Kotlin Program with 3 Constructors

```
kotlinCopyEditclass Student {
    var name: String
    var age: Int
    var grade: String

    // Primary constructor
    constructor(name: String, age: Int, grade: String) {
        this.name = name
        this.age = age
        this.grade = grade
        println("Primary Constructor: $name, $age, $grade")
    }

    // Secondary constructor 1 (only name)
    constructor(name: String) : this(name, 0, "Not Assigned") {
        println("Secondary Constructor 1: Only name provided.")
    }

    // Secondary constructor 2 (name and age)
    constructor(name: String, age: Int) : this(name, age, "Not Assigned") {
        println("Secondary Constructor 2: Name and Age provided.")
    }

    fun showDetails() {
        println("Student Details: Name = $name, Age = $age, Grade =
$grade\n")
    }
}

fun main() {
    val student1 = Student("Ankit", 20, "A")
    student1.showDetails()

    val student2 = Student("Priya")
    student2.showDetails()

    val student3 = Student("Rahul", 18)
    student3.showDetails()
}
```

▯ Explanation

- **Primary constructor** initializes all properties.
- **Secondary constructor 1** uses only name.
- **Secondary constructor 2** uses name and age.
- All constructors eventually call the **primary one** using `: this(...)`.

⍰ What is a `companion object` in Kotlin?

In Kotlin, a `companion object` is used to **define static-like members** (like Java's `static`) inside a class.

- You use it when you want to call a function or access a property **without creating an object of the class**.
- It belongs to the class, not to an instance of the class.

⍰ Basic Example

```
kotlinCopyEditclass MyClass {
    companion object {
        val companyName = "OpenAI"

        fun greet() {
            println("Welcome to $companyName!")
        }
    }
}

fun main() {
    // No object creation required
    println(MyClass.companyName)
    MyClass.greet()
}
```

⍰ *Output:*

```
cssCopyEditOpenAI
Welcome to OpenAI!
```

⍰ Why use `companion object`?

1. To define **factory methods** (alternate ways to create objects).
2. To define constants or utility methods.
3. To **access members without creating objects**.

⍰ Example with Factory Method

```
kotlinCopyEditclass User private constructor(val name: String) {

    companion object {
        fun createUser(name: String): User {
            println("Creating user with name: $name")
            return User(name)
        }
    }
}
```

```
fun main() {
    val user = User.createUser("Ankit")
    println(user.name)
}
```

▯ Rename the Companion Object

You can name your companion object for clarity:

```
kotlinCopyEditclass AppConfig {
    companion object Config {
        const val version = "1.0"
        fun info() = println("App Version: $version")
    }
}

fun main() {
    AppConfig.info()         // Valid
    AppConfig.Config.info()  // Also valid
}
```

▯ Important Points

- Kotlin classes **don't have `static`** keyword like Java.
- `companion object` members can be accessed using the **class name**.
- Only **one companion object** is allowed per class.

▯ What is Inheritance?

Inheritance allows a **class (child/subclass)** to **inherit** properties and functions from another class (parent/superclass). This promotes **code reusability**.

▯ 1. Open Classes (Kotlin Specific)

In Kotlin, **classes are final by default**. You must use the open keyword to allow them to be inherited.

```
kotlinCopyEditopen class Animal {
    fun eat() {
        println("Animal is eating")
    }
}
```

If you **don't** write open, no class can inherit from it.

⍰ 2. Creating a Subclass

```
kotlinCopyEditclass Dog : Animal() {
    fun bark() {
        println("Dog is barking")
    }
}

fun main() {
    val dog = Dog()
    dog.eat()   // inherited
    dog.bark()  // own method
}
```

✅Dog inherits `eat()` from `Animal`.

⍰ 3. Overriding Methods

Use open in the parent method and `override` in the child.

```
kotlinCopyEditopen class Animal {
    open fun sound() {
        println("Some generic animal sound")
    }
}

class Cat : Animal() {
    override fun sound() {
        println("Meow")
    }
}
```

⍰ 4. Constructor Inheritance

If the superclass has a constructor, the subclass must call it.

```
kotlinCopyEditopen class Person(val name: String) {
    fun showName() {
        println("Name is $name")
    }
}

class Student(name: String) : Person(name)
```

⍰ 5. Super Keyword

Used to access the parent class's method or constructor.

```
kotlinCopyEditopen class Bird {
    open fun fly() {
        println("Bird is flying")
    }
}

class Eagle : Bird() {
    override fun fly() {
        super.fly()
        println("Eagle is flying high")
    }
}
```

⍰ 6. Inheriting Properties

You can also override properties:

```
kotlinCopyEditopen class Shape {
    open val sides: Int = 0
}

class Triangle : Shape() {
    override val sides: Int = 3
}
```

⍰ 7. Abstract Classes

Used when you want **partially implemented** functionality.

```
kotlinCopyEditabstract class Vehicle {
    abstract fun run()

    fun start() {
        println("Vehicle starting")
    }
}

class Car : Vehicle() {
    override fun run() {
        println("Car is running")
    }
}
```

---

8. Interface vs Inheritance

Kotlin supports multiple **interfaces**, but only **single inheritance** for classes.

```
kotlinCopyEditinterface A {
    fun methodA()
}

interface B {
    fun methodB()
}

class MyClass : A, B {
    override fun methodA() = println("From A")
    override fun methodB() = println("From B")
}
```

---

9. Final Methods and Classes

Use `final` to prevent overriding (default behavior, but can be explicit).

```
kotlinCopyEditopen class Animal {
    final fun sleep() {
        println("Sleeping")
    }
}
```

✓Summary Table

Feature	Keyword	Purpose
Inheritable class	`open class`	Make a class inheritable
Inheritable method	`open fun`	Make a method overridable
Override method	`override`	Override parent's method
Call parent	`super`	Access superclass functionality
Prevent inheritance	`final`	Prevent further overriding
Partial implementation	`abstract`	Force child to implement method

In Kotlin, `lazy` is a **property delegate** used for **lazy initialization** — meaning the value is computed **only when it's first accessed**, and then **cached** for later use.

⍰ Syntax

```
kotlinCopyEditval myValue: String by lazy {
    println("Computed!")
    "Hello, Kotlin"
}
```

⍰ Here, `"Hello, Kotlin"` won't be created until you actually access `myValue`.

⍰ Example

```
kotlinCopyEditfun main() {
    val name: String by lazy {
        println("Calculating...")
        "Ankit"
    }

    println("Before accessing name")
    println(name)   // Lazy block runs here
    println(name)   // Uses cached value
}
```

⍰ Output:

```
mathematicaCopyEditBefore accessing name
Calculating...
Ankit
Ankit
```

⍰ How It Works

- `by lazy { ... }` creates a **delegate**.
- Runs only once, on **first access**.
- Thread-safe by default (can be customized).

