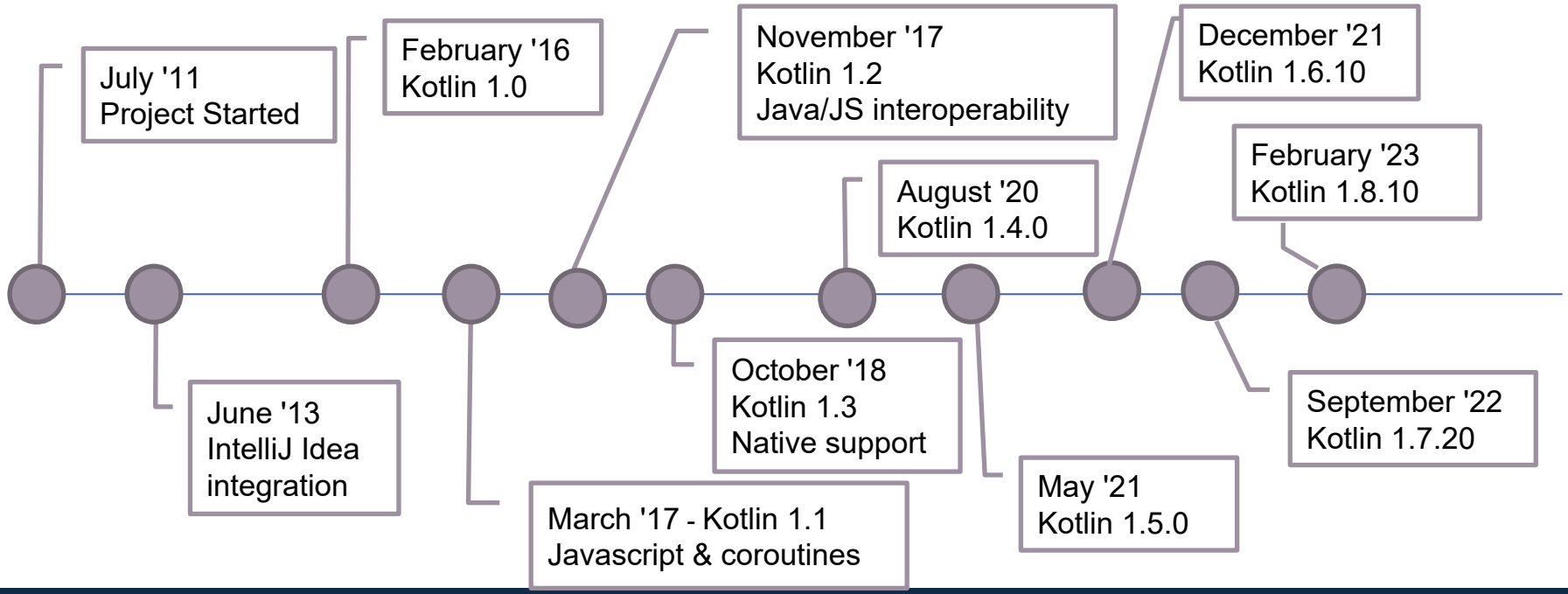
Kotlin

2022-23

Introduction and history

- Statically typed programming language for modern multi-platform applications
  - https://kotlinlang.org/
- Largely inspired by functional programming techniques
  - Higher order functions, closures, immutability, algebraic types, functors, monads, …
- Took sensible ideas from various sources
  - C#, Javascript, Scala, Groovy, Java, Haskell, …
- It targets several platforms
  - JVM, Android, browser (Javascript), native
- Development tools
  - IntelliJ IDEA, Android Studio, Eclipse, CLI, …

The Kotlin timeline



Playing around with Kotlin

- An online REPL interface is available as a playground
  - https://play.kotlinlang.org/
- It allows to easily experiment with the language constructs and see the differences among the various versions of the language and of the target platform
  - JVM, JVM + JUnit, Javascript, Javascript Intermediate Representation (IR), JS Canvas, …
  - https://proandroiddev.com/introducing-the-kotlin-playground-e3625c05a109
- It is possible to share the code created with the playground with other people and to host it in third-party sites

The Kotlin type system

- Kotlin is a statically typed language
  - Variables, temporaries, function parameters and return types are labelled with the type of the value they can be assigned to
- A powerful type inference system is used by the compiler in order to make deductions and detect possible errors as soon as possible
  - As well as saving the programmer from explicitly declare types whenever it is possible to deduce them from the context
- Variables and functions can be annotated with an explicit type in case of ambiguities
  - No type system violations are possible in any case

