KOTLIN CONCEPTS

TODAY'S AGENDA

- Kotlin Concepts – Classes & Objects –
  - Classes, Inheritance,
  - Interfaces, Functional (SAM) interfaces, Visibility modifiers, Extensions, Data classes,
  - Sealed classes, Enum classes, Inline classes
  - Nested and inner classes

CLASSES AND OBJECTS

- Classes in Kotlin are declared using the keyword class:
  - class Person { /*...*/ }
- The class declaration consists of the class name, the class header (specifying its type parameters, the primary constructor, and some other things), and the class body surrounded by curly braces. Both the header and the body are optional; if the class has no body, the curly braces can be omitted.
  - class Empty

Constructors

- A class in Kotlin can have a *primary constructor* and one or more *secondary constructors*. The primary constructor is a part of the class header, and it goes after the class name and optional type parameters.
  - class Person constructor(firstName: String) { /*...*/ }
  - class Person(firstName: String) { /*...*/ }
- The primary constructor cannot contain any code. Initialization code can be placed in *initializer blocks* prefixed with the init keyword.
- During the initialization of an instance, the initializer blocks are executed

```
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println)

    init {
        println("First initializer block that prints $name")
    }

    val secondProperty = "Second property: ${name.length}".also(::println)

    init {
        println("Second initializer block that prints ${name.length}")
    }
}
```

- Kotlin has a concise syntax for declaring properties and initializing them from the primary constructor:
  - class Person(val firstName: String, val lastName: String, var age: Int)
- Such declarations can also include default values of the class properties:
  - class Person(val firstName: String, val lastName: String, var isEmployed: Boolean = true)
- You can use a trailing comma when you declare class properties:
  - class Person(
    - val firstName: String,
    - val lastName: String,
    - var age: Int, // trailing comma
    -) { /*...*/ }

- Much like regular properties, properties declared in the primary constructor can be mutable (var) or read-only (val).
- If the constructor has annotations or visibility modifiers, the constructor keyword is required and the modifiers go before it:
  - class Customer public @Inject constructor(name: String) { /*...*/ }

Secondary constructors

- A class can also declare *secondary constructors*, which are prefixed with constructor:
  - class Person(val pets: MutableList<Pet> = mutableListOf())
  - class Pet {
    - constructor(owner: Person) {
    - owner.pets.add(this) // adds this pet to the list of its owner's pets }

    }

Creating instances of classes

- To create an instance of a class, call the constructor as if it were a regular function:
  - val invoice = Invoice()
  - val customer = Customer("Joe Smith")
- Kotlin does not have a new keyword.

Class members

- Classes can contain:
  - Constructors and initializer blocks
  - Functions
  - Properties
  - Nested and inner classes
  - Object declarations

Abstract classes

- A class may be declared abstract, along with some or all of its members. An abstract member does not have an implementation in its class. You don't need to annotate abstract classes or functions with open.
  - abstract class Polygon {
  - abstract fun draw()
  - }
  - class Rectangle : Polygon() {
  - override fun draw() {
  - // draw the rectangle
  - } }

Inheritance

- All classes in Kotlin have a common superclass, Any, which is the default superclass for a class with no supertypes declared:
  - class Example // Implicitly inherits from Any
- Any has three methods: equals(), hashCode(), and toString(). Thus, 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
- To declare an explicit supertype, place the type after a colon in the class header:
  - open class Base(p: Int)
  - class Derived(p: Int) : Base(p)

Overriding methods

- Kotlin requires explicit modifiers for overridable members and overrides:
  - open class Shape {
    - open fun draw() {
    - /*...*/
    - } fun fill() {
    - /*...*/ }
    - }
  - class Circle() : Shape() {
    - override fun draw() {
    - /*...*/ }
    - }

- The override modifier is required for Circle.draw(). If it were missing, the compiler would complain. If there is no open modifier on a function, like Shape.fill(), declaring a method with the same signature in a subclass is not allowed, either with override or without it.
- The open modifier has no effect when added to members of a final class – a class without an open modifier.