⍰ Use Case Examples

✅1. Expensive Computation

```
kotlinCopyEditval config: Config by lazy {
    loadConfigFromFile()
}
```

✅2. Android Example (inside Activity)

```
kotlinCopyEditval userNameTextView: TextView by lazy {
    findViewById(R.id.userName)
}
```

⍰ Thread-Safety Modes

You can choose lazy behavior with modes:

```
kotlinCopyEditval value by lazy(LazyThreadSafetyMode.NONE) {
    // No thread safety (faster)
}
```

Mode	Description
`SYNCHRONIZED`	Thread-safe (default)
`PUBLICATION`	Multiple threads may compute, only one used
`NONE`	No thread safety

✓Summary

Feature	Description
`lazy`	Lazily initializes a `val` on first access
Caches result?	✓Yes (only computed once)
Keyword used	`by lazy { ... }`
Thread safe?	✓Yes (by default)
Used with	`val` only (not `var`)

⍰ 1. `this` Keyword in Kotlin

✓Meaning:

`this` refers to the **current object** of the class. It is used to:

- Access current class **properties** or **functions**
- Resolve **naming conflicts** between local and instance variables
- Pass the current object to another function or constructor

⍰ Example:

```
kotlinCopyEditclass Person(val name: String) {
    fun showName() {
        println("Name is: ${this.name}")
    }
}
```

Here, `this.name` is **optional** (you can just use `name`), but it's useful for clarity.

When Needed (Variable Conflict):

```
kotlinCopyEditclass Student(name: String) {
    val name: String

    init {
        this.name = name // distinguishes between parameter and property
    }
}
```

`this` in Extension Functions

```
kotlinCopyEditfun String.printUpperCase() {
    println(this.uppercase())
}
```

`this` refers to the string on which the function is called.

---

2. `super` Keyword in Kotlin

✓Meaning:

`super` refers to the **parent class** (superclass). It is used to:

- Access **parent class methods or properties**
- Call **parent class constructor**
- Use when overriding methods

Example:

```
kotlinCopyEditopen class Animal {
    open fun speak() {
        println("Animal sound")
    }
}

class Dog : Animal() {
    override fun speak() {
        super.speak()         // Calls Animal's speak()
        println("Dog barks")  // Dog's additional behavior
    }
}
```

⍰ super Constructor Call

```
kotlinCopyEditopen class Person(val name: String)

class Employee(name: String, val job: String) : Person(name)
```

You call the superclass constructor using `super()` automatically or explicitly (if needed).

⍰ Comparison Table

Keyword	Refers To	Used For
`this`	Current class	Accessing current object’s properties or methods
`super`	Parent class	Accessing overridden methods or constructor from superclass

---

⍰ Summary

- Use `this` when you want to refer to the current class object.
- Use `super` when you want to refer to the superclass behavior or constructor.

⍰ What is an Interface in Kotlin?

An **interface** is like a **contract** that a class agrees to follow. It defines **abstract functions** (no body) and **properties**, and the implementing class must provide their definitions.

✅Why Use Interfaces?

- To **achieve abstraction**
- To support **multiple inheritance**
- To define **common behavior** across unrelated classes

⍰ 1. Declaring an Interface

```
kotlinCopyEditinterface MyInterface {
    fun show()  // abstract function
    val id: Int // abstract property
}
```

- All functions and properties are **implicitly abstract** and **open** (no need for `abstract` or `open`).
- Interfaces **cannot hold state** (no backing fields for properties).

⍰ 2. Implementing an Interface in a Class

```
kotlinCopyEditclass MyClass : MyInterface {
    override fun show() {
        println("Implemented show()")
    }

    override val id: Int = 10
}
```

⍰ Use `override` for both properties and methods.

⍰ 3. Interface with Default Method Implementation

Kotlin allows default method implementations in interfaces.

```
kotlinCopyEditinterface A {
    fun greet() {
        println("Hello from A")
    }
}

class B : A
```

- Class `B` inherits the default behavior of `greet()` without overriding it.
- This is different from Java (before Java 8).

⍰ 4. Multiple Interfaces

Kotlin supports **multiple interface inheritance**.

```
kotlinCopyEditinterface A {
    fun display()
}

interface B {
    fun display()
}

class C : A, B {
    override fun display() {
        println("From both A and B")
    }
}
```

⍰ 5. Resolving Method Conflicts

If two interfaces have the same function with default implementation, Kotlin forces you to resolve the conflict.

```
kotlinCopyEditinterface A {
    fun info() {
        println("Info from A")
    }
}

interface B {
    fun info() {
        println("Info from B")
    }
}

class D : A, B {
    override fun info() {
        super<A>.info() // specify which one to use
        super<B>.info()
    }
}
```

---

6. Interface Properties

```
kotlinCopyEditinterface Shape {
    val name: String
    val sides: Int
        get() = 0 // default property implementation
}
```

If you give a property a default `get()`, it's not abstract anymore.

---

7. Interface vs Abstract Class

Feature	Interface	Abstract Class
Inheritance	Multiple	Single
Method Implementation	Allowed (default)	Allowed
State (Backing fields)	❌Not allowed	✓Allowed
Constructors	❌Not allowed	✓Allowed
Primary Use	Behavior abstraction	Partial implementation

---

8. Interface in Android Example

```
kotlinCopyEditinterface ClickListener {
    fun onClick()
}

class Button : ClickListener {
    override fun onClick() {
        println("Button clicked")
    }
}
```

Useful for **callbacks** or **event handling**.

⍰ 9. Interface Inheritance

Interfaces can **extend** other interfaces.

```
kotlinCopyEditinterface Printable {
    fun print()
}

interface Scannable : Printable {
    fun scan()
}

class Machine : Scannable {
    override fun print() {
        println("Printing")
    }

    override fun scan() {
        println("Scanning")
    }
}
```

⍰ Final Example: Everything Together

```
kotlinCopyEditinterface Animal {
    val type: String
    fun makeSound()
    fun sleep() {
        println("$type is sleeping")
    }
}

interface Pet {
    fun play()
}

class Dog : Animal, Pet {
    override val type = "Dog"

    override fun makeSound() {
        println("Bark!")
    }

    override fun play() {
        println("Dog is playing fetch")
    }
}

fun main() {
    val dog = Dog()
    dog.makeSound()
    dog.sleep()
    dog.play()
}
```

✓Summary

Topic	Description
`interface`	Defines abstract behavior
`override`	Used to implement interface methods/properties
Default Methods	You can define methods inside interface
Multiple Inheritance	Kotlin supports multiple interfaces
Conflict Resolution	Use `super<InterfaceName>.method()` to resolve
Interface Properties	Can have `get()` but no backing field

Absolutely! Let's break down the **difference between an abstract class and an interface in Kotlin** — this is a very common question in interviews and real-world Kotlin design.

✓Quick Overview

Feature	**Abstract Class**	**Interface**
Inheritance Support	Single inheritance only	Multiple inheritance allowed
Constructor	✓Can have constructors	✗Cannot have constructors
Method Body	✓Can have both abstract & concrete methods	✓Can have default methods (from Kotlin 1.1)
State (fields/properties)	✓Can have state (backing fields)	✗Cannot hold state (only abstract or `get()`)
Visibility Modifiers	✓Can be `protected`, `private`, etc.	✗Members are always `public` by default
Use Case	"Is-a" relationship (common base class)	Behavioral capability (like "can-do")
Object Creation	✗Cannot instantiate	✗Cannot instantiate

⍰ 1. Abstract Class

An **abstract class** is like a **partially implemented class** — it can have abstract methods (no body) and also concrete methods (with body). It may or may not contain abstract members.