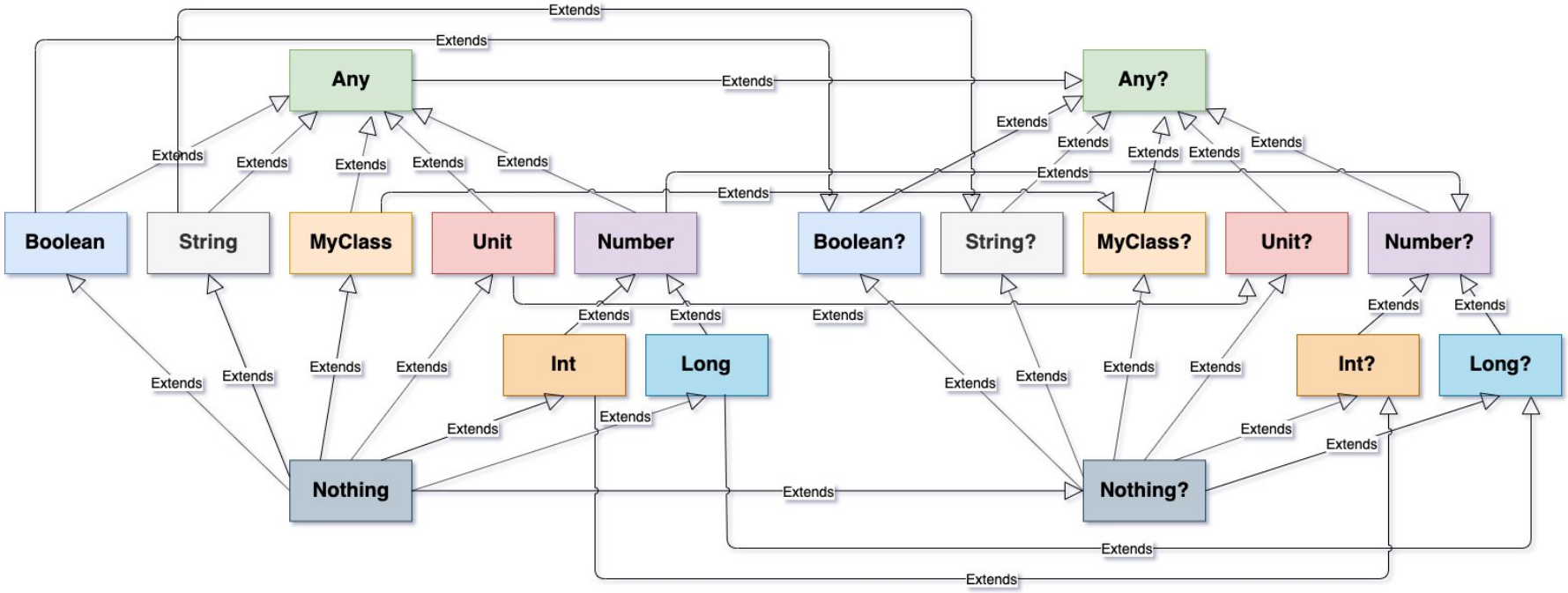
Primitive types

- Differently from the Java language, where a set of primitive types (int, short, long, byte, float, double, char, boolean) complements the classifier types, Kotlin does not have such a distinction
  - Kotlin is only based on the corresponding boxed type (**`Int`**, **`Short`**, **`Long`**, **`Byte`**, **`Float`**, **`Double`**, **`Char`**, **`Boolean`**), thus having **reference semantics** for every type
- This largely simplifies the type hierarchy, avoiding "strange" duplications and inconsistencies in parameter passing and variable assignments
  - However, the compiler internally optimises the actual data type, whenever possible, reducing the actual overhead due to boxing and heap based allocation and guaranteeing the same level of efficiency provided by the Java language

Null safety

- Although every value can be regarded as a reference, Kotlin explicitly distinguishes nullable types by their non-nullable counterpart
  - Nullable types have their name followed by '`?`', while non-nullable ones do not
  - `val s: String = null` results in a compile error, since the type `String` is non-nullable
- All non-nullable variable must be explicitly initialized before being used
  - Since the Java default of assigning them null values is not applicable
- In the type inheritance hierarchy, the class `Any` is the root of every non-nullable class
  - While `Any?` is the root of all the nullable ones
- Class `Nothing`, represents the bottom
  - It inherits from every other class, but there exist no instance of it
  - Class `Nothing?` inherits from every nullable class: its only instance is the `null` keyword

Kotlin type system hierarchy



Using nullable types

- It is not possible to assign null values to non-nullable variables
  - The compiler will prevent it
- It is mandatory to write null-safe code when accessing a nullable variable
  - The language offers special constructs to simplify code writing, while keeping expressivity and readability
- The safe-call operator (`?.`) can be used to access a property or a method of a nullable value
  - It resolves to the usual call operator (`.`) if the value to which is applied is not null, otherwise it immediately returns `null`
- The elvis operator (`?:`) can be used to provide an alternative value (else) if its left-hand side is `null`
  - The pretend-not-null operator (`!!`) throws a runtime exception if its left-hand-side is `null`

Using nullable types

```
var a: String = "abc"

// a = null // compilation error

var b: String? = "abc"

b = null  //ok

var len: Int? = b?.length // len: null

var c: String = b ?: "default" // c: "default"

var len2:Int = b!!.length // Null-Pointer Exception
```

The Unit type

- In Kotlin everything has a type
  - Expressions that do not return anything have the special type named **`Unit`**
  - It represents the value that is returned by an expression that does not explicitly returns anything else (this would be called **`void`** in Java or in C/C++)
- **`Unit`** is at the same type both a type and a value
  - Unlike void, **`Unit`** is a real class (Singleton) with only one instance, the **`Unit`** object
- Functions that return no value (procedures) have **`Unit`** as a return type in their declaration
  - However, it may be simply omitted, since the compiler will assume it as a default

https://dzone.com/articles/kotlin-pearls-7-unit-nothing-any-and-null

Variables and values

- A variable is a named storage location used to store data that can be modified during program execution
  - In kotlin, variables have a static type, and only values that conform to that type can be stored inside them
- Kotlin distinguishes between reassignable variables and final ones
  - The former are introduced by the **`var`** keyword and can be bound to different values during their lifetime
  - The latter are introduced by the **`val`** keyword and, once bound to a value, cannot be reassigned to anything else
- Warning: a final value is not, by itself, immutable!
  - If what is stored inside has a (mutable) state, it may evolve, even if the symbol is declared as **`val`**

Variables and values

```
var age: Int = 25
age = 26 // value of age is changed from 25 to 26

val name: String = "John"
// name = "Jane" // Error: val cannot be reassigned

var x = 10 // type of x is inferred as Int
val y = "Hello" // type of y is inferred as String

val seq: CharSequence = "Jane" // this works because a String is also a CharSequence

val l: MutableList<String> = mutableListOf() // l is an empty mutable list
l.add(name)   // now l contains ["John"]
l.add("Jane") // now l contains ["John", "Jane"]
```

Function types

- Kotlin type system also supports function types
  - I.e., types that define values that are invokable
  - The notation `( Int, Int ) -> Double` denotes a function type that receive two integers and returns a double
  - The notation `() -> String` denotes a function that has no arguments and returns a String
- Function types can be instantiated with regular functions, function literals, and lambda expressions
  - Function types are a powerful feature of Kotlin that can be used to define functions as values, making it easy to pass them around as arguments or return them from other functions
  - They are commonly used in functional programming and can be used to create code that is more modular, flexible, and expressive

Named functions

- Named functions are functions that have a name that can be used to call them
  - They are defined using the `fun` keyword followed by the function name and the function body
  - The function body may contain arbitrary statements and may return a value
  - `fun greet(name:String) { println("hello, $name") }`
- A named function is invoked using its name and passing in the required arguments
  - `greet("Alice") // hello, Alice`
- They are the most common type of function in Kotlin, and are used to define reusable blocks of code that can be called from different parts of a program
  - When a named function is used to instantiate a function type, its name is preceded by `::`
  - `val f: (String) -> Unit = ::greet`

Function parameters

- They specify the types and number of arguments that a function accepts
  - They appear inside parentheses `()` after the function name, and separated by commas
- Function parameters can have default values
  - This allows the function to be called with fewer arguments, if the default values are acceptable
- Function parameters can be named when calling functions
  - This is very convenient when a function has a high number of parameters or default ones

```
fun reformat(str: String,                                   // Mandatory
             normalizeCase: Boolean = true,                 // Optional
             wordSeparator: Char = ' '): String { … }       // Optional

reformat("Hello Kotlin!", wordSeparator = '_')
```

Function body

- Kotlin functions can contain a wide variety of statements, which allow you to define the logic and behavior of your program
  - Assignment statements (`=`)
  - Expression statements (*function calls*, *operators*, and *literals*)
  - Control flow statements (`if`, `else`, `when`, `for`, `while`, and `do-while`)
  - Loop control statements (`break` and `continue`)
  - Exception handling statements (`try`, `catch`, and `finally` blocks)
  - Return statement (`return`)
  - Assertion statement (`assert`)
- Most of them behave in similar way to the corresponding statements in Java, but there are some notable exceptions

Control statements

- The **`if`** clause is an expression: each branch returns a value
  - `val max1 = if (a > b) a else b`
- If a branch contains a block, the last value of the block is returned
  - `val max1 = if (a > b) { println(a); a } else { println(b); b }`
- The **`when`** clause replaces the ***`switch`*** construct of Java
  - The compiler forces to control all possible alternatives
  - It can also be used with ranges, collections, and other types of expressions
  - It is also possible to write it without an argument to match against arbitrary conditions

