ANDROID

SESSION TWO

Agenda

- Intro To OOP
- Classes and Objects
- Encapsulation
- Inheritance
- Polymorphism
- Data Classes
- Enum
- Singelton
- Companion Object
- Interface
- Abstract Class
- Exceptions

INTRO TO OOP

Intro To OOP

- OOP stands for Object-Oriented Programming.
- Object-oriented programming has several advantages over procedural programming:
  - OOP is faster and easier to execute
  - OOP provides a clear structure for the programs
  - OOP helps to keep the Kotlin code **DRY "Don't Repeat Yourself",** and makes the code easier to maintain, modify and debug

What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Class

Fruit

Objects

Apple

Mango

Classes and Objects

To create a class, use the **class** keyword, and specify the name of the class:

Example : Create a Car class along with some properties

```
class Car {
  var brand = ""
  var model = ""
  var year = 0
}
```

A property is basically a **variable** that belongs to the class.

Constructors

A constructor is like a special function

- **Primary Constructor**

1. It is defined by using two parantheses () after the class name.
2. Class header can't contain any runnable code (If you want to run some code during object creation, use **initializer blocks** inside the class body) .
3. Declarations can also include default values of the class properties:

```
class Person(val firstName: String, val lastName: String, var age: Int)
```

Classes and Objects

Cont’d Primary Constructor

- If the constructor has annotations or visibility modifiers, the **constructor keyword** is required and the modifiers go before it:

```
class Person private constructor(val name: String, val age: Int)
```

Secondary constructors

- A secondary constructor is an additional constructor that you can define in a class to provide different ways of initializing it.

- While primary constructors are declared **as part of the class header,** secondary constructors are declared **inside the class body.**

Classes and Objects

Cont’d Secondary Constructor

- A class can also declare secondary constructors, which are prefixed with **constructor**

- each secondary constructor needs to **delegate** to the primary constructor either directly or indirectly

```
class Person (var age:Int=20,val name: String ){
    constructor(address:String ):this(name = " "){
        println("The Address =$address")
    }
```

Classes and Objects

Cont’d Secondary Constructor

Delegation to another constructor of the same class is done using the **this** keyword:

```
class Person(val name: String) {
    val children: MutableList<Person> = mutableListOf()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Intializer Block

- Initializer blocks are declared with the **init** keyword followed by curly braces
- Code in initializer blocks effectively becomes part of the **primary constructor**.
- Delegation to the primary constructor happens at the moment of access to the first statement of a secondary constructor

Classes and Objects

Intializer Block Cont’d

- the code in all initializer blocks and property initializers is executed **before the body of the secondary constructor.**
- Even if the class has no primary constructor, **the delegation still happens implicitly**, and the initializer blocks are still executed:

```
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor $i")
    }
}
```

Example with Primary and Secondary Constructors

```
class Person(val name: String, val age: Int) {

    // Secondary constructor delegating to the primary constructor
    constructor(name: String) : this(name, age: 0) {
        println("Secondary constructor called with name only")
    }
    // Another secondary constructor
    constructor() : this( name: "Unknown") {
        println("Secondary constructor called with no parameters")
    }
}
```

```
fun main() {
    // Using primary constructor
    val person1 = Person( name: "John", age: 25)
    println("${person1.name}, ${person1.age}")  // Output: John, 25
    // Using secondary constructor with one parameter
    val person2 = Person( name: "Jane")
    println("${person2.name}, ${person2.age}")  // Output: Jane, 0
    // Using secondary constructor with no parameters
    val person3 = Person()
    println("${person3.name}, ${person3.age}")  // Output: Unknown, 0
}
```

Example with Secondary Constructor (No Primary Constructor):

```
class Animal {
    var name: String
    var age: Int

    // Secondary constructor that initializes 'name' and 'age'
    constructor(name: String, age: Int) {
        this.name = name
        this.age = age
    }

    // Secondary constructor that initializes 'name' and defaults 'age' to 0
    constructor(name: String) : this(name, age: 0)
}
```

Attributes & Parameters

- In Kotlin, Attributes (also known as **properties**) and Parameters **(local variables)** have distinct roles within primary and secondary constructors.
    - In the primary constructor, parameters can be directly converted into attributes by using var or val
    - in secondary constructors, parameters are usually local to the constructor and often delegate to the primary constructor to initialize attributes.

Attributes & Parameters

Attributes

- Also called properties, they are the variables associated with an instance of a class.
- Declared with var or val, they define the state of an object and persist throughout the object’s lifetime.
- Accessible from anywhere in the class and beyond (depending on visibility modifiers).
- Can have default values and be initialized within constructors or initializer blocks.
- Represent the state of the object and persist throughout its lifecycle.

Attributes & Parameters

Parameters (Local Variables):

- These are local variables that exist only within the scope of the constructor (primary or secondary).
- They are temporary and cease to exist once the constructor finishes execution unless they are assigned to class attributes.
- Used for temporary data during the object creation process or to help initialize attributes.
- Declared in the constructor header but are not class properties unless explicitly assigned to them.

Attributes & Parameters

Attributes and Parameters in Primary Constructor

Attribute:

- **name: String** : is declared with val in the primary constructor, making it a property (attribute) of the class. This means name can be accessed throughout the class and outside of it as well (due to public visibility).

Parameter:

- **age: Int** : is declared without var or val in the primary constructor, making it a local parameter within the constructor. It only exists inside the constructor unless it is assigned to a class property, which is done in the class body (var age: Int = age).

```
class Person(val name: String, age: Int) {
    var age: Int = age

    init {
        println("Person's name is $name and age is $age")
    }
}
```

Attributes & Parameters

Usage in Secondary Constructors

- In a secondary constructor, parameters **are local to that constructor** and exist only within the constructor's scope.

- If you need to assign the values of those parameters to class attributes (properties), you must do so explicitly.