⍰ Example:

```
kotlinCopyEditabstract class Animal(val name: String) {
    abstract fun sound()

    fun eat() {
        println("$name is eating")
    }
}
kotlinCopyEditclass Dog : Animal("Dog") {
    override fun sound() {
        println("Barks")
    }
}
```

✓Features:

- Has **constructor**
- Can **hold state** (`val name`)
- Can have **non-abstract methods**
- **Only one** abstract class can be inherited

⍰ 2. Interface

An **interface** defines a contract. It contains only abstract methods and properties (though default method implementations are allowed).

⍰ Example:

```
kotlinCopyEditinterface Runnable {
    fun run()
}
kotlinCopyEditclass Car : Runnable {
    override fun run() {
        println("Car is running")
    }
}
```

✓Features:

- No constructors
- No state (no real variables, only `val` with `get()` allowed)
- **Multiple interfaces** can be implemented

- Used for **common behavior across unrelated classes**

⍰ When to Use What?

Situation	Choose
You need to share code between related classes	✅Abstract Class
You need to define behavior for unrelated classes	✅Interface
You need to hold state (properties with values)	✅Abstract Class
You want to use multiple “capabilities” like `Runnable`, `Drawable`	✅Interface

⍰ Combining Both

You can even use **both**:

```
kotlinCopyEditabstract class Animal {
    abstract fun eat()
}

interface Pet {
    fun play()
}

class Dog : Animal(), Pet {
    override fun eat() = println("Dog eats")
    override fun play() = println("Dog plays")
}
```

✅Summary Table

Feature	Abstract Class	Interface
Inheritance	Single	Multiple
Constructors	✅Yes	❌No
Properties with state	✅Yes	❌No (can use `get()` only)
Method body	✅Yes (concrete + abstract)	✅Yes (default allowed)
Purpose	Common base structure	Define common behavior

⍰ What is an Abstract Class in Kotlin?

An `abstract class` is a class that **cannot be instantiated directly** and is meant to be **subclassed**. It can have:

- **Abstract members** (no body)
- **Concrete members** (with body)

- **Constructors**
- **State (variables)**

⍰ Syntax

```
kotlinCopyEditabstract class Animal(val name: String) {
    abstract fun makeSound()

    fun sleep() {
        println("$name is sleeping")
    }
}
```

⍰ 1. Key Characteristics

Feature	Abstract Class
Can have constructors	✓Yes
Can have state/variables	✓Yes
Can have concrete methods	✓Yes
Can have abstract methods	✓Yes
Can be inherited	✓Yes
Can be instantiated	✖No (only subclasses can use it)

⍰ 2. Example: Abstract Class with Subclass

```
kotlinCopyEditabstract class Animal(val name: String) {
    abstract fun sound()

    fun eat() {
        println("$name is eating.")
    }
}

class Dog : Animal("Dog") {
    override fun sound() {
        println("Dog barks")
    }
}

fun main() {
    val myDog = Dog()
    myDog.eat()       // Dog is eating.
    myDog.sound()     // Dog barks
}
```

---

3. Abstract Properties

You can define **abstract properties**, which must be overridden in the subclass.

```
kotlinCopyEditabstract class Vehicle {
    abstract val wheels: Int
    abstract fun drive()
}

class Car : Vehicle() {
    override val wheels = 4

    override fun drive() {
        println("Car is driving with $wheels wheels")
    }
}
```

---

4. Abstract Class vs Concrete Class

Concept	Abstract Class	Concrete Class
Instantiation	❌No	✅Yes
Incomplete Implementation	✅Yes (abstract methods)	❌No (everything is complete)
Usage	For base class / design / template	For objects

---

5. Constructor in Abstract Class

Yes, you can use constructors:

```
kotlinCopyEditabstract class Person(val name: String) {
    init {
        println("Person $name created")
    }

    abstract fun work()
}

class Teacher(name: String) : Person(name) {
    override fun work() {
        println("$name is teaching")
    }
}
```

⍰ 6. Real-World Use Case (Android Example)

```
kotlinCopyEditabstract class BaseActivity {
    abstract fun setLayout()

    fun initView() {
        println("View initialized")
    }
}

class MainActivity : BaseActivity() {
    override fun setLayout() {
        println("MainActivity layout set")
    }
}
```

⍰ 7. Abstract Class vs Interface (Quick Recall)

Feature	Abstract Class	Interface
Inheritance	Only one	Multiple allowed
Constructors	✅Yes	❌No
State (Properties with value)	✅Yes	❌No (only get(), no fields)
Default Methods	✅Yes	✅Yes

✅When to Use Abstract Class?

Use `abstract class` when:

- You want to share **code** and **state**
- You need to **define a base class** for other classes
- You want to **partially implement** logic and let subclasses finish the rest

---

⍰ Bonus: Abstract Class with Interface

```
kotlinCopyEditinterface Flyer {
    fun fly()
}

abstract class Bird(val name: String) {
    abstract fun sing()
}

class Parrot : Bird("Parrot"), Flyer {
    override fun sing() {
        println("Parrot sings")
    }
    override fun fly() {
        println("Parrot flies")
    }
}
```

✓Summary

Topic	Meaning
Abstract class	A class with incomplete methods meant to be subclassed
Can have constructor	✓Yes
Can have body and state	✓Yes
Can be instantiated?	✗No
Needs subclass to work	✓Yes

What is an `enum` in Kotlin?

An `enum` (short for **enumeration**) is a **special class** used to represent a **group of constant values**.

For example:

```
kotlinCopyEditenum class Direction {
    NORTH, SOUTH, EAST, WEST
}
```

Here, `Direction` is an enum class with 4 fixed values.