Control statements

```
fun describeNumber(num: Int): String {
    return when (num) {
        0 -> "Zero"
        1 -> "One"
        2 -> "Two"
        else -> "Other" //mandatory!
    }
}

println(describeNumber(1)) // → "One"
println(describeNumber(7)) // → "Other"
```

```
fun describe(obj: Any): String =
  when {
    obj is String && obj.length < 10 ->
                  "Short string"
    obj is String && obj.length >= 10->
                  "Long string"
    obj is List<*> ->
                  "List (${obj.size})"
    else -> "Other"
  }

println(describeLength("Hello"))
// → "Short string"
println(describeLength(listOf(1,2)))
// → "List (2)"
```

Loops and iterators

- The for statement only operates with *iterators*, *ranges*, or *iterables*
  - The traditional, C-like, syntax has been removed

```
val iter = intArrayOf(1,2,4,8,16,32,64).iterator();

for (item in iter)
  println(item)
```

```
val data = intArrayOf(1,2,4,8,16,32,64);

for ((index,value) in data.withIndex())
  println("data[$index] = $value")
```

```
for (i in 1..10)
  println(i)
```

Function literals

- A function literal consists of an anonymous fun block
  - To be used, it must be assigned to a variable
  - `val sum = fun(a:Int, b:Int) { return a + b }`
- A function literal is invoked via the variable it is bound to
  - `val result = sum( 5, 3 ) // 8`
- Function literals allow to define anonymous functions on the fly, making the code more concise and expressive
  - They are commonly used to pass behavior as an argument to higher-order functions, or to create small, self-contained functions that perform a specific task

Lambda expressions

- Lambda expressions are a more concise way to define anonymous functions
  - They are enclosed in braces and use the `->` operator to separate the parameter list from the body
  - `val l1 : (Int) => Int = { n -> n+1 }`
- If lambda has only one parameter, it can be omitted together with the `->` operator, and can be referred to using the name `it`
  - `val l2 : (Int) => Int = { it + 1 }`
- Similarly to function literals, lambda expressions automatically create **closures** when they reference variables that are in scope where they are defined
  - Powerful feature used in connection to of higher order functions

Trailing lambdas

- If the type of the last parameter of a function is a function itself, and if the invoker is going to supply a lambda function as argument, then it is legal to move the lambda expression outside of the function call parentheses
  - This approach is used extensively in many Kotlin libraries and APIs, and it is an essential part of the language's functional programming capabilities

```
fun combine(a: Int, b: Int, operation: (Int, Int) -> Int): Int {
    return operation(a, b)
}

val result = combine(5, 10) { x, y -> x + y }  // → 15
```

Higher order functions

- Functions can receive other functions as parameters or return them as values
  - Typical pattern used by functional-programming paradigm
- HOFs make code more concise and expressive
  - Since they allow to pass behavior (in the form of functions) as arguments to other functions, or to return a behavior (function) that conditionally depends on received arguments
- HOFs are extremely powerful, and can be used to implement various design patterns
  - such as the strategy pattern, the observer pattern, and the decorator pattern, among others

Closures

- Closures are functions that capture the state of its surrounding environment, including any variables or objects that are in scope
  - In Kotlin, closures are created automatically when lambda expressions or function literals reference variables from the enclosing scopes
  - The life-cycle of the captured variable is thus extended: as long as the closure exists, the captured variable remains accessible, even if the function where it was defined returned

```
fun createGenerator(): () -> Int {
  var c = 0
  return { ++c }
}
val g1 = createGenerator()
val g2 = createGenerator()
```

```
println( g1() ) // 1
println( g1() ) // 2

println( g2() ) // 1
println( g2() ) // 2
```

User defined types

- **Classes**
  - Kotlin fully supports the object-oriented paradigm and provides a set of built-in classes with a single inheritance hierarchy
  - More classes can be defined as extensions of existing ones
  - If no parent class is specified, the compiler automatically assumes the built-in class **Any** as a parent class
  - A class encapsulates constructors, methods, and properties
- Several variations of the class concept are provided
  - **Abstract** classes, **data** classes, **sealed** classes, singleton **objects**, …
- **Interfaces**
  - An interface introduces a new abstract type, declaring a set of methods and properties to be implemented
  - Interfaces may be organised in a multiple (zero or more) inheritance hierarchies among themselves

Classes

- The `class` keyword introduces a new class
  - Classes have a **primary constructor** and zero or more **secondary** constructors

```
class Person constructor(val name: String) {

    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }

           constructor(name: String, parent: Person, age: Int): this(name) {
        parent.children.add(this)
        this.age=age;
    }

    var children: MutableList<Person> = mutableListOf<Person>();
    var age = 0;
}
```

Constructors

- The primary constructor is part of the class **header**
  - Its parameters prefixed by **`val`** or **`var`** are automatically promoted to properties
  - No initialization can happen inside it: if necessary, an **`init`** block must be provided
  - Init blocks are run in declaration order after the primary constructor is finished
  - If it has no annotations or visibility modifiers, the **constructor** keyword may be dropped
- Secondary constructors are introduced by the **constructor** keyword
  - They must always delegate to the primary one before any extra action can take place inside the curly braces
  - If a new instance has been created using a secondary constructor, the code block associated with the invoked constructor is executed last

Constructors

```
class Segment(val start: Point, val end: Point) {
  val length: Double = sqrt(
      (end.x - start.x).toDouble().pow(2.0) + (end.y - start.y).toDouble().pow(2.0))
  // initializer block
  init {
    println("Point starting at $start with length $length")
  }
  // secondary constructor
  constructor(x1: Int, y1: Int, x2: Int, y2: Int):  this(Point(x1, y2), Point(x2, y2) {
      println("Secondary constructor")
  }
}
```

```
val s = Segment(1,2,3,4)
// >>>>>>>>
Point starting at Point(x=1, y=2) with length 2.8284271247461903 //the properties length and start are
initialized with their constructor-supplied value since the primary constructor is run before the init block
Secondary constructor //the init block is run before the code block associated with the secondary
constructor
```

Properties

- Each class can declare zero or more properties that are used to store and represent data within instances of that class
  - They can be read/write (introduced by the `var` keyword) or read/only (introduced by the `val` keyword)
  - The compiler automatically generates hidden fields, to store the value of each property, as well as a set of getters and setters function, to manipulate their content: `val` properties only have a getter
- Getters and setters can be explicitly overriden, if necessary
  - By adding an extra `get()` or `set(...)` block after their declaration
  - This allow to create computed properties as well as to delegate their behaviour
- By default, all properties are public
  - But their visibility can be restricted to private, protected, or internal

Inner fields

- No fields can be declared by the programmer to back properties up
  - They are automatically handled by the compiler
  - If needed, an accessor function may refer to the backing field, by using the **`field`** keyword
  - The **`field`** identifier has a special meaning only within the custom getter and setter, where it refers to the backing field that contains the property's state

```
class Document(_summary: String) {
    var summary: String = _summary
        set(value) {  field = value; hashCode = computeHash() }
    var hashCode: Int = computeHash()
        private set
    private fun computeHash(): Int { println("Hashing..."); return summary.hashCode() }
}

val d = Document("abc") // → Hashing…
d.summary = "qwerty" // → Hashing…
```