```
class Animal {
    var species: String
    var age: Int

    constructor(species: String, age: Int) {
        this.species = species
        this.age = age
    }
    constructor(species: String) : this(species, age: 0) {
        println("Secondary constructor called: $species, Age: $age")
    }
}
```

OOP PRINCIPLES

Encapsulaion

- Encapsulation is the concept of restricting direct access.
- Providing controlled access to it through methods (getters and setters).
- Define properties using val (read-only) or var (mutable),

private var name: String = "John"

GETTER

```
// Public getter for 'name'
fun getName(): String {
    return name
}
```

SETTER

```
// Public setter for 'name'
fun setName(newName: String) {
    if (newName.isNotEmpty()) {
        name = newName
    }
}
```

Encapsulaion

- **Example on Getter and Setter :**

```
class Rectangle(initialWidth: Int, initialHeight: Int) {
    var width: Int = initialWidth
        get() = field // Default getter
        set(value) { field = value } // Default setter

    var height: Int = initialHeight
        get() = field // Default getter
        set(value) { field = value } // Default setter

    val area: Int
        get() = width * height // Read-only property with a custom getter
}
```

Inheritance

- **All classes in Kotlin have a common superclass Any**
- **Any** has three methods :

1.equals()

2.hashCode()

3. toString()

these methods are defined for all Kotlin classes.

- By Default , Kotlin classes are **final** , they **can't be inherited**
- To make a class inheritable, mark it with the **open** keyword:

```
open class Base // Class is open for inheritance
```

Inheritance

- To declare an explicit supertype

  If the derived class has a **primary constructor**, the base class **can (and must)** be initialized in that primary constructor.

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

If the derived class has **no primary** constructor, then each **secondary constructor** has to initialize the base type using the **super** keyword.

```
open class Base(val value: String)

class Derived : Base {
    constructor(message: String) : super(message)
}
```

Polymorphism

- **Overriding Methods**

  The **override** modifier is required. If it's missing, the compiler will complain.

```
open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
}

class Circle() : Shape() {
    override fun draw() { /*...*/ }
}
```

- A member marked override is itself **open**, so it may be overridden in subclasses. If you want to prohibit re-overriding, use **final** Keyword

```
final override fun draw()
{}
```

Polymorphism

- **Overriding properties**

  A **declared property** in a base class can be overridden in a derived class using either:

1. A property with an initializer.

```
open class Base {
    open val message: String = "Base Message"
}

class Derived1 : Base() {
    override val message: String = "Derived1 Message"
}
```

2. A property with a custom getter.

```
open class Base {
    open val message: String = "Base Message"
}

class Derived2 : Base() {
    override val message: String
        get() = "Custom message from Derived2"
}
```

BONUS QUESTION ?

Can we override a val property with a var property ?. if yes why ?

Polymorphism

- **Overloading Methods**

  This occurs when multiple functions with **the same name** but **different parameters** exist in the same class. Can Differ In JUST :

  - **Number** of Parameters
  - **Type** of Parameters
  - **Order** of Parameters

```
class Calculator {
    fun add(a: Int, b: Int): Int {
        return a + b
    }

    fun add(a: Int, b: Int, c: Int): Int {
        return a + b + c
    }
}
```

KOTLIN CLASSES

Data Classes

- **Data classes** in Kotlin are primarily used to **hold** data
- For each data class, the compiler **automatically generates** additional member functions :
  - **equals()**
  - **toString()**
  - **copy()**
- Data Classes have to fulfill the following requirements :
  - The primary constructor must have at least one parameter.
  - All primary constructor parameters must be marked as val or var.
  - Data classes can't be abstract, open, sealed, or inner.

Enum Class

- An enum class in Kotlin is a special data type **used to define a set of constants.** These constants are implicitly **instances** of the enum class.

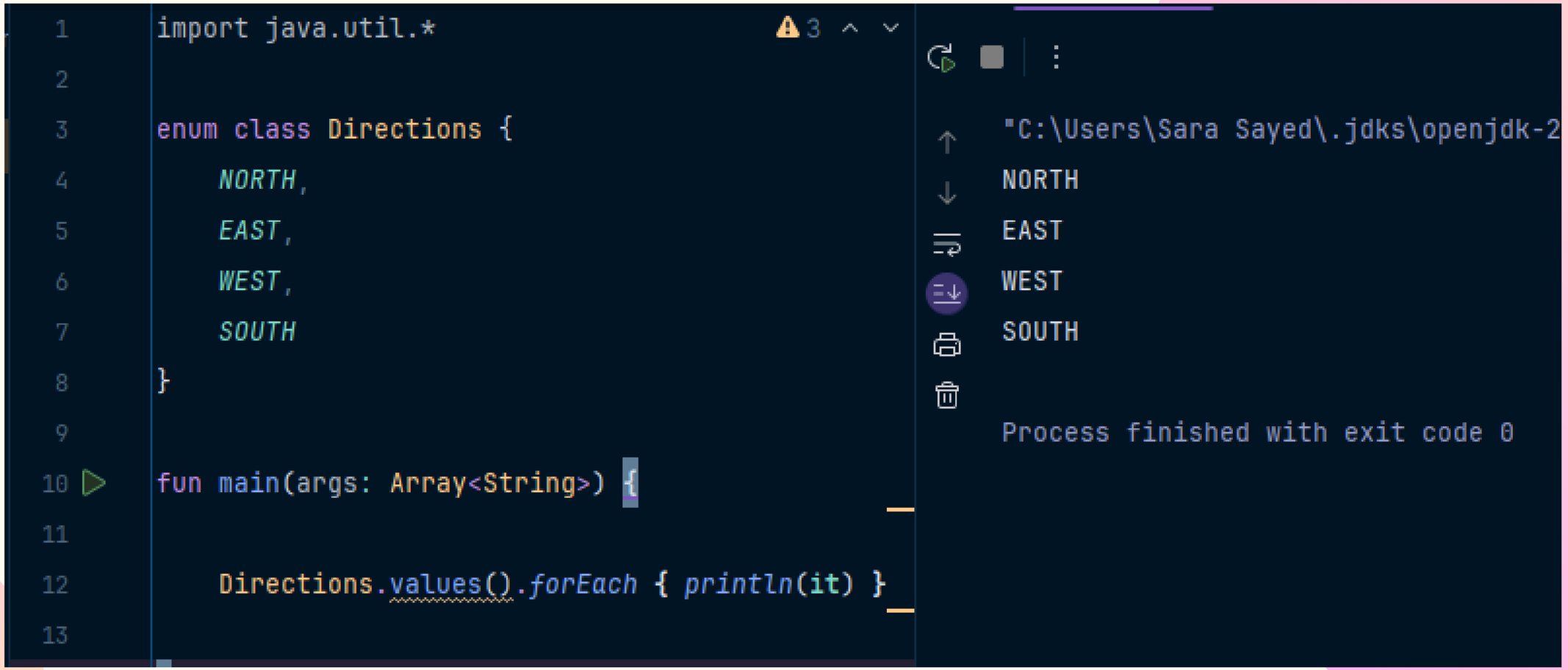
- Examples: Days, colors, directions and etc...