Syntax

```
kotlinCopyEditenum class EnumName {
    VALUE1, VALUE2, VALUE3
}
```

✓Simple Example

```
kotlinCopyEditenum class Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

fun main() {
    val today = Day.FRIDAY
    println("Today is $today")
}
```

⍰ Features of Enums in Kotlin

Feature	Supported
Fixed set of constants	✓Yes
Can have properties	✓Yes
Can have methods	✓Yes
Can implement interfaces	✓Yes
Cannot inherit other classes	✗No

⍰ Enum with Properties and Methods

You can add properties and functions to enum constants:

```
kotlinCopyEditenum class Planet(val mass: Double, val radius: Double) {
    EARTH(5.97, 6371.0),
    MARS(0.642, 3389.5),
    JUPITER(1898.0, 69911.0);

    fun surfaceGravity(): Double {
        val G = 6.674 * 1e-11
        return G * mass / (radius * radius)
    }
}

fun main() {
    val planet = Planet.EARTH
    println("Gravity on ${planet.name} = ${planet.surfaceGravity()}")
}
```

⍰ Enum with `when` Statement

```
kotlinCopyEditenum class TrafficLight {
    RED, YELLOW, GREEN
}

fun action(light: TrafficLight) {
    when (light) {
        TrafficLight.RED -> println("Stop")
        TrafficLight.YELLOW -> println("Ready")
        TrafficLight.GREEN -> println("Go")
    }
}
```

▯ Enum Iteration

You can loop through all enum constants using `.values()`:

```
kotlinCopyEditfor (day in Day.values()) {
    println(day)
}
```

▯ Enum Properties

- `name`: String value of the enum
- `ordinal`: Index of the enum constant (starting from 0)

```
kotlinCopyEditval d = Day.MONDAY
println(d.name)     // Output: MONDAY
println(d.ordinal)  // Output: 0
```

▯ Enum Implementing Interface

```
kotlinCopyEditinterface Drawable {
    fun draw()
}

enum class Shape : Drawable {
    CIRCLE {
        override fun draw() = println("Drawing circle")
    },
    SQUARE {
        override fun draw() = println("Drawing square")
    }
}
```

▯ Real-World Use Case

Imagine you are building an Android app that shows weather status:

```
kotlinCopyEditenum class Weather {
    SUNNY, CLOUDY, RAINY, SNOWY
}

fun showMessage(weather: Weather) {
    when (weather) {
        Weather.SUNNY -> println("Wear sunglasses")
        Weather.CLOUDY -> println("Maybe take a jacket")
        Weather.RAINY -> println("Carry an umbrella")
        Weather.SNOWY -> println("Wear warm clothes")
    }
}
```

✅Summary

Feature	Description
Enum Definition	`enum class Name { VALUE1, VALUE2 }`
Properties	`name`, `ordinal`
Custom Properties	Yes (like `mass`, `radius`, etc.)
Methods	Yes (functions inside enum)
Implements Interface	Yes
Use `when`	Yes

⍰ What Are Generics?

Generics allow classes, functions, and interfaces to **operate on different data types** without repeating the same code for each type.

> ⍰ Think of it as writing one flexible version of code that works with **any type** (like `Int`, `String`, `Float`, custom types, etc.).

✅Basic Generic Class

```
kotlinCopyEditclass Box<T>(val value: T) {
    fun getValue(): T {
        return value
    }
}
```

- `T` is a **type parameter**.
- You can create a `Box` of **any type**:

```
kotlinCopyEditval intBox = Box(123)         // Box<Int>
val stringBox = Box("Hello")  // Box<String>

println(intBox.getValue())    // 123
println(stringBox.getValue()) // Hello
```

⍰ Why Use Generics?

1. **Type Safety** – Catches errors at compile-time.
2. **Code Reusability** – One class/method works for all types.
3. **Cleaner Code** – No need to write multiple overloads.

⍰ Generic Class with Two Parameters

```
kotlinCopyEditclass PairBox<A, B>(val first: A, val second: B) {
    fun printTypes() {
        println("First: $first, Second: $second")
    }
}

val pair = PairBox("Age", 25)
pair.printTypes() // First: Age, Second: 25
```

⍰ Generic Functions

```
kotlinCopyEditfun <T> printItem(item: T) {
    println("Item: $item")
}

printItem("Hi")    // String
printItem(99)      // Int
printItem(3.14)    // Double
```

⍰ Bounded Generics (Upper Bound)

You can restrict generics to **a certain type or its subtypes** using `:`.

```
kotlinCopyEditfun <T : Number> doubleValue(value: T): Double {
    return value.toDouble() * 2
}

println(doubleValue(10))      // OK (Int is Number)
println(doubleValue(5.5))     // OK (Double is Number)
// println(doubleValue("Hi")) // ❌Error
```

⍰ Generic Interfaces

```
kotlinCopyEditinterface Storage<T> {
    fun save(item: T)
    fun get(): T
}

class StringStorage : Storage<String> {
    private var item: String = ""

    override fun save(item: String) {
        this.item = item
    }
    override fun get(): String = item
}
```

Variance: `in` and `out`

Kotlin uses **variance** to control how generics are used with inheritance.

`out` – Covariant (Read-Only)

```
kotlinCopyEditclass Producer<out T>(val item: T) {
    fun produce(): T = item
}
```

`in` – Contravariant (Write-Only)

```
kotlinCopyEditclass Consumer<in T> {
    fun consume(item: T) {
        println("Consumed: $item")
    }
}
```

Reified Type Parameters (only in inline functions)

```
kotlinCopyEditinline fun <reified T> isOfType(value: Any): Boolean {
    return value is T
}

println(isOfType<String>("Kotlin")) // true
println(isOfType<Int>("Kotlin"))    // false
```

✅Summary Table

Concept	Description
`<T>`	Type parameter
`Box<T>`	Generic class
`fun <T> myFun()`	Generic function
`T : Number`	Bounded type
`in`	Accept values of T (consumer)
`out`	Return values of T (producer)
`reified`	Retain type info at runtime (inline only)

Final Example

```
kotlinCopyEditclass Stack<T> {
    private val items = mutableListOf<T>()
    fun push(item: T) = items.add(item)
    fun pop(): T = items.removeAt(items.size - 1)
}

val intStack = Stack<Int>()
intStack.push(10)
intStack.push(20)
println(intStack.pop()) // 20
```

⍰ What is a Lambda?

A **lambda expression** is an **anonymous function** that you can treat as a value — pass it, return it, store it, etc.

> In simple terms: A lambda is a **short function** you can pass around like a variable.

⍰ Basic Syntax

```
kotlin


CopyEdit
val lambdaName: (InputType) -> ReturnType = { input -> result }
```

✓Example:

```
kotlinCopyEditval square: (Int) -> Int = { number -> number * number }

println(square(5))  // Output: 25
```

✨Simplified Lambda Usage

```
kotlinCopyEditval greet = { println("Hello Kotlin!") }
greet()  // Output: Hello Kotlin!
```

⍰ Lambdas with Parameters

```
kotlinCopyEditval add = { a: Int, b: Int -> a + b }
println(add(10, 5)) // Output: 15
```

⍰ Lambdas as Function Parameters

```
kotlinCopyEditfun operate(a: Int, b: Int, op: (Int, Int) -> Int): Int {
    return op(a, b)
}
```

```
val result = operate(3, 4, { x, y -> x + y })
println(result) // Output: 7
```

✅Kotlin shorthand:

```
kotlin


CopyEdit
val result = operate(3, 4) { x, y -> x + y }  // Lambda outside the parentheses
```

□□ `it` keyword in Kotlin

If your lambda takes **only one parameter**, Kotlin provides `it` automatically:

```
kotlinCopyEditval printUpper = { str: String -> println(str.uppercase()) }
val printIt = { println(it.uppercase()) } // Same as above, uses 'it'

printIt("hello") // Output: HELLO
```

□ Lambda Returning Value

```
kotlinCopyEditval isEven: (Int) -> Boolean = { it % 2 == 0 }
println(isEven(4)) // true
```