Properties

```
class Circle(var radius: Double) {
    val area: Double
        get() = Math.PI * radius * radius
}


val c = Circle(2.0)  // Instantiates a circle with radius 2.0
c.radius *= 0.5      // modifies the radius property, setting it to 1.0

println(c.area)      // access the (computed) property area and prints 3.141592…
```

Visibility modifiers

- `public`
  - The default visibility modifier in Kotlin.
  - A public class, member or top-level function can be accessed from anywhere in the program
- `private`
  - A private class, member or top-level function can only be accessed from within the same file or class that defines it
- `protected`
  - A protected member can be accessed from the same class or its subclasses, but not from outside the class hierarchy
- `internal`
  - An internal class, member or top-level function can be accessed from any class or file within the same module
  - A module is a set of Kotlin files that are compiled together

Lateinit var properties

- The value of a variable may not be available at the object construction time
  - Declaring a variable as lateinit allow the compiler not to complain that its missing an initialization
  - The variable must be mutable because it will be assigned a value, later
  - If the variable is accessed before it's initialized, an UninitializedPropertyAccessException is thrown

```
class MyActivity: Activity(){
      private lateinit var button: Button //this will be initialized later

      override fun onCreate(bundle: Bundle?) {
          super.onCreate(bundle)
          button = Button(this)   //initialization occurs here
      }
}
```

Delegates

- Usually, properties are automatically stored inside class instances in hidden fields
  - If some common behaviour need to be applied, code tends to repeat
- Kotlin address the case via **delegated properties**
  - The keyword **by** is used in the property declaration to define a delegate
- The Kotlin class library offers some ready made delegates that are function responsible for getting and setting the value of the property
  - **`Lazy`** properties store a value that is computed only on first access
  - **`Observable`** properties invoke a listener whenever the value changes
  - **`Vetoable`** properties let a listener decide whether a value should be stored or not in the property

Lazy properties

- `lazy()` is a function that takes a lambda and returns an instance of `Lazy<T>` where `T` is the type returned by the lambda
  - The first call to `get()` executes the lambda and remembers the result
  - Subsequent calls return the remembered value

```
val lazyValue: String by lazy {
    print("computed!")
    "Hello"
}

fun main() {
    println(lazyValue) // → computed!Hello
    println(lazyValue) // → Hello
}
```

Observable properties

- `Delegates.observable()` implements the observable pattern
  - It takes two arguments: the initial value and a lambda, which will be invoked after any assignment is performed
  - The lambda has three parameters: the property being assigned to, the old value and the new one

```
class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new -> println("$old -> $new")
    }
}

fun main() {
    val user = User()
    user.name = "first"   // <no name> -> first
    user.name = "second"  // first -> second
}
```

Custom delegates

```
class Example {
  private val p$delegate = MyCustomDelegate()
  var p: String
    get() = p$delegate.getValue()
    set(value:String) =p$delegate.setValue(value)
}
```

```
class Example {
  var p: String by MyCustomDelegate()
}

class MyCustomDelegate {
  operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
    // compute the value…
  }

  operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {
    // store the value…
  }

}
```

Methods

- Methods are functions defined within the scope of a class
  - They are used to define behavior for the instances of that class
  - Each method receive an implicit parameter, named `this`, that represent the instance the method is operating on
- A method can be invoked on an object using dot notation
  - `someObject.someMethod(param1, param2)`
- If a method implementation can be reduced to the evaluation of a single expression, its body can be replaced by the operator '`=`' followed by the expression
  - `fun sum(a:Int, b:Int) = a + b`

Operator overloading

- Operator overloading is a feature that allows to redefine the behavior of operators such as **+**, **-**, *****, and **/** for instances of a class
  - If class **C** overloads operator '**+**', and **c1** and **c2** are instances of **C**, it is legal to use expression **c1 + c2**
- To overload an existing operator in Kotlin, a method with a suitable name must be defined inside the class
  - Such a method should be preceded by the **operator fun** keywords

```
class Vector2D(val x: Int, val y: Int) {
    operator fun plus(other: Vector2D): Vector2D {
        return Vector2D(x + other.x, y + other.y)
    }
}
val res = Vector(1,2) + Vector(2,3) // → Vector(3,5)
```

Equality testing

- In Kotlin, the equality operator `==` is used to compare the values of two objects
  - While the identity operator `===` is used to check whether two references point to the same object
  - This maps to the `equals(other: Any?): Boolean` method, which is available for each object since it is defined in the `Any` class and inherited by all classes
- If two objects need to be equality-compared, a suitable overloaded implementation of `equals(...)` must be provided
  - It must be coherent with the general equality contract, guaranteeing the respect of the reflexive, symmetric, and transitive properties

hashCode()

- The Any class also defines a `hashCode()` method, which is intended to provide a hash value that subsumes the object state
  - `hashCode()` and `equals(...)` must be defined coherently with each other
  - If two objects are equal, they should have the same hash code
  - The reverse is not true: two objects having the same hash code can be different
- Whenever the `equals(...)` method is overridden, the `hashCode()` method must be overridden as well
  - Failure to do so will prevent the object from being compatible with some standard library containers, like `Set`s and `Map`s

Order testing

- Primitive types can be compared using order operators (`<`, `<=`, `>=`, `>`) according to natural ordering
  - Numerical for numbers
  - Alphabetical for String
- Other types can be compared using the same operators if they implements the `Comparable<T>` interface
  - `operator fun compareTo(other: T): Int`
- The method must return:
  - Zero if `this` is equal to `other`
  - Any negative number if it's less than `other`
  - Any (strictly) positive number if it's greater than `other`

Singleton objects

- Whenever a class is going to have one single instance, and the primary constructor has no arguments, it is possible to declare a singleton object
  - It is obtained replacing the **`class`** keyword with the **`object`** one, and keeping the remaining syntax
  - An object is similar to a regular class in that it can have properties, methods, and implement interfaces
  - It can be referenced to it by its name

```
object MySingleton {
  private var _counter = 0
  fun increment() { ++ _counter }
  val counter = _counter
}
```

```
MySingleton.increment()
println(MySingleton.counter) // → 1

MySingleton.increment()
println(MySingleton.counter) // → 2
```

Companion objects

- Properties and methods defined inside a class operate at the instance level
  - Each instance has its own values and they are referred to via the dot-notation applied to instances of the class
- Sometimes it is necessary to define properties and behaviors that are shared amongst all the instances of the class
  - Java declares them as '**`static`**'
  - In Kotlin, they are declared using the **`companion object`** keyword, and are accessed using the class name

```
class Person(val name: String, var age: Int) {
    companion object {
        val defaultName = "Unknown"
    }
}
println(Person.defaultName) // → Unknown
```

Extension functions

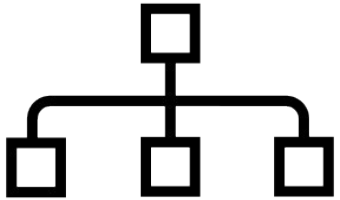
- An extension function is defined as a regular function, but with the class or interface that it extends as its **receiver type**
  - The function can then be called on any instance of that class or interface, as if it were a member function of that class or interface
- Extension functions can be used to add new functionality to an existing class without having to modify the class's source code or subclass it
  - The compiler transforms the extension function into a regular named function having one more parameter named “this” which is bound to the receiver object
  - As such, the extension function has no access to the private member of the receiver and represent just a form of “syntactic sugar”, which - however - keep the code concise and expressive