- A member marked override is itself open, so it may be overridden in subclasses. If you want to prohibit re-overriding, use final:
  - open class Rectangle() : Shape() {
  - final override fun draw() {
  - /*...*/
  - } }
-

Overriding properties

- The overriding mechanism works on properties in the same way that it does on methods. Properties declared on a superclass that are then redeclared on a derived class must be prefaced with override, and they must have a compatible type. Each declared property can be overridden by a property with an initializer or by a property with a get method:
  - open class Shape {
  - open val vertexCount: Int = 0
  - }
  - class Rectangle : Shape() {
    - override val vertexCount = 4
    - }

- Note that you can use the override keyword as part of the property declaration in a primary constructor:
  - interface Shape {
    - val vertexCount: Int }
  - class Rectangle(override val vertexCount: Int = 4) : Shape
    // Always has 4 vertices
  - class Polygon : Shape {
    - override var vertexCount: Int = 0 // Can be set to any number later
    }

Visibility modifiers

- Classes, objects, interfaces, constructors, and functions, as well as properties and their setters, can have *visibility modifiers*. Getters always have the same visibility as their properties.
- There are four visibility modifiers in Kotlin: private, protected, internal, and public. The default visibility is public.

Packages

- Functions, properties, classes, objects, and interfaces can be declared at the "top-level" directly inside a package:
  - // file name: example.kt
  - package foo fun baz() { ... }
  - class Bar { ... }

- If you don't use a visibility modifier, public is used by default, which means that your declarations will be visible everywhere.
- If you mark a declaration as private, it will only be visible inside the file that contains the declaration.
- If you mark it as internal, it will be visible everywhere in the same module.
- The protected modifier is not available for top-level declarations.(class + sub class)

Class members

- For members declared inside a class:
  - private means that the member is visible inside this class only (including all its members).
  - protected means that the member has the same visibility as one marked as private, but that it is also visible in subclasses.
  - internal means that any client *inside this module* who sees the declaring class sees its internal members.
  - public means that any client who sees the declaring class sees its public members.

Constructors

- Use the following syntax to specify the visibility of the primary constructor of a class:
- You need to add an explicit constructor keyword.
  - class C private constructor(a: Int) { ... }
- Here the constructor is private. By default, all constructors are public, which effectively amounts to them being visible everywhere the class is visible (this means that a constructor of an internal class is only visible within the same module).

- **Local declarations**
  - Local variables, functions, and classes can't have visibility modifiers.
- **Modules**
  - The internal visibility modifier means that the member is visible within the same module. More specifically, a module is a set of Kotlin files compiled together, for example:
    - An IntelliJ IDEA module.
    - A Maven project.
    - A Gradle source set (with the exception that the test source set can access the internal declarations of main).
    - A set of files compiled with one invocation of the <kotlinc> Ant task.

Abstract Class

- An abstract class cannot be instantiated, which means we cannot create the object of an abstract class. Unlike other class, an abstract class is always open so we do not need to use the open keyword.
- **Points to Note:**
  1. We cannot create the object of an abstract class.
  2. Property and member function of an abstract class are by default **non-abstract**. If you want to override these in the child class then you need to use open keyword for them.
  3. If a member function is abstract then it must be implemented in the child class. An abstract member function doesn't have a body only method signature, the implementation is done in the child class.

Interfaces

- Interfaces in Kotlin can contain declarations of abstract methods, as well as method implementations. What makes them different from abstract classes is that interfaces cannot store state. They can have properties, but these need to be abstract or provide accessor implementations.
- An interface is defined using the keyword interface:
  - interface MyInterface {
    - fun bar()
    - fun foo() {
    - // optional body} }

Implementing interfaces

- A class or object can implement one or more interfaces:
  - class Child : MyInterface {
  - override fun bar() {
    - // body } }

Properties in interfaces

- You can declare properties in interfaces. A property declared in an interface can either be abstract or provide implementations for accessors. Properties declared in interfaces can't have backing fields, and therefore accessors declared in interfaces can't reference them:

```
interface MyInterface {
    val prop: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

Interfaces Inheritance

- An interface can derive from other interfaces, meaning it can both provide implementations for their members and declare new functions and properties. Quite naturally, classes implementing such an interface are only required to define the missing implementations:

```kotlin
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}

data class Employee(
    // implementing 'name' is not required
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```

Functional (SAM) interfaces

- An interface with only one abstract method is called a *functional interface*, or a *Single Abstract Method (SAM) interface*. The functional interface can have several non-abstract members but only one abstract member.
- To declare a functional interface in Kotlin, use the fun modifier.
  - fun interface KRunnable {
    - fun invoke()
    - }

SAM conversions

- For functional interfaces, you can use SAM conversions that help make your code more concise and readable by using lambda expressions.
- Instead of creating a class that implements a functional interface manually, you can use a lambda expression. With a SAM conversion, Kotlin can convert any lambda expression whose signature matches the signature of the interface's single method into the code, which dynamically instantiates the interface implementation.