```
enum class Directions {
    NORTH,
    EAST,
    WEST,
    SOUTH
}
```

Enum Class

- There are some Methods for Enum class

Print values of class



Enum Class

- **Give value to each constant**
  - must put value in constructor
  - put (;) in the end of class

```
enum class Directions (val value :Int){
    NORTH( value: 15),
    EAST( value: 20),
    WEST( value: 45),
    SOUTH( value: 10);
}
```

Enum Class

It is the same if you want to make it Functions not variables

- Here we must take care about point that we must make a “calculate” function for using the Enum functions.
- Keyword “this” in when statement refer to the Enum Operation you will choose.

```
3   enum class Operation {
4       ADD,
5       MULTIPLY;
6
7       fun calculate(a: Int, b: Int): Int {
8           return when (this) {
9               ADD → a + b
10              MULTIPLY → a * b
11          }
12      }
13  }
14
15  fun main(args: Array<String>) {
16      println(Operation.ADD.calculate( a: 10,  b: 4))
17      println(Operation.MULTIPLY.calculate( a: 10,  b: 4))
18  }
19
```

```
"C:\Users\Sara Sayed\.jd
14
40

Process finished with ex
```

Enum Class

- Use constant with logic functions

```
3   enum class Day(val isWeekend: Boolean=false) {
4       SATURDAY( isWeekend: true),
5       SUNDAY,
6       MONDAY,
7       TUESDAY,
8       WEDNESDAY,
9       THURSDAY,
10      FRIDAY( isWeekend: true);
11      fun checkWeekend()= (this == FRIDAY || this == SATU
12  }
13
14  fun main(args: Array<String>) {
15      println(Day.MONDAY.checkWeekend())
16      println(Day.FRIDAY.checkWeekend())
17  }
```