Extension functions

```
// add a new method to the List class named “second()”
fun <T> List<T>.second(): T? {
    return if (this.size >= 2) {
        this[1]
    } else {
        null
    }
}

val list = listOf("one", "two", "three")
val second = list.second()  // → two
```

Inheritance



- Kotlin classes are part of a single-parent inheritance hierarchy
  - The root of which is **`Any`**
  - **`Any`** roughly corresponds to Java **`Object`**, but it lacks some of its methods
- To create a subclass, the **`:`** symbol is used as a preifx of the superclass name
  - If a class does not indicate a superclass, it defaults to **`Any`**
- By default, classes cannot be extended
  - To allow extensions, a class should be declared with the **`open`** annotation
  - Moreover, if any method is to be overridden, they, too, must be preceded by the **`open`** keyword
  - In subclasses, overriding methods must be preceded by the keyword **`override`**

Inheritance

```
open class Base(val p: Int) {
  open fun method1(): Int = p

  fun method2(): String = "Cannot be overridden"
}

class Derived(p: Int) : Base(p) {
  override fun method1(): Int = p*2

  // override fun method2(): String = "Error" // method2 is final
                                              // and cannot be overridden
}
```

Inheritance

- A class has a single superclass and may implement zero or more interfaces
  - Both the superclass and the interfaces are specified after the primary constructor, in a comma separated list

```
open class Rectangle(var width: Double, var height: Double) {
    open fun draw() { /* ... */ }
}
interface Polygon {
    fun area(): Double // interface members are 'open' by default
}
class Square(side: Double) : Rectangle(side, side), Polygon {
    // Both draw() and area must be overridden:
    override fun draw() {
        super<Rectangle>.draw() // call to Rectangle.draw()
    }
    override fun area() = width*height
}
```

Data classes

- A data class is a class that is designed to store data, and only has state but no behavior
  - The primary constructor should have at least one parameter
  - All primary constructor parameters should be marked either as `val` or as `var`
  - Data classes cannot be `abstract`, `open`, `sealed` or `inner`
- Data classes are useful because they provide a lot of useful functionality "for free", since the compiler automatically implements several methods:
  - `equals(...)` and `hashCode()`
  - `toString()`
  - `component1()`, …, `componentN()`
  - `copy(...)`

Plain classes vs data classes

```
class Point(       var x: Int, var y: Int)
val p1 = Point(1,1)
val p2 = Point(1,1)

println( "Same point?  ${p1 == p2}" )

>>>> "Same point? false"

This definition of the Point class does not override the equals method, but refers to
that of Point's base class, Any, which returns true only when an object is compared
with itself
```

Plain classes vs data classes

```
data class Point( var x: Int, var y: Int)

val p1 = Point(1)
val p2 = Point(1)
println("Same point? ${p1 == p2}")

>>>> "Same point? true"

The new definition of Point as a data class automatically includes overrides for
methods, as equals, whose implementation depend on the properties of the class.For
example, the data class version of Point contains an equals method that is equivalent
to this:
override fun equals(o: Any?): Boolean {
  // If it's not a Point, return false. Note that null is not a Point
  if (o !is Point) return false
  return x == o.x && y == o.y
}
```

Enum classes

- Enums are a way of creating a class that has a specified, static set of instances
  - The enums classes cannot subclassed, but can implement interfaces and can have constructors, properties, and methods

```
enum class GymActivity {
     BARRE, PILATES, YOGA, FLOOR, SPIN, WEIGHTS
}

enum class HttpStatusCode(val value: Int) {
     OK(200), CREATED(201), NOT_FOUND(404),
     ACCESS_DENIED(403), INTERNAL_SERVER_ERROR(500)
}
```

Enum classes

- Storing properties in enum classes provides a convenient and type-safe way to define a fixed set of constants associated to each possible value
  - By using an enum class, you can ensure that a variable or function parameter can only take on one of a fixed set of values, and any other value is considered invalid, thus making the code more robust
  - By having one or more properties associated with each value, you can guarantee that when that specific value is manipulated, you will access the right constants related to it
- Enum classes are conveniently tested using the `when` expression
  - Since an enum has a known set of instances, Kotlin is able to determine that all possible values are covered as explicitly listed instances and allows the omission of the `else` clause
  - If, in the future, a new value is added to the enum set, a compile error will be flagged, to notify that a branch is missing

Enum classes

```
fun getSuccessfulCode(code: HttpStatusCode): Int? = when (code) {
        HttpStatusCode.OK -> code.value
        HttpStatusCode.CREATED -> code.value
    HttpStatusCode.NOT_FOUND -> null
    HttpStatusCode.ACCESS_DENIED -> null
    HttpStatusCode.INTERNAL_SERVER_ERROR -> null
}
```

Sealed classes

- Sealed classes do for types what enums do for instances: they indicate the compiler that a given base type has a certain, well-defined set of subtypes
  - In the example, the Kotlin compiler knows that the only possible subclasses are Success and Failure
  - Because of this, the else clause can be removed

```
sealed class Result {
    data class Success(val data: List) : Result()
    data class Failure(val error: Throwable?) : Result()
}
fun processResult(result: Result): List = when (result) {
    is Success -> result.data
    is Failure -> listOf()
}
```

Generics

- Generics is a programming paradigm that allows to define a class or a function that can work with different types of data
  - Without having to rewrite the code for each specific type
  - Functions, classes, interfaces, …, can be defined as depending on one or more type parameters, which are enclosed in angular brackets (`<` … `>`) and become part of the type name
- Differently from what happens in other languages, in Kotlin (and in Java) the implementation of generic types is based on type erasure
  - This means that the type parameter information is not available at runtime because it is replaced by `Any?` (or `Object`, in Java) thus making the type capable to work with all other types, thanks to inheritance
  - At compile time, however, the type information is retained and used to check the validity of the program

Generic functions

- A function that receives as parameter or returns a value of generic type **T** must be prefixed by the **<T>** annotation
  - T may be constrained to extend a given class or implement an interface by adding an upper bound after it, like **<T: Runnable>**
  - More complex constraints may be expressed using the **where** clause

```
fun <T> sort(l: List<T>) where T : CharSequence, T : Comparable<T> { … }

sort( listOf( "three", "two", "one" ) ) //ok
//String implements both Comparable<String> and CharSequence

// sort( listOf( 1, 2, 3 ) )
// Type mismatch: inferred type is IntegerLiteralType[Int,Long,Byte,Short]
// but CharSequence was expected
```

Generic classes

- A class may depend on generic types, too
  - The Kotlin syntax requires that type parameters have to be declared after the class name, before the primary constructor
- A generic parameter may be prefixed with the **`in`** and **`out`** keyword to specify variance in generic types
  - Variance refers to the way in which the subtyping relationship between types is preserved or reversed when the types are used as type arguments

```
class Box<T>(private val content: T) {
    fun getContent(): T {
        return content
    }
}
```

```
val b1 = Box("hello")
val b2 = Box(12)
val s: String = b1.getContent() // "hello"
val i: Int = b2.getContent() // 12
// val b = b1 is Box<Int>
```