□ Working with Collections

□ `map`

```
kotlinCopyEditval nums = listOf(1, 2, 3)
val squares = nums.map { it * it }
println(squares) // [1, 4, 9]
```

□ `filter`

```
kotlinCopyEditval evens = nums.filter { it % 2 == 0 }
println(evens) // [2]
```

□ `forEach`

```
kotlin


CopyEdit
nums.forEach { println(it) }
```

Higher-Order Functions

A function that **takes a lambda** or **returns a lambda** is called a **higher-order function**.

```
kotlinCopyEditfun calculator(op: (Int, Int) -> Int): Int {
    return op(5, 3)
}

val sub = calculator { a, b -> a - b }
println(sub) // 2
```

Lambdas Inside Classes

```
kotlinCopyEditclass Button {
    var onClick: (() -> Unit)? = null

    fun click() {
        onClick?.invoke()
    }
}

val btn = Button()
btn.onClick = { println("Button clicked!") }
btn.click() // Output: Button clicked!
```

Lambda with `return`

```
kotlinCopyEditfun greetPeople(names: List<String>) {
    names.forEach {
        if (it == "John") return@forEach  // labeled return to skip John
        println("Hello, $it")
    }
}
```

Inline Lambda Functions

```
kotlinCopyEditinline fun measure(block: () -> Unit) {
    val start = System.currentTimeMillis()
    block()
    val end = System.currentTimeMillis()
    println("Time taken: ${end - start}ms")
}

measure {
    println("Doing something...")
}
```

✅Summary

Concept	Example
Basic lambda	`{ a: Int -> a * a }`
With function param	`operate(2, 3) { a, b -> a + b }`
Single param (`it`)	`{ println(it) }`
Used in collections	`map`, `filter`, `forEach`, etc.
Returning lambdas	`fun returnLambda(): () -> Int`
Inline function	`inline fun doSomething(block: () -> Unit)`

⍰ Practice Task Ideas

1. Create a `List<String>` and filter names starting with `'A'`.
2. Write a function that takes a lambda to multiply two numbers.
3. Sort a list of objects using a lambda on a property.
4. Create a mini calculator using lambdas (`add`, `subtract`, etc.)

✅Project: Filter Names Starting with a Specific Letter

⍰ Objective:

- Create a list of names.
- Use a **lambda function** to filter out names starting with a specific letter.
- Print the filtered names.

⍰⍰ Full Code:

```
kotlinCopyEditfun main() {
    val names = listOf("Ankit", "Amit", "Bharat", "Ajay", "Ravi", "Alok",
"Ramesh")

    println("Enter starting letter to filter names:")
    val input = readLine()?.uppercase() ?: "A"

    // Lambda function to filter names
    val filterNames: (String) -> Boolean = { it.uppercase().startsWith(input)
}

    val filteredList = names.filter(filterNames)

    println("Names starting with '$input':")
    filteredList.forEach { println(it) }
}
```

How It Works:

- The `filterNames` lambda checks if each name starts with the user input.
- The `filter` function uses this lambda to return only matching names.
- `forEach` prints the final result.

Sample Output:

```
cssCopyEditEnter starting letter to filter names:
A
Names starting with 'A':
Ankit
Amit
Ajay
Alok
```

What Are Higher-Order Functions?

A **Higher-Order Function** is a function that either:

1. Takes one or more functions as parameters.
2. Returns a function.

Why Use Them?

- They allow **flexibility and reusability**.
- They're the foundation of **functional programming**.
- Help in building **custom behaviors** like event handling, filtering, mapping, etc.

Syntax

```
kotlinCopyEditfun highOrderFunction(param: (Type) -> ReturnType): OutputType
{
    // use param as a function
}
```

✓Example 1: Passing a Lambda as a Parameter

```
kotlinCopyEditfun operate(a: Int, b: Int, op: (Int, Int) -> Int): Int {
    return op(a, b)
}

fun main() {
    val sum = operate(10, 5) { x, y -> x + y }
    val product = operate(10, 5) { x, y -> x * y }

    println("Sum: $sum")        // Output: 15
    println("Product: $product")// Output: 50
}
```

✅Example 2: Function Returning Another Function

```
kotlinCopyEditfun getGreetingFunction(): () -> String {
    return { "Hello from returned function!" }
}

fun main() {
    val greet = getGreetingFunction()
    println(greet())  // Output: Hello from returned function!
}
```

✅Example 3: Named Function as Parameter

```
kotlinCopyEditfun square(x: Int): Int = x * x

fun applyFunction(num: Int, func: (Int) -> Int): Int {
    return func(num)
}

fun main() {
    val result = applyFunction(4, ::square)
    println(result)  // Output: 16
}
```

✅Real Use Case: Filtering a List

```
kotlinCopyEditfun filterList(list: List<String>, condition: (String) ->
Boolean): List<String> {
    return list.filter(condition)
}

fun main() {
    val names = listOf("Ankit", "Amit", "Ravi", "Ajay")
    val filtered = filterList(names) { it.startsWith("A") }
    println(filtered)  // Output: [Ankit, Amit, Ajay]
}
```

Inline Functions (with Higher-Order Functions)

Kotlin allows you to mark higher-order functions as `inline` for performance:

```
kotlinCopyEditinline fun log(block: () -> Unit) {
    println("Before")
    block()
    println("After")
}

fun main() {
    log {
        println("Inside the block")
    }
}
```

⍰ `crossinline` & `noinline`

- `crossinline`: Prevents non-local returns.
- `noinline`: Tells the compiler **not to inline** that specific lambda.

Example:

```
kotlinCopyEditinline fun runOperation(crossinline op: () -> Unit) {
    val thread = Thread {
        op()  // Valid because crossinline prevents non-local return
    }
    thread.start()
}
```

⍰ Common Collection Functions (that use higher-order logic)

Function	Purpose
`map`	Transform elements
`filter`	Keep elements that match
`forEach`	Perform action for each item
`reduce`	Combine all into one
`sortedBy`	Sort using custom rule

✓Summary

Feature	Example
Takes function as input	`operate(2, 3, { a, b -> a + b })`
Returns a function	`fun getFunc(): () -> String`
Built-in uses	`map`, `filter`, `forEach`
Performance tip	Use `inline`

⍰ What are Scope Functions?

In Kotlin, **scope functions** allow you to execute a block of code within the context of an object. They help you:

- Access the object without repeating its name.
- Modify the object or perform operations on it.
- Return the object itself or the result of the lambda.

⍰ The 5 Main Scope Functions

Function	Object Reference	Return Value	Typical Use
`let`	`it`	Lambda result	Transform or avoid null
`run`	`this`	Lambda result	Compute or initialize
`with`	`this`	Lambda result	Group actions on object
`apply`	`this`	Object itself	Configure object
`also`	`it`	Object itself	Additional actions

---

1. `let` – for transforming or avoiding null

```
kotlinCopyEditval name: String? = "Ankit"

name?.let {
    println("Name is $it")
}
```

⍰ **Think:** “Do something *with* the object and return something else.”

