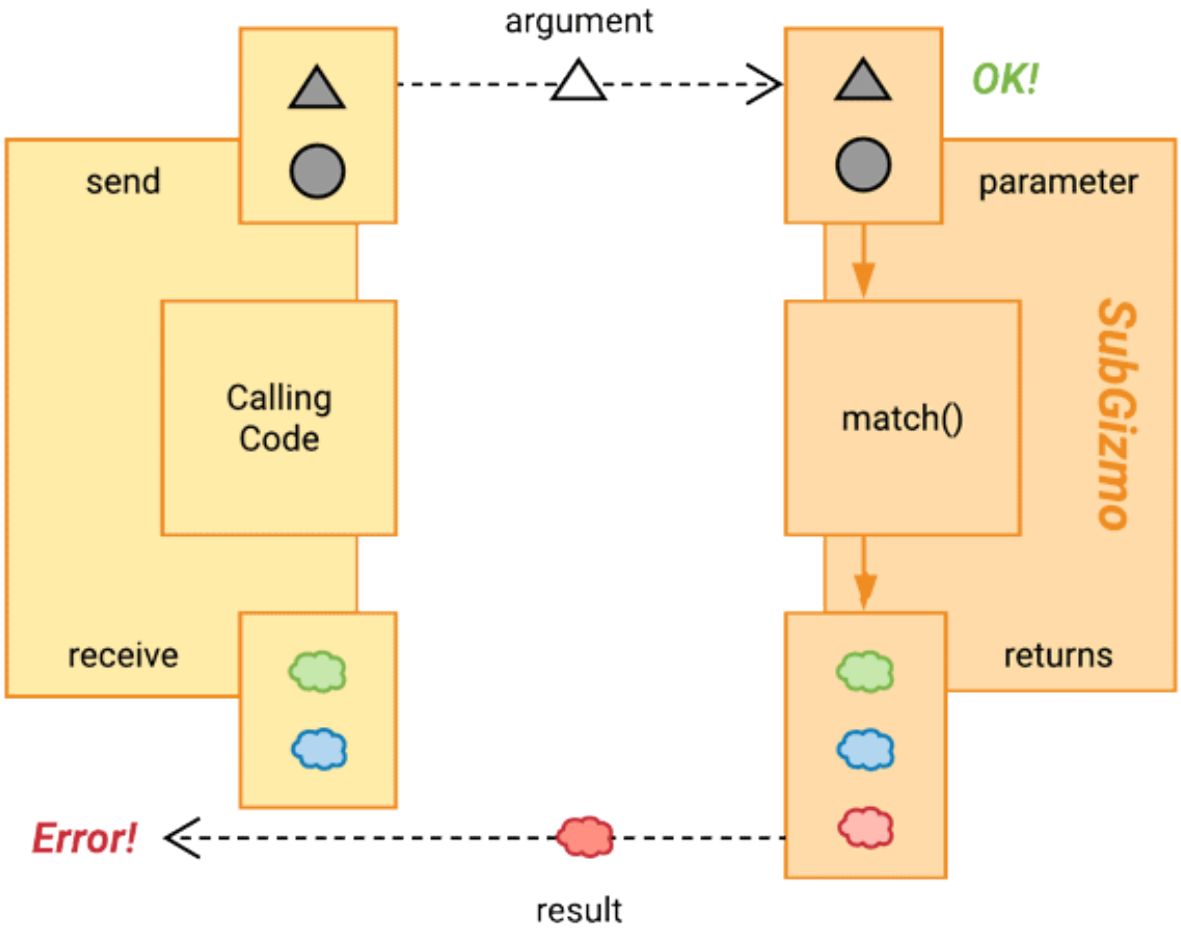


Source: https://typealias.com/guides/illustrated-guide-covariance-contravariance

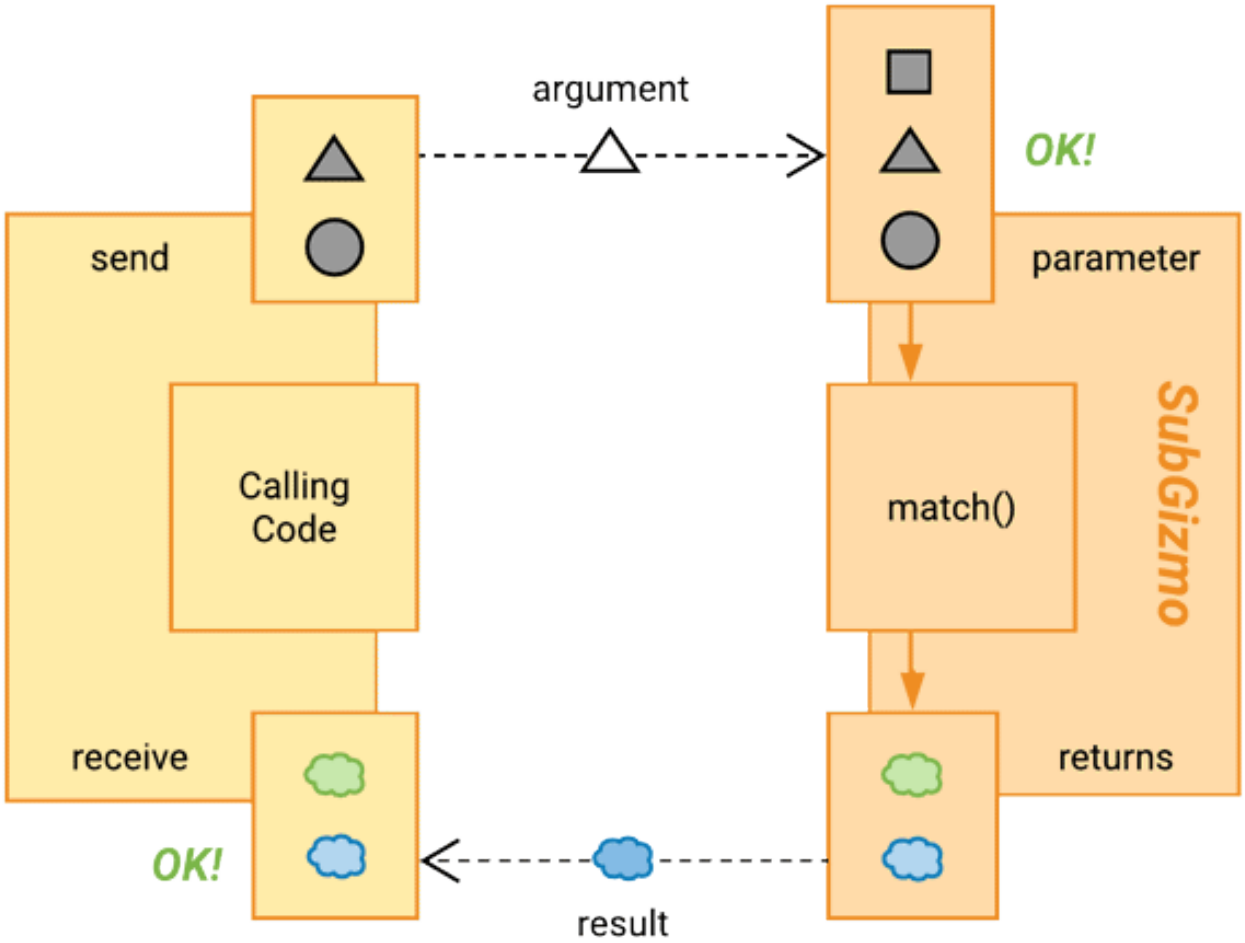