```
"C:\Users\Sara Saye
false
true

Process finished wi
```

Singleton

- The Singleton pattern ensures that a class has only one instance
- provides a global point of access to That Instance.
- In Kotlin, this is achieved effortlessly using the *object* keyword.
  - **Example** : when you use database or api which must be on object across whole project.

Key Features

- **Single Instance:** The Singleton object is created at runtime and is shared throughout the application.
- **Thread-Safe:** Kotlin’s object is thread-safe by design, eliminating the need for additional synchronization.
- **Global Access:** The Singleton instance is accessible globally without requiring instantiation.

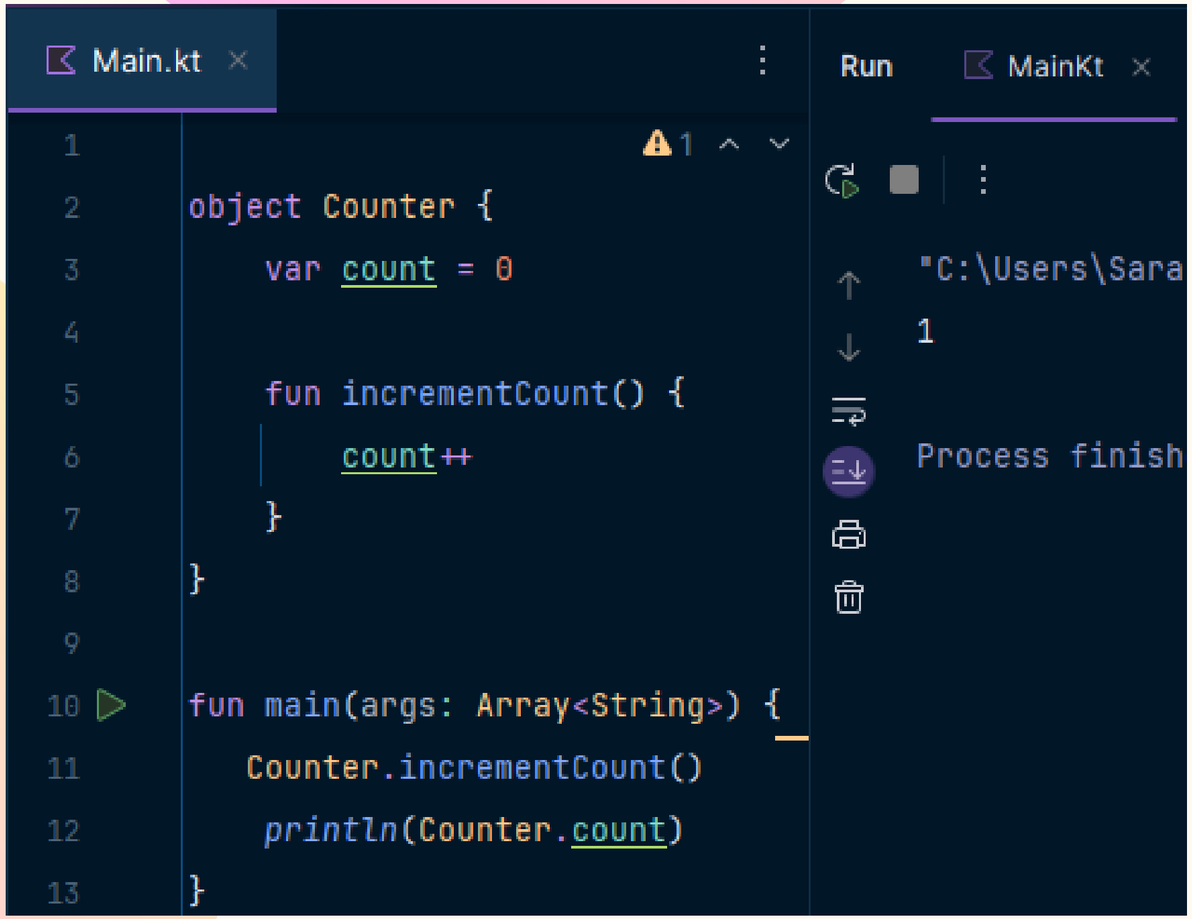
Singleton

- In traditional implementations **(e.g., in Java)**, this is done by making the class **constructor private** and providing a global access point using a static instance.
- Kotlin simplifies this process with the **object** keyword. When you declare a class as an object, Kotlin :
  - Automatically ensures a single instance of that class is created.
  - Provides thread safety out of the box.
  - Makes the Singleton globally accessible without the need for explicit initialization or synchronization.