Use Case: Useful for null checks, transformations.

---

2. `run` – for executing a block and returning a result

```
kotlinCopyEditval length = "Ankit".run {
    println("The name is $this")
    length
}
```

⍰ **Think:** “Run some code in the object’s context and return a result.”

Use Case: Combining object actions with a final return value.

---

3. `with` – for grouping multiple operations

```
kotlinCopyEditval person = Person("Ankit", 22)

val description = with(person) {
    "Name: $name, Age: $age"
}
```

 Think: “Apply multiple things on an object, then return something.”

Use Case: Cleaner syntax for multiple operations on the same object.

4. apply – for configuring objects

```
kotlinCopyEditval person = Person().apply {
    name = "Ankit"
    age = 22
}
```

 Think: “Apply changes and return the object itself.”

Use Case: Object creation and initialization (like builder pattern).

5. also – for doing extra work (like logging)

```
kotlinCopyEditval name = "Ankit".also {
    println("Name is $it")
}
```

 Think: “Also do something and return the same object.”

Use Case: Logging, debugging, extra steps without changing the object.

Quick Memory Trick

Function	Use For	Use `this` or `it`	Returns
`let`	Transforming	`it`	Lambda result
`run`	Running logic	`this`	Lambda result
`with`	Grouping actions	`this`	Lambda result
`apply`	Initializing	`this`	Object itself
`also`	Side-effects	`it`	Object itself

✓Summary Example:

```
kotlinCopyEditval student = Student().apply {
    name = "Ankit"
    age = 20
}.also {
    println("Created student: ${it.name}")
}.let {
    "Student name is ${it.name} and age is ${it.age}"
}
```

?What does `?` mean in Kotlin?

In Kotlin, the **?** symbol is used to **handle nullability**. It tells the compiler that **a variable can hold a null value**.

✓1. Nullable Type

```
kotlinCopyEditval name: String = "Ankit"  // Non-nullable
val nickName: String? = null  // Nullable because of `?`
```

□ Without `?`, the variable **cannot** be `null`.

□ 2. Why Do We Need `?`?

Kotlin was designed to prevent the **“NullPointerException”** (NPE), which is a common crash in Java apps.

By using `?`, Kotlin forces you to **handle null cases** safely.

□ 3. Safe Call Operator (`?.`)

```
kotlinCopyEditval name: String? = "Ankit"
println(name?.length) // Prints: 5
```

- `?.` means “if the value is **not null**, then call the method.”
- If it’s `null`, it simply returns `null` instead of crashing.

□ 4. Elvis Operator (`?:`) – Provide a default

```
kotlinCopyEditval name: String? = null
val finalName = name ?: "Unknown"
println(finalName) // Output: Unknown
```

□ It means: “If `name` is null, use `"Unknown"` instead.”

□ 5. Not-null Assertion (`!!`) – Dangerous!

```
kotlinCopyEditval name: String? = null
val length = name!!.length // Throws NullPointerException!
```

□ `!!` says: “I’m **sure** it’s not null.” But if it is, your app **crashes**.

⍰ 6. Use with `let` for null-safe code

```
kotlinCopyEditval name: String? = "Ankit"

name?.let {
    println("Length: ${it.length}")
}
```

⍰ Only runs if `name` is not null.

⍰ In Summary:

Syntax	Meaning	Safe?
`?`	Nullable type (`String?`)	✓Yes
`?.`	Safe call	✓Yes
`?:`	Elvis operator (default)	✓Yes
`!!`	Not-null assertion	✗No (use with caution)

⍰ What Are Collections in Kotlin?

Collections in Kotlin are used to **store groups of related data** — like lists of names, ages, scores, etc.

There are 3 main types:

Type	Description	Example
`List`	Ordered, can contain duplicates	listOf(1, 2, 3)
`Set`	Unordered, no duplicates	setOf(1, 2, 2, 3) → [1, 2, 3]
`Map`	Key-value pairs (like a dictionary)	mapOf("A" to 1, "B" to 2)

⍰ Mutable vs Immutable

- `listOf()`, `setOf()`, `mapOf()` → **Immutable** (cannot change)
- `mutableListOf()`, `mutableSetOf()`, `mutableMapOf()` → **Mutable** (can add, remove, update)

✓Let's Explore All Operations with Examples

⍰ 1. List Operations

```
kotlinCopyEditfun main() {
    val myList = mutableListOf("Apple", "Banana", "Cherry")
    // Add
    myList.add("Date")
    println(myList) // [Apple, Banana, Cherry, Date]

    // Remove
    myList.remove("Banana")
    println(myList) // [Apple, Cherry, Date]

    // Update
    myList[1] = "Coconut"
    println(myList) // [Apple, Coconut, Date]

    // Access
    println(myList[0]) // Apple

    // Size
    println(myList.size) // 3

    // Loop
    for (item in myList) {
        println(item)
    }
}
```

2. Set Operations

```
kotlinCopyEditfun main() {
    val mySet = mutableSetOf("Apple", "Banana", "Apple", "Cherry")

    println(mySet) // [Apple, Banana, Cherry] - no duplicates

    // Add
    mySet.add("Date")
    println(mySet)

    // Remove
    mySet.remove("Banana")
    println(mySet)

    // Check
    println(mySet.contains("Apple")) // true

    // Loop
    for (item in mySet) {
        println(item)
    }
}
```

---

3. Map Operations

```
kotlinCopyEditfun main() {
    val myMap = mutableMapOf("A" to "Apple", "B" to "Banana")

    // Add
    myMap["C"] = "Cherry"
    println(myMap) // {A=Apple, B=Banana, C=Cherry}

    // Update
    myMap["A"] = "Avocado"
    println(myMap) // {A=Avocado, B=Banana, C=Cherry}

    // Remove
    myMap.remove("B")
    println(myMap) // {A=Avocado, C=Cherry}

    // Access
    println(myMap["A"]) // Avocado

    // Loop
    for ((key, value) in myMap) {
        println("$key → $value")
    }
}
```

Summary Table of Common Functions

Collection	Add	Remove	Update	Access
List	`add()`	`remove()`	`list[i] = value`	`list[i]`
Set	`add()`	`remove()`	No update by index	`contains()`
Map	`map[k]=v`	`remove(k)`	`map[k] = newValue`	`map[k]`

Tip

- Use **List** when you need ordered data (like a playlist).
- Use **Set** when you want **unique** items.
- Use **Map** when you need **key-value** lookup (like ID → Name).

Would you like a **visual diagram** for Collections too? Or a small **project** to practice them in a real app?

You said:

is i can store only similar type of data

ChatGPT said:

Great question! Let’s explore this like you’re in a Kotlin classroom ⍰⍰.

?Can We Store Only Similar Type of Data in Kotlin Collections?

✓Short Answer:

No, you can store **different types of data** in Kotlin collections **if you want**, but **only if** you declare them that way.