Variance


argument
Error!
send
parameter
Calling
Code
match()
SubGizmo
receive
returns

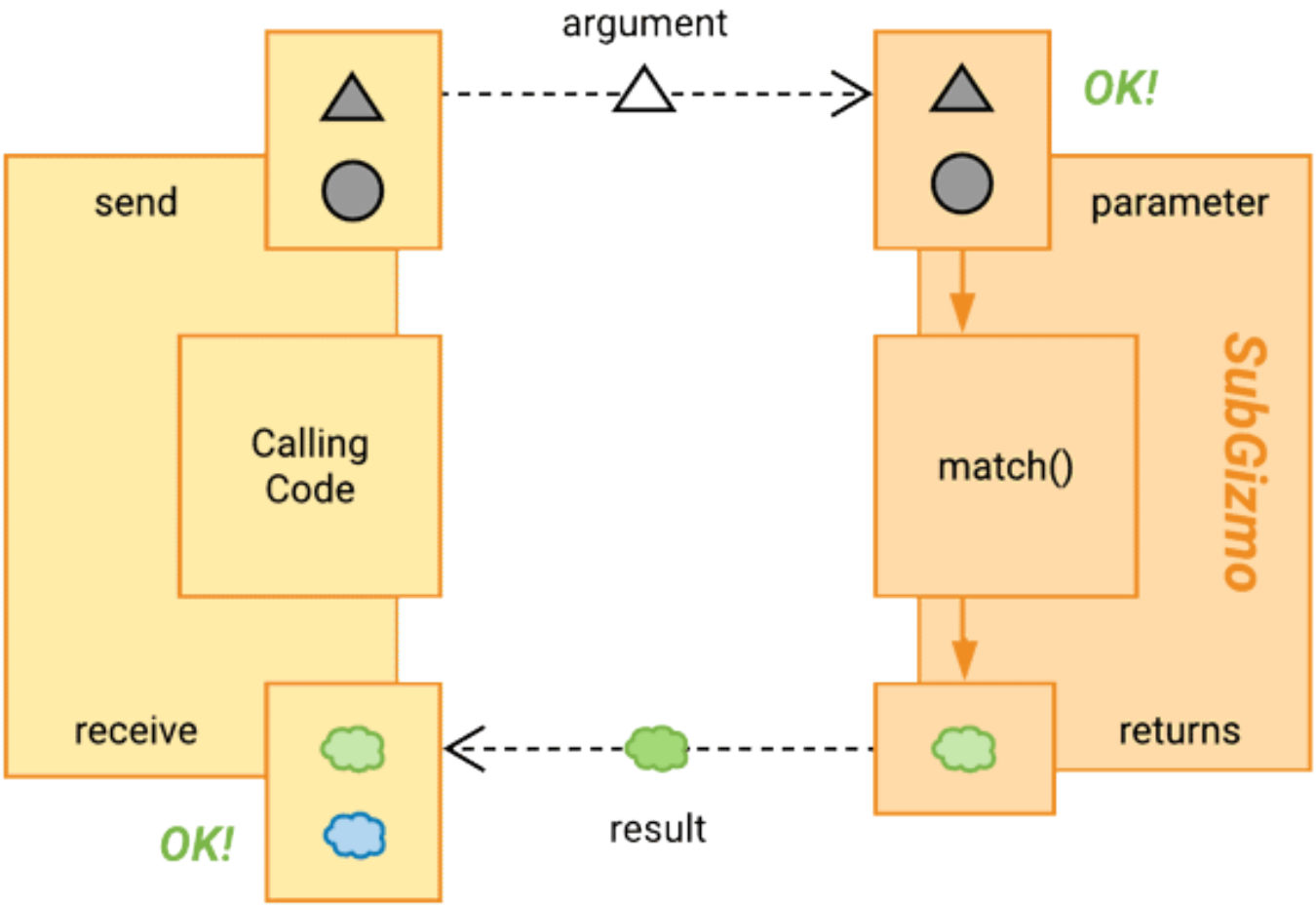
Variance


argument
OK!
send
parameter
Calling Code
match()
SubGizmo
receive
returns
Error!
result

Variance


argument
OK!
send
parameter
Calling Code
match()
SubGizmo
receive
returns
OK!
result