This means you can use the object keyword to define a Singleton and seamlessly access the same instance across your entire project.

Singleton

How to write it ?



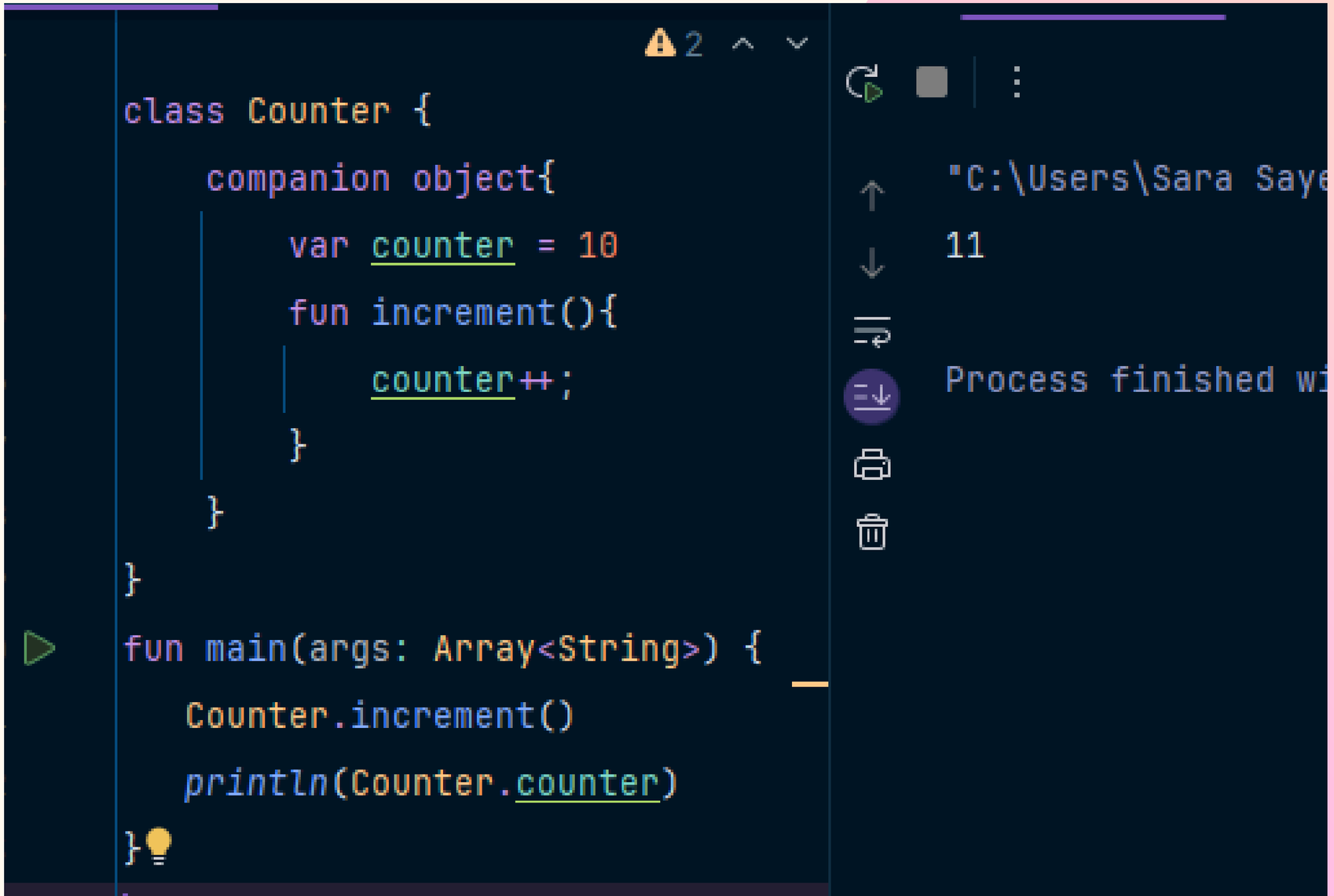
Companion Object

In Kotlin, a Companion Object is a special object that is **associated with a class** and allows you **to define properties and functions that behave like static members** in Java or other languages.

However, unlike static members, companion objects are **more powerful as they allow you to implement interfaces or inherit from other classes.**

Companion Object

How to write it?



INTERFACES AND ABSTRACT CLASSES

Interface

- An interface in Kotlin is a blueprint for a class that defines a set of properties and methods without implementing them.

- A class that implements an interface must provide the implementation for its members, unless the interface itself provides default implementations.

- **Use Case:**
  - Interfaces are used to achieve abstraction and multiple inheritance in Kotlin, allowing classes to share behavior across unrelated hierarchies.

Interface

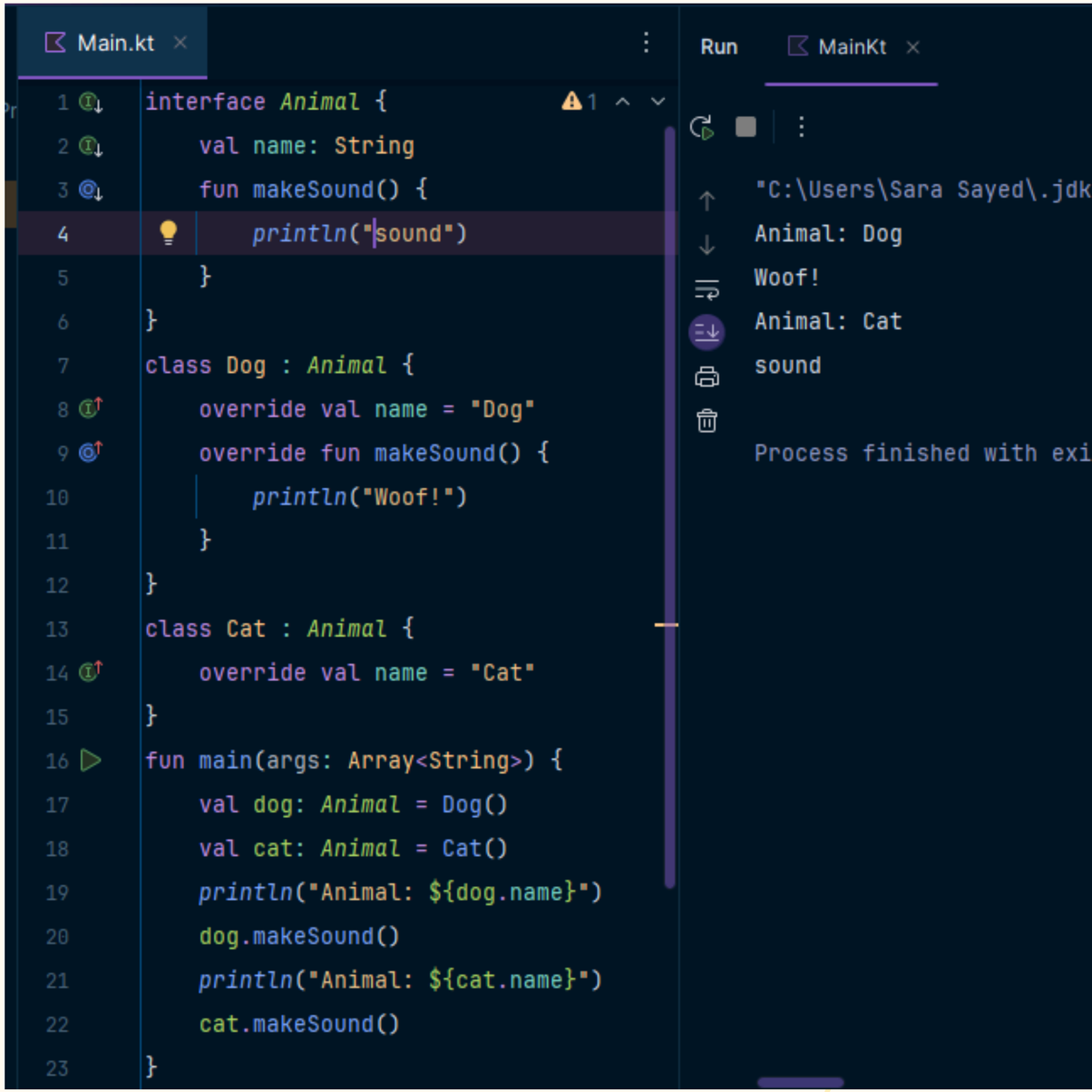
How to use ?