Cannot check for instance of erased type: Box<Int>

Variance in generic classes

```
interface Source<out T> {
    fun next(): T
}
```

- The **out** keyword specifies that T is a **covariant** type parameter in **Source**
  - It means that **Source<S>** is a subtype of **Source<T>** if **S** is a subtype of **T**
- Covariant type parameters can only appear in output positions
  - Such as the return type of a function or a property

```
interface Sink<in T> {
    fun put(item: T)
}
```

- The **in** keyword specifies that T is a **contravariant** type parameter in **Sink**
  - It means that **Sink<T>** is a subtype of **Sink<S>** if **T** is a subtype of **S**
- Contravariant type parameters can only appear in input positions
  - Such as the parameter types of a function or a property setter

Variance in generic classes

- Class Pipeline is **invariant** with respect to parameter **T**
  - This means that **Pipeline<S>** and **Pipeline<T>** are two unrelated classes, independently of any possible inheritance relationship between **S** and **T**
- Invariant type parameters can appear in both input and output positions

```
class Pipeline<T>(private val source: Source<T>, private val sink: Sink<T>) {
    fun start() {
        while (true) {
            val item = source.next()
            sink.put(item)
        }
    }
}
```

Scoped functions

- Scoped functions are functions that execute a block of code within the context of an object
  - They provide a way to give temporary scope to the object under consideration where specific operations can be applied, thereby, resulting in a clean and concise code
- The standard library provides five ready-made scoped functions: they differ in purpose and in syntax
  - `let…`, `run…`, and `with(…)` executes a block of code on an object and returns the result, while `apply…` and `also…` modifies the object and returns it
  - `let…`, `run…`, `apply…`, and `also…` are extension functions of a generic type `T`, while `with(…)` is a regular function operating on a generic parameter `T`
  - `run…`, `with(…)`, and `apply…` accept a block based on a lambda with receiver, while `let…` and `also…` have their block defined as a simple lambda

Scoped functions

```
var name: String? = null
// …name is updated…

// let is used to prevent a
// NullPointerException from occurring
name?.let {
  println(it.reversed)
  println(it.length)
}
```

```
class Car{ var name:String = "",
           var maker:String = "" }

// apply is used to configure an object
val car = Car().apply {
  name = "Panda"
  maker = "Fiat"
}
```

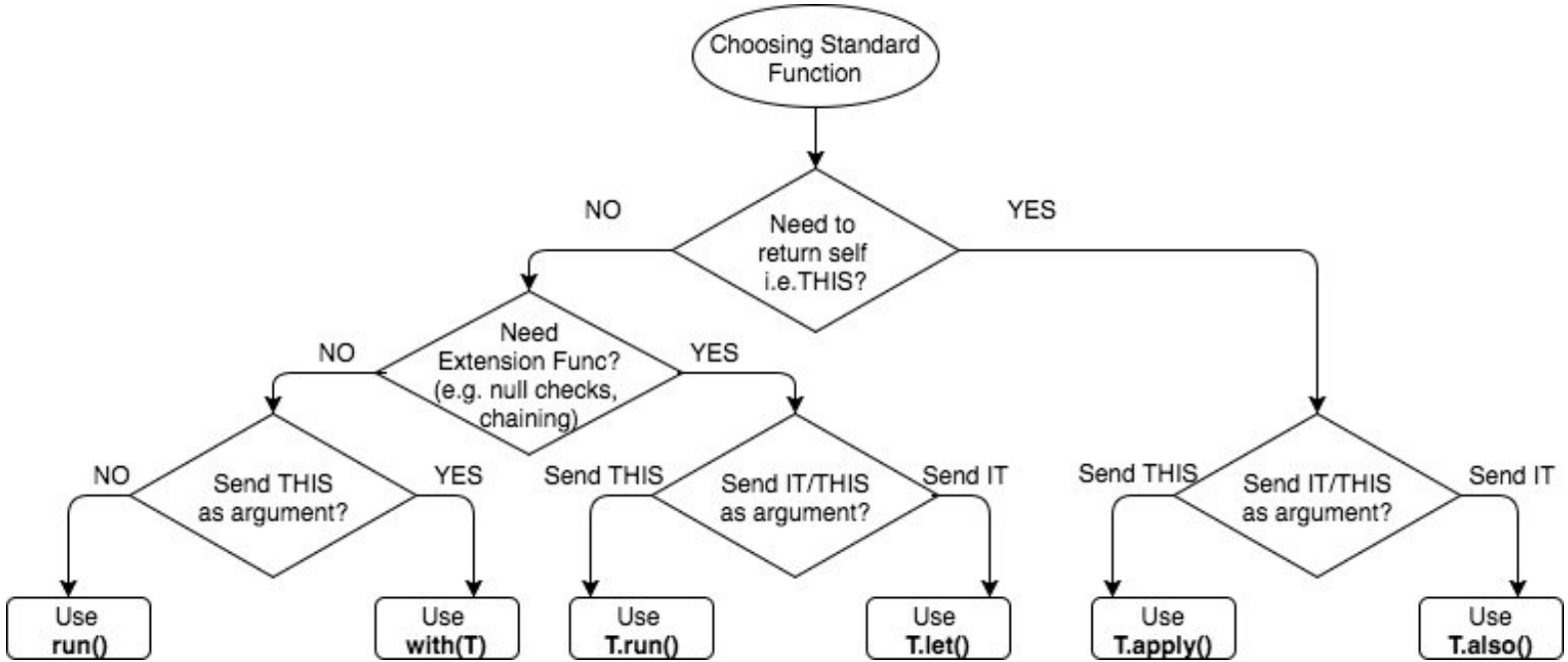
```
var occurrences: MutableMap<String,Int> = …
var name: String? = null
// …name is updated…

// run is used to operate on a nullable object
val count = name?.run {
  val c = (occurrences[this]?:0) + 1
  occurrences[this] = c; c
}
```

```
val car = Car()
// configure car…

// with operates on a non-null object
with (car) {
    println(name)
    println(maker)
}
```