Variance

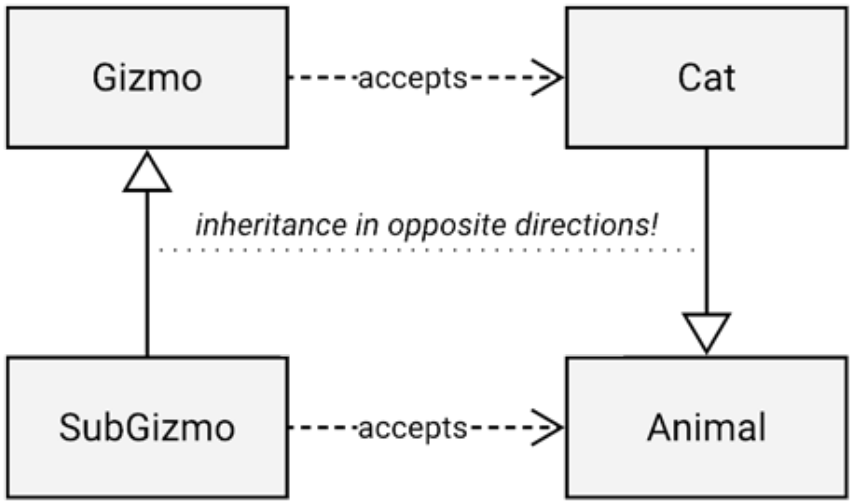

argument
OK!
send
parameter
Calling Code
match()
SubGizmo
receive
returns
OK!
result

Rules of Variance

1. A subtype must **accept at least** the same range of types as its supertype declares.
2. A subtype must **return at most** the same range of types as its supertype declares.

Contravariance

- A subtype must **accept at least** the same range of types as its supertype declares.
- This relationship is called **contravariance**



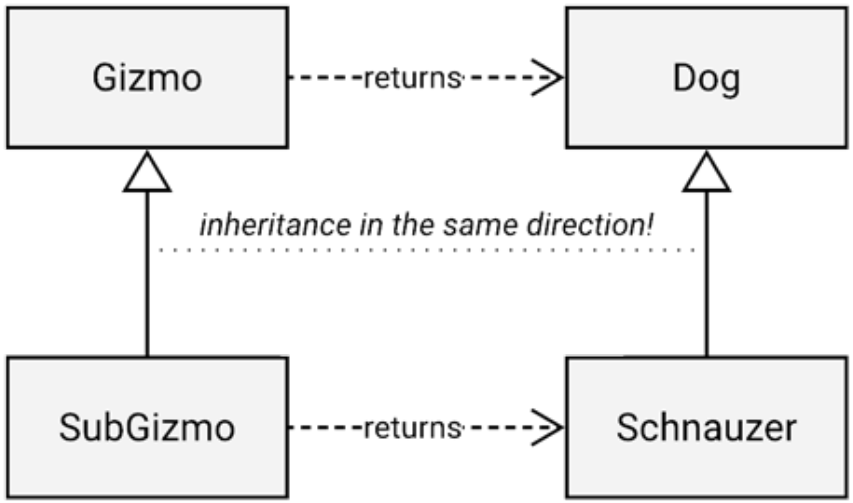
Contravariance

- We specify a type parameter of a class or interface to be **contravariant** by prepending the `in` keyword.
- A contravariant type can only be used as function argument types; it can never be used as a function return type. (why?)
- A contravariant type is said to be in the **in** position.

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}
```

Covariance

- A subtype must **return at most** the same range of types as its supertype declares.
- This relationship is called **covariance**



Covariance

- We specify a type parameter of a class or interface to be **covariant** by prepending the `out` keyword.
- A contravariant type can only be used as a function return type; it can never be used as a function argument type (why?)
- A covariant type is said to be in the **out** position.