```kotlin
interface Animal {
    val name: String
    fun makeSound()
}

class Dog : Animal {
    override val name = "Dog"
    override fun makeSound() {
        println("Woof!")
    }
}

fun main(args: Array<String>) {
    val dog: Animal = Dog()
    println("Animal: ${dog.name}")
    dog.makeSound()
}
```

```
"C:\Users\Sara Saye
Animal: Dog
Woof!

Process finished wi
```

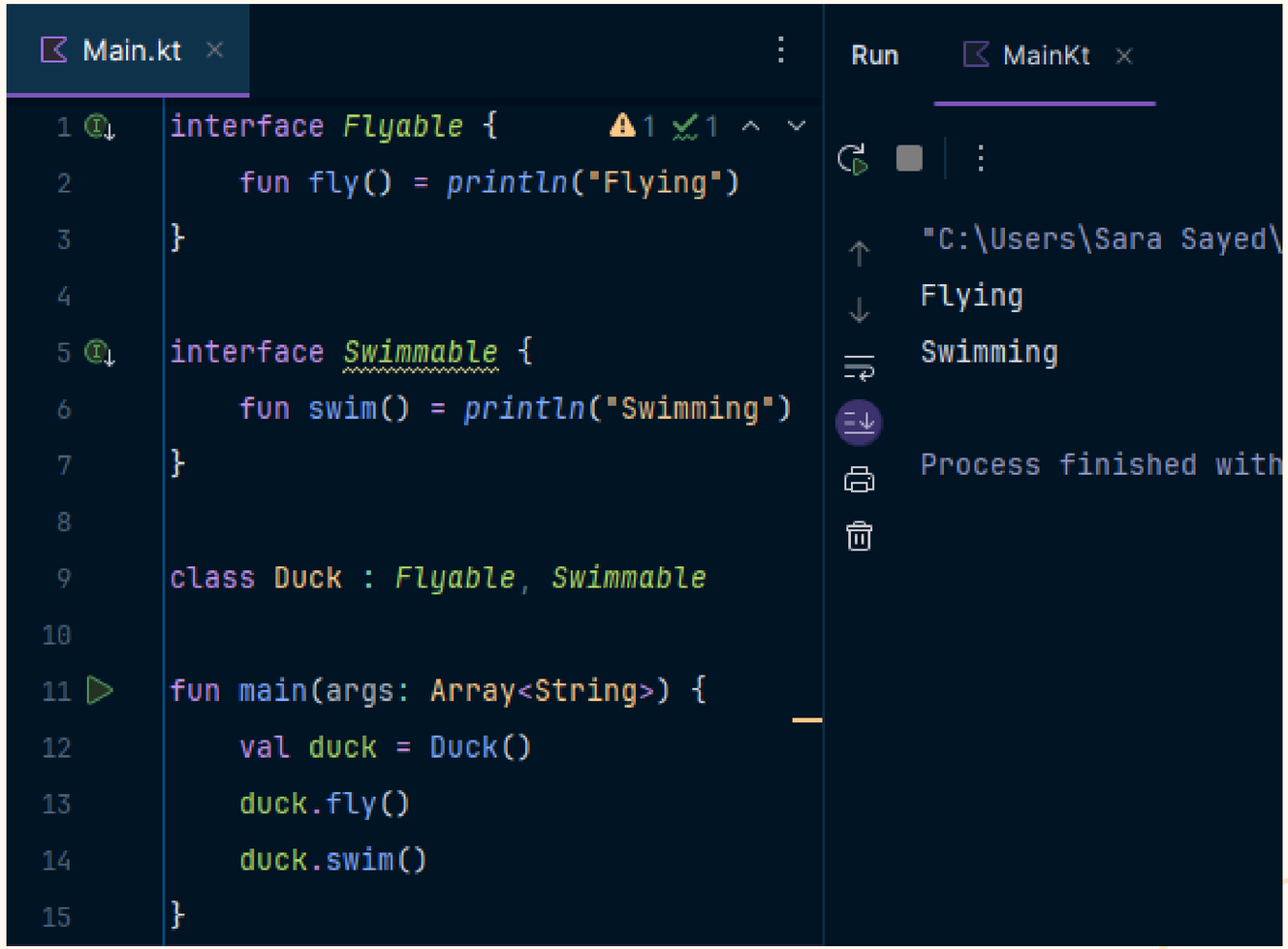
Interface

- Default Implementations:
  Kotlin allows interfaces to provide default implementations for methods.
- when i give “makeSound()”
  a simple Implementation i can use it in class “Cat” without override it



Interface

- Multiple Inheritance: A class can implement multiple interfaces, inheriting behavior from all.



Abstract Class

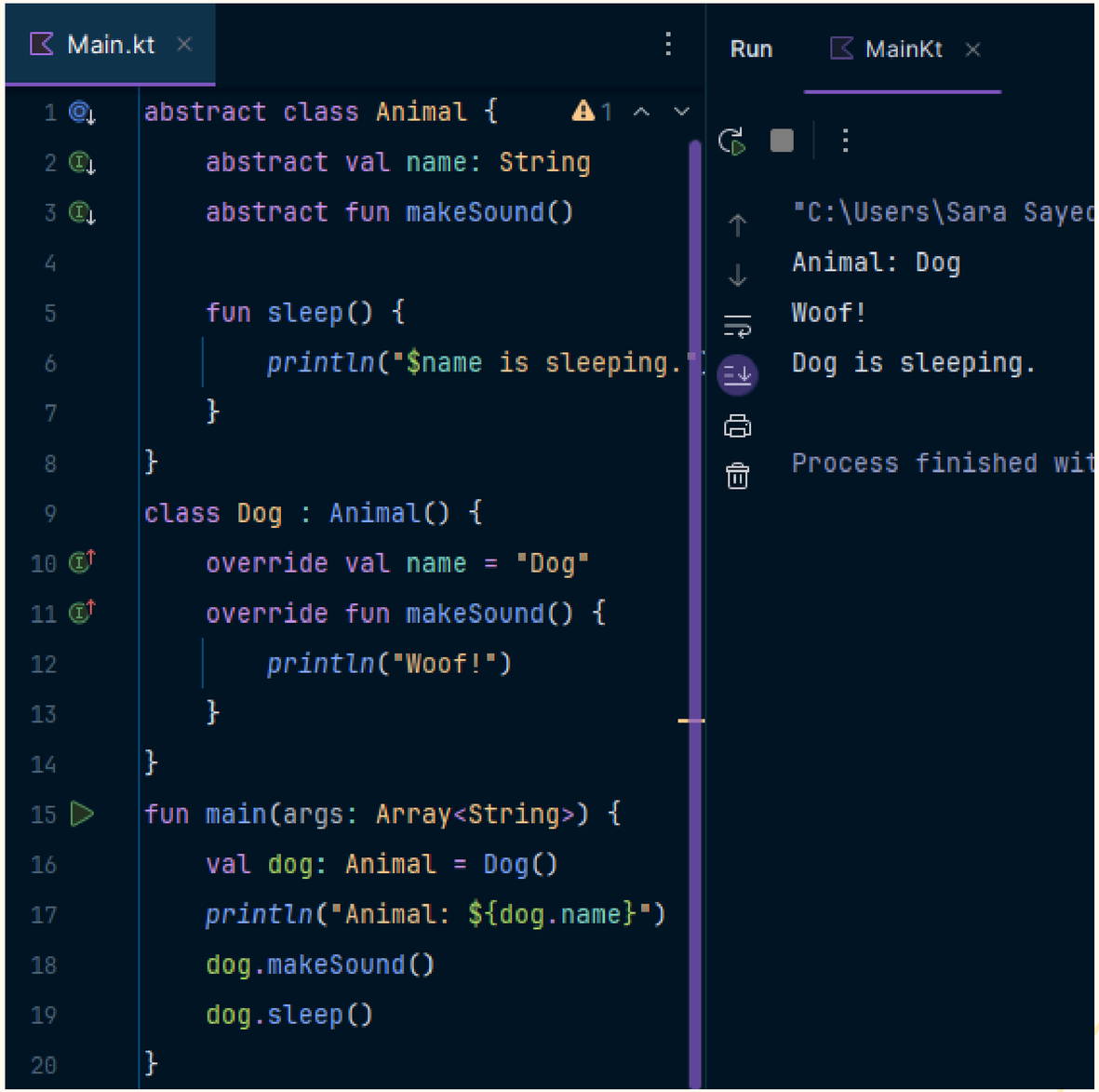
- An abstract class in Kotlin is a class that **cannot be instantiated** and is designed to be inherited by other classes.
- It can contain both abstract members (methods and properties without implementation) and concrete members (methods and properties with implementation).

Use Case:

- Abstract classes are ideal for scenarios where you want to provide a common base class with shared functionality, while leaving some parts of the implementation to be defined by subclasses.