⍰ 1. Homogeneous Collections (same type) – Most common

```
kotlinCopyEditval numbers = listOf(1, 2, 3, 4)         // List<Int>
val names = listOf("Ankit", "Aman")     // List<String>
```

Here, Kotlin **infers the type** and you can only store that type in the collection.

Trying to do this will throw an error:

```
kotlin


CopyEdit
val mix = listOf(1, "Hello") // ✕Error if declared as List<Int>
```

⍰ 2. Heterogeneous Collections (mixed types)

If you really want to store **different types**, you can use `Any` (the super type of all types in Kotlin):

```
kotlinCopyEditval mixedList: List<Any> = listOf(1, "Hello", 3.14, true)

for (item in mixedList) {
    println("$item → ${item::class.simpleName}")
}
```

✓Output:

```
kotlinCopyEdit1 → Int
Hello → String
3.14 → Double
true → Boolean
```

⍰ But Be Careful:

Using `List<Any>` means you **lose type safety** — you'll need to check the type at runtime using:

```
kotlin


CopyEdit
if (item is String) { ... }
```

So, unless you really need it, it's best to use **typed collections**.

✅Best Practice

- Use `List<String>` or `List<Int>` when you expect one type → safer, cleaner.
- Use `List<Any>` only when your logic really needs different types.

Would you like me to show how to use `is` and `when` with `List<Any>` to safely access items based on their type?

You said:

if in my set or list i want delete a particular data or want enter a data according my choice index

ChatGPT said:

Excellent question! Let's break it down into two parts just like a teacher would explain on a board:

⍰ 1. Can I Delete a Particular Item from a `List` or `Set` in Kotlin?

✅Yes, you can delete by value or index.

For `List` (since it's ordered and index-based):

```
kotlinCopyEditval myList = mutableListOf("A", "B", "C", "D")

// Remove by value
myList.remove("C")
println(myList) // [A, B, D]

// Remove by index
```

```
myList.removeAt(1)
println(myList) // [A, D]
```

For Set (no duplicates, no index):

```
kotlinCopyEditval mySet = mutableSetOf("Apple", "Banana", "Cherry")

// Remove by value only (no index in Set)
mySet.remove("Banana")
println(mySet) // [Apple, Cherry]
```

✗You **cannot remove by index** in a Set because it doesn't have index positions.

⍰ 2. Can I Insert Data at a Specific Index?

✓Yes, in List using `add(index, value)`.

```
kotlinCopyEditval myList = mutableListOf("A", "B", "C")

// Insert at index 1
myList.add(1, "X")
println(myList) // [A, X, B, C]
```

✗No, in Set — because it's unordered, there's no concept of "index".

```
kotlinCopyEditval mySet = mutableSetOf("A", "B", "C")

// You can only add like this:
mySet.add("X")
println(mySet) // [A, B, C, X] → Position not guaranteed
```

⍰ Summary Table

Action	List ✓	Set ✓/✗
Add item at end	✓`add()`	✓`add()`
Add item at specific index	✓`add(index, value)`	✗
Remove item by value	✓`remove(value)`	✓
Remove item by index	✓`removeAt(index)`	✗
Update item at index	✓`list[index] = value`	✗

So, can you delete or add at a specific index in a Kotlin List?

✓Yes, lists are ordered, so you can totally control position.

Example:

```
kotlinCopyEditval myList = mutableListOf("A", "B", "C")

// Insert at index 1
myList.add(1, "X") // Now: [A, X, B, C]

// Remove by value
myList.remove("B") // Now: [A, X, C]

// Remove by index
myList.removeAt(0) // Now: [X, C]

// Update value at index
myList[1] = "Z"     // Now: [X, Z]
```

⍰ What about a Set?

Set doesn't keep order. You can:

- ✅Add
- ✅Remove by value
- ❌No index-based control (so can't add or remove at a specific position)

Example:

```
kotlinCopyEditval mySet = mutableSetOf("Apple", "Banana", "Cherry")

mySet.add("Date")           // adds
mySet.remove("Banana")      // removes
// mySet.removeAt(1) ❌not allowed
```

So in short:

- Use `List` when you care about **order or position**
- Use `Set` when you care about **uniqueness**, not order

⍰ What is a `data class` in Kotlin?

A `data class` is just a class that's **designed to hold data**.

Kotlin automatically gives you useful stuff like:

- `toString()`
- `equals()`
- `hashCode()`
- `copy()`

without writing all the boilerplate code yourself.

⍰ Example:

```
kotlinCopyEditdata class User(val name: String, val age: Int)

val user1 = User("Ankit", 22)

println(user1)  // Output: User(name=Ankit, age=22)
```

⍰ Why it's cool?

1. **Auto `toString()`**
    - You get a nice string representation automatically.
2. **Auto `equals()`**
    - Two users with the same data are considered equal.

```
kotlinCopyEditval user2 = User("Ankit", 22)
println(user1 == user2)  // true ✅
```

1. **Auto `copy()`**
    - Make a copy with changed values easily.

```
kotlinCopyEditval updatedUser = user1.copy(age = 23)
println(updatedUser)  // User(name=Ankit, age=23)
```

1. **Destructuring**
    - You can break it into variables quickly.

```
kotlinCopyEditval (name, age) = user1
println(name)  // Ankit
println(age)   // 22
```

⍰ When to use a data class?

Use it when your class is **just for holding data** — no business logic or heavy functionality inside.

⍰ Rules for data classes

- Must have at least **one primary constructor parameter**.
- Cannot be **abstract**, **open**, **sealed**, or **inner**.

Let me know if you want me to show how to use it with a list or use it in a real example like a product or contact model.

2/2

You said:

can you explain it with realworld examples

ChatGPT said:

Absolutely! Let’s take **real-world examples** you can relate to — no fluff, just clean usage of data classes ⍰

✅1. User Profile (like in a social media app)

```
kotlinCopyEditdata class UserProfile(
    val username: String,
    val email: String,
    val age: Int,
    val isVerified: Boolean
)

val user = UserProfile("ankit_007", "ankit@example.com", 22, true)

println(user.username)   // ankit_007
println(user)            // UserProfile(username=ankit_007, email=...,
age=22, isVerified=true)
```

✅2. Product in an E-commerce App

```
kotlinCopyEditdata class Product(
    val id: Int,
    val name: String,
    val price: Double,
    val inStock: Boolean
)

val product1 = Product(101, "Wireless Mouse", 499.99, true)
val product2 = product1.copy(price = 450.0)  // Sale price

println(product2)  // Product(id=101, name=Wireless Mouse, price=450.0,
inStock=true)
```

✅3. Location Data in a Maps App

```
kotlinCopyEditdata class Location(
    val latitude: Double,
    val longitude: Double,
    val name: String
)

val loc = Location(28.6139, 77.2090, "New Delhi")
println("Latitude: ${loc.latitude}, Location: ${loc.name}")
```

✅4. Contacts in a Phone Book App

```
kotlinCopyEditdata class Contact(
    val name: String,
    val phoneNumber: String,
    val isFavorite: Boolean
)

val contact = Contact("Aman", "+91-1234567890", true)
val (name, number, favorite) = contact

println("Name: $name, Number: $number, Fav: $favorite")
```

✅5. Student in a School App

```
kotlinCopyEditdata class Student(
    val rollNo: Int,
    val name: String,
    val marks: Int
)

val student1 = Student(1, "Priya", 87)
val student2 = Student(1, "Priya", 87)

println(student1 == student2) // true ✅because data is same
```

Summary:

Use Case	Example Data Class Name
Social Media	`UserProfile`
E-commerce	`Product`
Maps / Travel	`Location`
Phonebook	`Contact`
Education	`Student`

Perfect — let's break down **file handling in Kotlin** step-by-step with real-world examples. You'll see how to:

- Create and write to a file

- Read from a file
- Append content
- Check if file exists
- Delete a file

⍰ Kotlin makes file handling super easy using Java’s `File` class from `java.io`.