```
data class Pair<out A, out B>(
    val first: A, val second: B)
fun <T> Pair<T, T>.toList(): List<T>
    = listOf(first, second)
```

Invariance

- By default a type parameters are in neither the in nor the out position.
- An invariant type can be used in both function arguments and function return types.
- A covariant type cannot be the type of a var property in a primary constructor of a class.
- A contravariant type cannot be the type of any property in a primary constructor of a class.

Type Projection

- All of the variance described above with the 'in' and 'out' keywords is called **declaration-site variance**
- In Kotlin we can also use **use-site variance**, also called **type projection**
- When the `in` or `out` keyword is specified in a function declaration, then the type passed in is treated as if it were contravariant or covariant, respectively.

`? super T <==> in T`
`? extends T <==> out T`

Sequences

- A **sequence** in Kotlin is pretty much the same as a stream in Java 8.
- Kotlin re-implemented this because Java streams aren't available in every platform.
  - On Android in the past, Java 8 is not fully supported.
- Sequences have a greater range of supported operations than Java streams and are less verbose.
  - Eg. filterIsInstance(), zip(), and associate()
- We can perform all stream operations on any collection without converting them to a stream first.

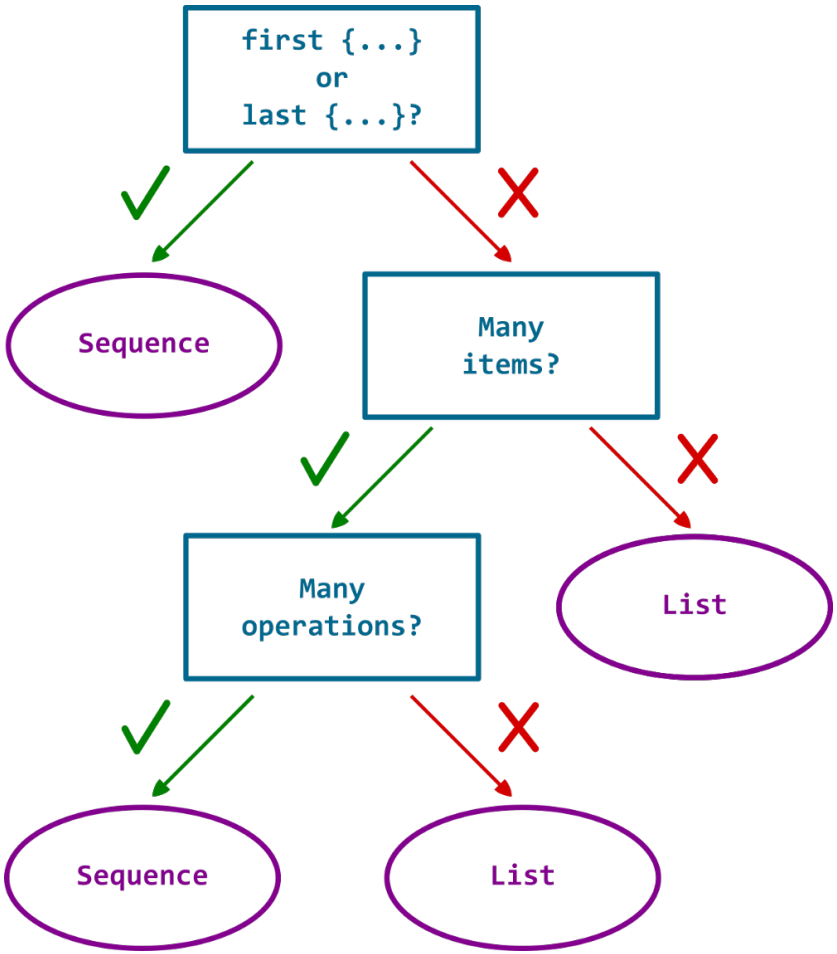
Using Sequences

- As with streams, sequences are evaluated **lazily**, whereas collections are evaluated **eagerly**.
- Two types of operations:
  - **Intermediate operation**: returns another sequence, which is produced lazily.
  - **Terminal operation**: returns a concrete collection or value, and evaluates all operations previously defined in an optimized manner.
- Unlike in Java, sequences can be iterated multiple times (unless specified in the docs).
  - We can constrain a sequence to be only iterable once with `constrainOnce()`.

Using Sequences

- When we **should** use sequences:
  - When dealing with large collections with many operations.
  - When the number of items is infinite or unknown.
  - When processing a large number of items with some filtering operation. (eg. first{} or last{})
- When we **should not** use sequences:
  - When dealing with small collections.
  - When passing/returning the intermediate results to functions.
  - When you need to access only the nth item.

Using Sequences



Source: https://proandroiddev.com/sequences-a-pragmatic-approach-9d4296086a9d

Creating Sequences

- Any iterable can be converted to a sequence using the `asSequence()` method.
- The `generateSequence()` function and its variants which are helpers for creating sequences.
- The `buildSequence()` function can be used to yield values in the sequence, similar to how Typescript's generators are implemented.
  - This feature is experimental.

```
val naturalNumbers: Sequence<Int>
    = generateSequence(0) {it + 1}
```

Resources

- Assignment: Udemy's Kotlin challenges (Round Five)
- Additional problems are in the git repository of this session, under the `com.rbc.rbcone.studygroup.kotlin` package.
  - Six problems total.
  - Solutions are in the `solutions` branch.
- Slides and code examples are in the git repository.

Conclusion

Thank you