Abstract Class

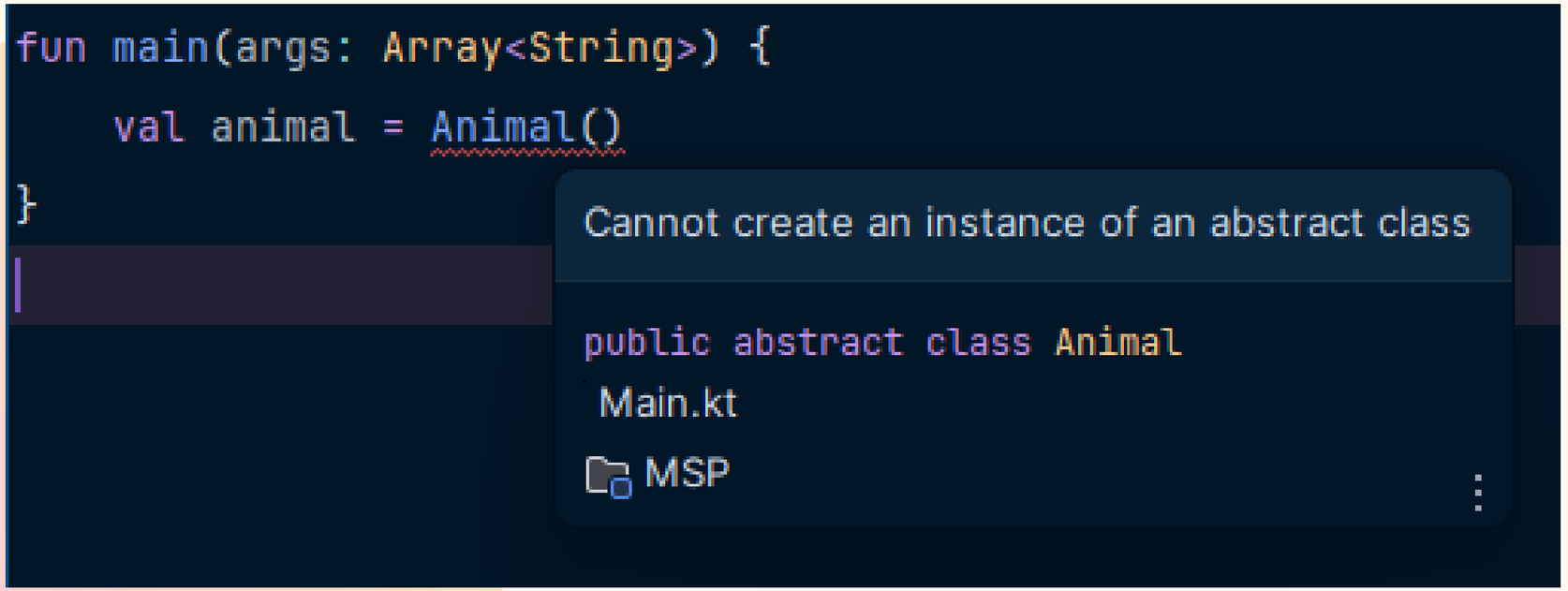
- To be noted here we use “sleep()” function without implemented it in “Dog” class because it already implemented in “Animal” abstract class



Abstract Class

Special Features of Abstract Classes

- Cannot Be Instantiated: You cannot create an instance of an abstract class directly.



Abstract Class

- Can Contain Concrete Methods and Properties: Unlike interfaces, abstract classes can have fully implemented methods and properties.

That is mean it has abstracted methods which must be override and normal functions

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

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

Abstract Class

- Constructor Support: Abstract classes can have constructors, which can be used to initialize shared properties.

Note : In kotlin Interface not supported constructors

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

    fun sleep() {
        println("is sleeping.")
    }
}

class Dog(name: String) : Animal(name) {
    override fun makeSound() {
        println("Woof!")
    }
}
```