Choosing a scoped function



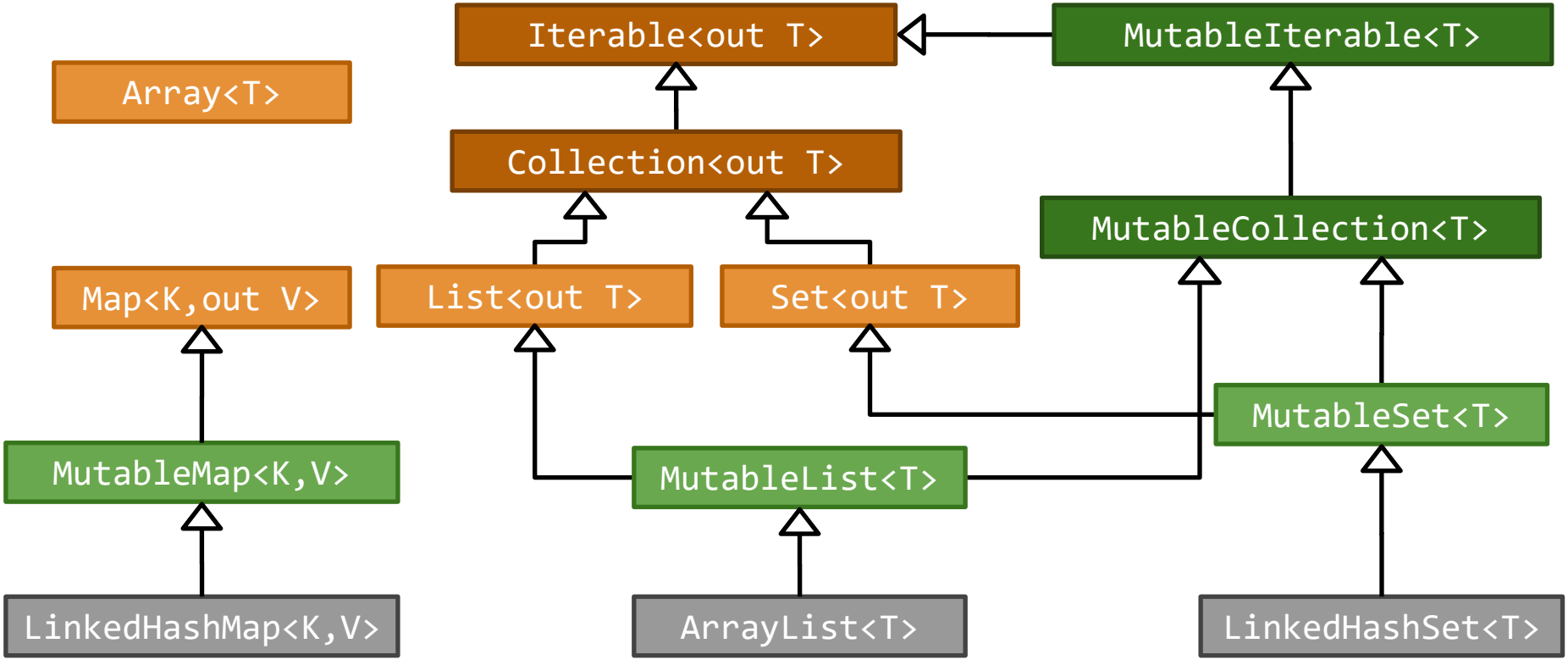
Type checking and casting

- In some situations, it is necessary to assess the actual type of the value stored in a variable
  - Operator `is` (and its negated form `!is`) can be used to this purpose
  - `let v: Any = …; if (v is String) { … }`
- A value can be casted to a different type using the unsafe `as` operator
  - It will throw an exception if the conversion is not possible
- Alternatively, the `as?` safe cast operator can be used
  - It will return `null` if the conversion is not possible
- The compiler tracks is-checks
  - It inserts safe casts on your behalf
  - This includes also negative checks & short circuiting

Smart casts

- After an is-check, the compiler tries to automatically cast the checked variable to the verified class
  - This is called smart casting
- Smart casting only applies if the compiler can guarantee that the field cannot be mutated since the is-check
  - **`val`** local variables – smart cast is always performed
  - **`val`** properties - if the property is private or internal or the check is performed in the same module where the property is declared, the smart cast is performed
  - **`var`** local variables – if the variable is not modified between the check and its usage and it is not captured in a lambda that may modify it, the smart cast is performed
  - **`var`** properties - the smart cast is never performed

The Kotlin collection library



Working with collections

- The Kotlin Standard Library provides several **helper functions** to create and initialize collections
  - Immutable factories:
    - `listOf(values...)`, `setOf(values…)`, `mapOf(values…)`, `arrayOf(values…)`, `listOfNotNull(collection)`, `emptyList()`, `emptySet()`, `emptyMap()`
  - Mutable factories:
    - `mutableListOf(values...)`, `mutableSetOf(values…)`, `mutableMapOf(values…)`
- Moreover, a large set of **manipulation methods** is available
  - Providing support for filtering, grouping, mapping, and testing values inside the collection
- These functions are designed to be easy to use and can help you write more concise and expressive code when working with collections

Filtering methods

- `filter(predicate: (T) -> Boolean): List<T>`
  - returns a list containing only elements matching the predicate
  - `filterNot(...)` does the reverse
  - `filterNotNull(...)` removes all of the nulls from a collection
- `find(predicate: (T) -> Boolean): T?`
  - returns the first element matching the given predicate, or `null` if none is found

```
val nums = listOf(10, 20, 100, 5)
val numbers = nums.filter { it > 20 } // → [100]

val divisibleByThree = nums.find { it % 3 == 0} // → null
val divisibleByFour = nums.find { it % 4 == 0} // → 20
```

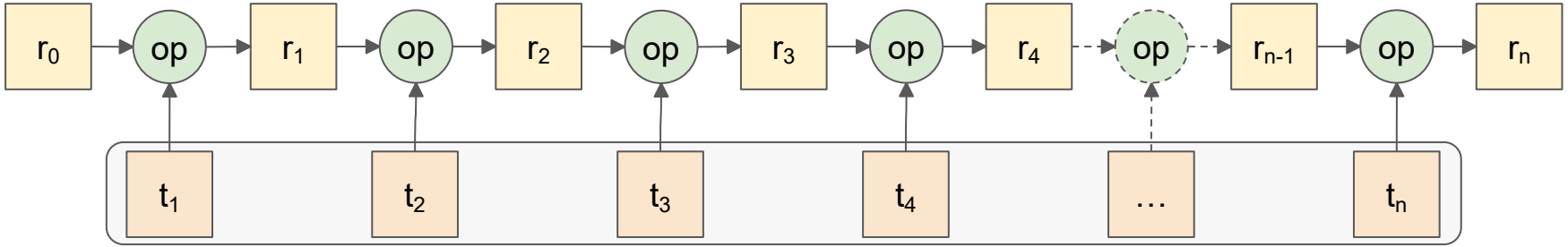
Grouping methods

- `groupBy(keySelector: (T) -> K): Map<K, List<T>>`
  - groups elements having the same key into a list and returns a map where each key is associated to its list

```
val numbers = listOf(1, 20, 18, 37, 2)
val groupedNumbers = numbers.groupBy {
     when {
          it < 20 -> "less than 20"
          else -> "greater than or equal to 20"
     }
}
// The variable groupedNumbers now contains a Map>. The map has two keys,
“less than 20” whose value is [1, 18, 2], and “greater than or equal to 20”
whose value is [20, 37]
```

Grouping methods

- `partition(predicate: (T) -> Boolean): Pair<List<T>, List<T>>`
  - Splits the collection into a pair of lists: the first element contains all elements that satisfy the predicate, the second one contains the others
- `fold(initial: R, operation: (R,T) -> R): R`
  - Accumulates a value starting with `initial` and applying `operation` from left to right