⍰ First, import this:

```
kotlin


CopyEdit
import java.io.File
```

✅1. Create a File & Write Content (e.g. Save user feedback)

```
kotlinCopyEditval file = File("feedback.txt")

file.writeText("User: Ankit\nFeedback: Great app experience!")

println("File created and feedback written.")
```

⍰ This will create the file `feedback.txt` and write the text inside it.

✅2. Read Content from File (e.g. Show saved feedback)

```
kotlinCopyEditval readData = file.readText()

println("Feedback content:\n$readData")
```

✅It will show:

```
yamlCopyEditFeedback content:
User: Ankit
Feedback: Great app experience!
```

✅3. Append More Data (e.g. Another user gives feedback)

```
kotlin


CopyEdit
file.appendText("\nUser: Priya\nFeedback: Loved the UI!")
```

Now the file will contain feedback from both users.

✅4. Read File Line-by-Line (e.g. loop through all feedbacks)

```
kotlinCopyEditfile.forEachLine { line ->
    println(line)
}
```

Useful when you want to handle each line separately (like log data, or student records).

✅5. Check if File Exists

```
kotlinCopyEditif (file.exists()) {
    println("Yes, the file exists!")
} else {
    println("No file found.")
}
```

✅6. Delete a File (e.g. admin deletes old logs)

```
kotlinCopyEditval deleted = file.delete()

if (deleted) {
    println("File deleted successfully.")
} else {
    println("File deletion failed.")
}
```

✅Real-World Example: Saving Login Logs

```
kotlinCopyEditval loginLog = File("login_logs.txt")

fun logLogin(username: String) {
    val entry = "User $username logged in at ${System.currentTimeMillis()}"
    loginLog.appendText("$entry\n")
    println("Logged: $entry")
}

logLogin("ankit007")
logLogin("priya88")
```

▯ Extra Tip: File Permissions

```
kotlinCopyEditprintln("Can read: ${file.canRead()}")
println("Can write: ${file.canWrite()}")
```

Let me know if you want to work with **CSV files**, **JSON**, or even store `data class` objects into files!

You said:

can you teach me if i want know where actually my files saved like i want find the exact location through programs

ChatGPT said:

Absolutely! If you want to **find the exact path where your file is saved**, Kotlin makes it easy using the `absolutePath` or `canonicalPath` of the `File` object.

Let's break it down ⍰

✓1. Get Exact File Path

```
kotlinCopyEditimport java.io.File

fun main() {
    val file = File("mydata.txt")
    file.writeText("Hello, this is a test!")

    println("Absolute Path: ${file.absolutePath}")
    println("Canonical Path: ${file.canonicalPath}")
}
```

⍰ Difference:

- `absolutePath`: Shows the full path **even if** the file doesn't exist yet.
- `canonicalPath`: Resolves symbolic links or `..` in path — it's cleaner and real-world accurate.

✓Example Output (on Windows):

```
mathematicaCopyEditAbsolute Path:
C:\Users\Ankit\IdeaProjects\MyApp\mydata.txt
Canonical Path: C:\Users\Ankit\IdeaProjects\MyApp\mydata.txt
```

✓Example Output (on Linux/Mac):

```
mathematicaCopyEditAbsolute Path: /home/ankit/mydata.txt
Canonical Path: /home/ankit/mydata.txt
```

⍰ Want to write it inside a specific folder?

```
kotlinCopyEditval file = File("C:/MyKotlinData/logs.txt") // For Windows
// or
val file = File("/home/ankit/logs.txt") // For Linux/Mac
```

```
file.writeText("Custom path log")
println("Saved at: ${file.absolutePath}")
```

> ⍰ Make sure the folder exists — otherwise, it will throw an error.

✓Create folder if it doesn’t exist:

```
kotlinCopyEditval folder = File("logs")
if (!folder.exists()) {
    folder.mkdir()  // or .mkdirs() for nested dirs
}

val logFile = File(folder, "user_log.txt")
logFile.writeText("New log entry!")
println("Log file saved at: ${logFile.absolutePath}")
```

Absolutely! Let’s talk about **taking input from the console after the program starts running** — this is super useful for making interactive programs.

✓Console Input in Kotlin (Using `readLine()`)

`readLine()` is a function that reads **one line of input** from the console as a `String`.

⍰ Example: Add two numbers entered by user

```
kotlinCopyEditfun main() {
    print("Enter first number: ")
    val input1 = readLine()!!.toInt()

    print("Enter second number: ")
    val input2 = readLine()!!.toInt()

    val sum = input1 + input2
    println("The sum is: $sum")
}
```

✓Step-by-step Breakdown:

- `readLine()` → reads input as a `String?`
- `!!` → says “I’m sure it’s not null”
- `.toInt()` → converts the string to an integer

▯ Example with Name Input:

```
kotlinCopyEditfun main() {
    print("What's your name? ")
    val name = readLine()!!

    println("Hello, $name! Welcome to Kotlin ▯")
}
```

✅Handle Different Data Types:

```
kotlinCopyEditfun main() {
    print("Enter your age: ")
    val age = readLine()!!.toInt()

    print("Enter your height in cm: ")
    val height = readLine()!!.toDouble()

    println("You're $age years old and $height cm tall.")
}
```

! Safe Input (Optional: With Null Checks)

If you don’t want to crash your app when user enters nothing or letters instead of numbers:

```
kotlinCopyEditfun main() {
    print("Enter a number: ")
    val input = readLine()

    val number = input?.toIntOrNull()

    if (number != null) {
        println("Square is: ${number * number}")
    } else {
        println("Invalid input! Please enter a number.")
    }
}
```

✅Real World Example: Simple Calculator

```
kotlinCopyEditfun main() {
    print("Enter first number: ")
    val num1 = readLine()!!.toDouble()

    print("Enter operator (+, -, *, /): ")
    val operator = readLine()!!
    print("Enter second number: ")
    val num2 = readLine()!!.toDouble()

    val result = when (operator) {
        "+" -> num1 + num2
        "-" -> num1 - num2
        "*" -> num1 * num2
        "/" -> if (num2 != 0.0) num1 / num2 else "Cannot divide by zero"
        else -> "Invalid operator"
    }

    println("Result: $result")
}
```