Abstract Class VS Interfaces

Feature	Abstract class	Interface
Instantiation	Cannot be instantiated	Cannot be instantiated
Concrete Members	Allowed	Allowed (methods only)
Multiple Inheritance	Not allowed	allowed
Constructors	Allowed	Not allowed

EXCEPTIONS

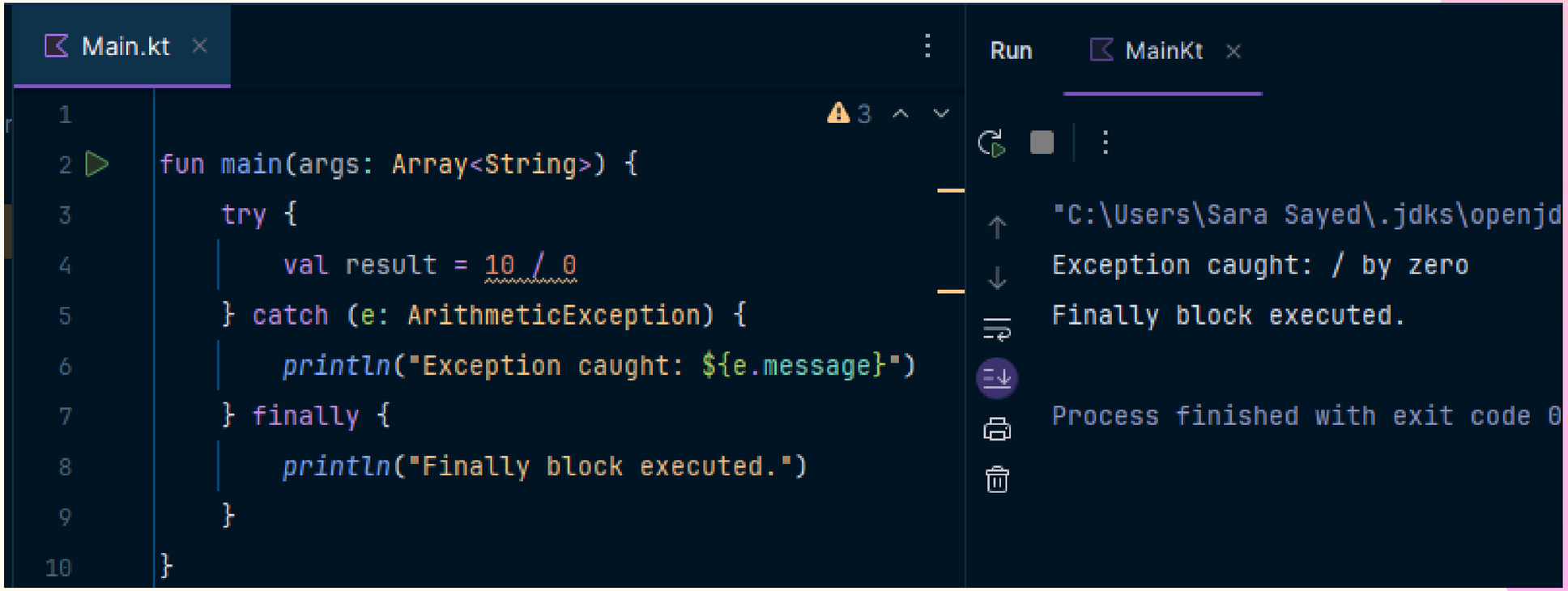
Exceptions

- An exception is an event that **disrupts the normal flow** of a program.
- It typically occurs due to unexpected situations such as invalid input, a file not found, or division by zero.
- In Kotlin, exceptions are used to handle such errors gracefully without crashing the application.
- Exceptions provide a way to manage errors and take corrective actions, ensuring the program can recover or terminate safely.

Exceptions

Exception Handling

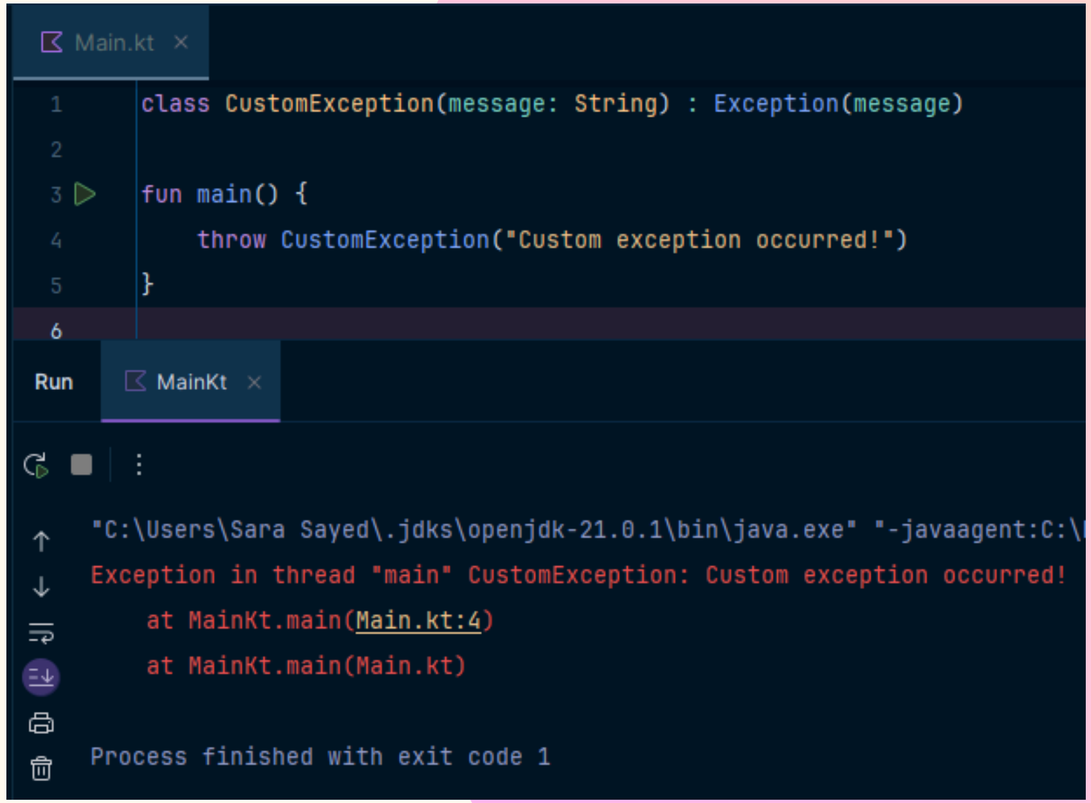
- Kotlin uses try, catch, and finally blocks to handle exceptions



Exceptions

- **Unchecked Exceptions:** Unlike Java, Kotlin does not differentiate between checked and unchecked exceptions. This means you are not forced to declare exceptions in function signatures or handle them explicitly

- **Custom Exceptions:** You can create your own exceptions by extending the Throwable class or its subclasses like Exception or RuntimeException.



Thank You