Mapping methods

- `map(transform: (T) -> R): List<R>`
  - Transforms all the elements of the collection by applying them the given function
  - It does not mutate the collection to which it is applied, but it returns a new list
- `mapIndexed(transform: (index: Int, T) -> R): List<R>`
  - The transforming function is supplied with both the element and the corresponding index
- `mapNotNull(transform: (T) -> R?): List<R>`
  - returns a list of all non-null outcomes obtained applying the supplied transform function to each element of the original collection

```
val doubles: List = listOf(1.0, 2.0, 3.0, null, 5.0)
val squares: List = doubles.mapNotNull { it?.pow(2) }
//squares = [1.0, 4.0, 9.0, 25.0]
```

Mapping methods

- `flatMap(transform: (T) -> Iterable<R>): List<R>`
  - Applies a transformation function to each element of the original collection, and then flattens the resulting collections into a collection of plain values

```
val list: List> = listOf(listOf(1, 2, 3, 4), listOf(5, 6))
val flatList: List = list.flatMap { it.filter {v -> v%2 == 0} }
//       The variable flatList will have the value [ 2, 4, 6].
```

Mapping methods

```
// example that returns a list of valid starting dates for employee records
// that have those starting dates stored as strings:

data class Hire(
     val name: String,
     val position: String,
     val startDate: String )

fun getStartDates(l: List<Hire>): List<Date> {
     val formatter = SimpleDateFormat("yyyy-MM-d", Locale.getDefault())
     return l.mapNotNull {
          try { formatter.parse(it.startDate) }
          catch (e: Exception) { null }
     }
}
```

Boolean methods

- `any(predicate: (T) -> Boolean) : Boolean`
  - Will return true if the predicate evaluates true for any element in the collection
- `all(predicate: (T) -> Boolean): Boolean`
  - returns true only if every element in the list matches the predicate
- `none() : Boolean`
  - Returns true only if there the collection is empty
- `none(predicate: (T) -> Boolean): Boolean`
  - returns true only if the predicate evaluates to true for none of the elements in the collection

Boolean methods

```
val nums = listOf(10, 20, 100, 5)
val isAny = nums.any() // true

val isAnyOdd = nums.any { it % 2 > 0 } // true

val isAnyBig = nums.any { it > 1000} // false

val areAllEven = nums.all { it % 2 == 0 } // false

val isNone = nums.none() // false

val isNone4 = nums.none { it == 4 } // true
```

Sequences

- A **`Sequence<T>`** is an iterable container type that operate lazily, potentially allowing infinite elements
  - Similar to Java 8+ **`java.util.stream.Stream<T>`**
- When a sequence undergoes a multi-step processing, elements are extracted and processed one-by-one
  - While collections such as lists, sets, and maps are eagerly evaluated, sequences are evaluated only when necessary
  - No intermediate container is usually created, improving the performance of the chain

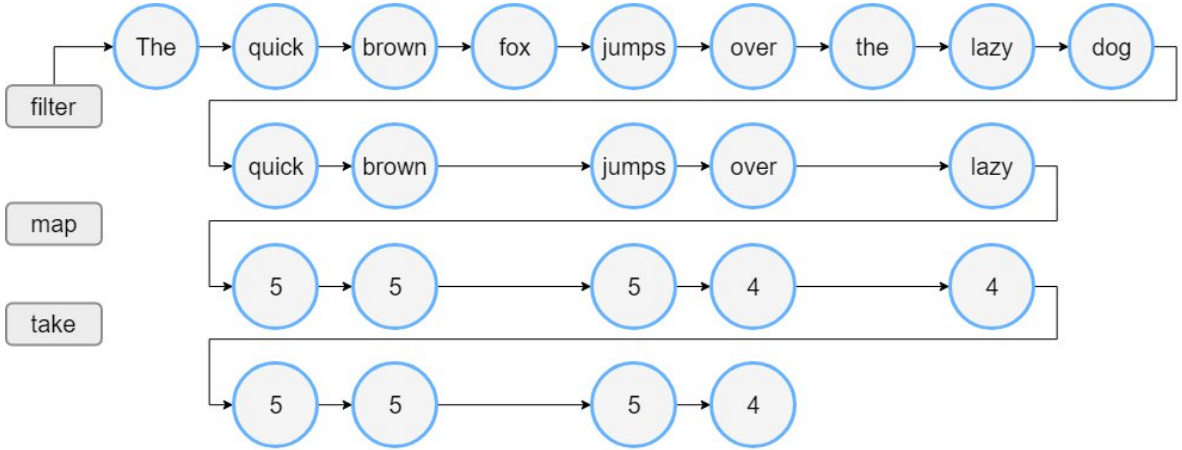
Creating sequences

- From elements
  - `val numbers = sequenceOf( "three",  "two", "one" )`
- From any Iterable
  - `val numbers = listOf("three", "two", "one").asSequence()`
- From a function
  - ```
    val numbers = generateSequence { try { readLine().toInt() }
                                     catch (e:Exception) { null } }
    ```
- From chunks
  - ```
    val primes = sequence {
          yield(1)
          yieldAll( listOf(2,3,5) )
    }
    ```

Operating on collections vs sequences

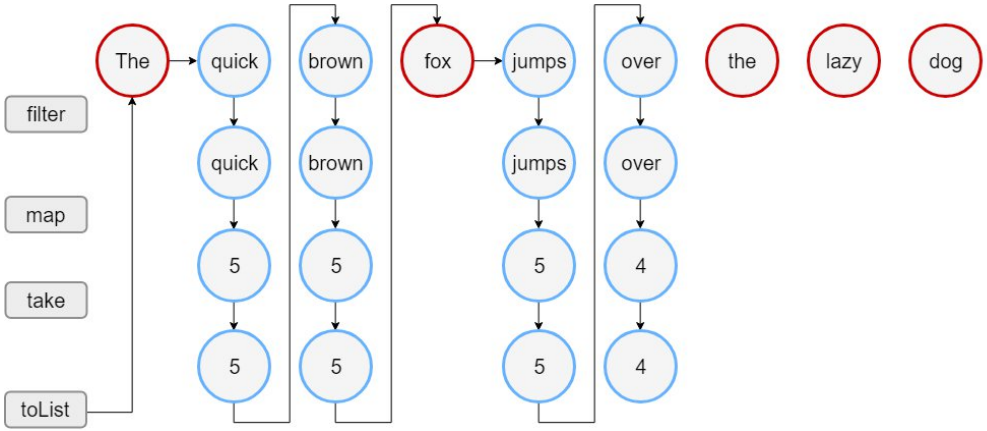
```
val words = "The quick brown fox jumps over the lazy dog".split(" ")

val lengthsList = words.filter { it.length > 3 }
    .map { it.length }
    .take(4)
```



Operating on collections vs sequences

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
val wordsSequence = words.asSequence()
val lengthsList = wordsSequence.filter {it.length > 3 }
    .map { it.length }
    .take(4)
    .toList()
```



Learning more

- Diving Deeper Into Kotlin Standard Library Functions
  - https://blog.kiprosh.com/kotlin-standard-library-functions/
- Kotlin Nitpicks: Language and Standard Library
  - https://niklaslochschmidt.com/blog/2022/kotlin-nitpicks-language-and-stdlib/
- Kotlin's way to make DSLs and many standard library functions work
  - https://proandroiddev.com/kotlins-way-to-make-dsls-and-many-standard-library-functions-work-a8e750c38628