Data classes

- It is not unusual to create classes whose main purpose is to hold data. In such classes, some standard functionality and some utility functions are often mechanically derivable from the data. In Kotlin, these are called *data classes* and are marked with data:
  - data class User(val name: String, val age: Int

- The compiler automatically derives the following members from all properties declared in the primary constructor:
  - equals()/hashCode() pair
  - toString() of the form "User(name=John, age=42)"
  - componentN() functions corresponding to the properties in their order of declaration.
  - copy() function (see below).

- To ensure consistency and meaningful behavior of the generated code, data classes have to fulfill the following requirements:
  - The primary constructor needs to have at least one parameter.
  - All primary constructor parameters need to be marked as val or var.
  - Data classes cannot be abstract, open, sealed, or inner.

Properties declared in the class body

- The compiler only uses the properties defined inside the primary constructor for the automatically generated functions. To exclude a property from the generated implementations, declare it inside the class body:
  - data class Person(val name: String) {
  - var age: Int = 0 }

Copying

- Use the copy() function to copy an object, allowing you to alter *some* of its properties while keeping the rest unchanged. The implementation of this function for the User class above would be as follows:
  - fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
- You can then write the following:
  - val jack = User(name = "Jack", age = 1) val olderJack = jack.copy(age = 2)

Data classes and destructuring declarations

- *Component functions* generated for data classes make it possible to use them in destructuring declarations:
  - val jane = User("Jane", 35)
  - val (name, age) = jane
  - println("$name, $age years of age") // prints "Jane, //35 years of age"

Kotlin Nested Class

- When a class is declared inside another class then it is called nested class. Nested classes are similar to static nested class in Java.
- By default nested class is **static** so we can access the nested class property or variables using dot(.) notation without creating an object of the class.
  class OuterClass { ....
  - class NestedClass { ...
  - }
  - }
- **Points to Note:**
  1. A Nested class cannot access the members of the outer class.
  2. To access the members of nested class, we use the dot (.) operator with outer class

Kotlin Inner class

- Kotlin inner class is declared using inner modifier. Inner classes have access to the members of the outer class.

SEALED CLASS

- A **sealed class** is used for representing restricted class hierarchy where an object or a value can have one of the types from a limited set of values. You can think of a sealed class as an extension of enum class. The set of values in enum class is also restricted, however an enum constant can have only single instance while a subclass of a sealed class can have multiple instances.

Advantages of Sealed Classes

- **Multiple Instances**
- While an enum constant exists only as a single instance, a subclass of a sealed class can have multiple instances. That allows objects from sealed classes to contain state.
- Look at the following example:
- sealed class Months { class January(var shortHand: String) : Months()
  - class February(var number: Int) : Months()
  - class March(var shortHand: String, var number: Int) : Months()
  - }

- Now you can create two different instances of February. For example, you can pass the 2019 as an argument to first instance, and 2020 to second instance, and compare them.

- **Inheritance**
- You can't inherit from enum classes, but you can from sealed classes.
- Here's an example:
- sealed class Result {
- data class Success(val data : List<String>) : Result()
data class Failure(val exception : String) : Result()
- }
- Both Success and Failure inherit from the Result sealed class in the code above.
- Kotlin 1.1 added the possibility for data classes to extend other classes, including sealed classes.

- **Architecture Compatibility**
- Sealed classes are compatible with commonly-used app architectures, including:
- MVI
- MVVM
- Redux
- Repository pattern
- This ensures that you don't have to change your existing app architecture to leverage the advantages of sealed classes.

- **When" Expressions**
- Kotlin lets you use when expressions with your sealed classes. When you use these with the Result sealed class, you can parse a result based on whether it was a success or failure.
- Here's how this might look:
- when (result) { is Result.Success -> showItems(result.data) is Result.Failure -> showError(result.exception) }

ENUM CLASS

- The most basic use case for enum classes is the implementation of type-safe enums:
  - enum class Direction { NORTH, SOUTH, WEST, EAST }
- Each enum constant is an object. Enum constants are separated by commas.
- Since each enum is an instance of the enum class, it can be initialized as:
  - enum class Color(val rgb: Int) { RED(0xFF0000), GREEN(0x00FF00), BLUE(0x0000FF) }

Anonymous classes

- Enum constants can declare their own anonymous classes with their corresponding methods, as well as with overriding base methods.
- enum class ProtocolState {
  - WAITING { override fun signal() = TALKING },
  - TALKING { override fun signal() = WAITING };
  - abstract fun signal(): ProtocolState }
- If the enum class defines any members, separate the constant definitions from the member definitions with a semicolon.

What is an inline class?

- An inline class is a special type of class defined in Kotlin that only has one property. At runtime, the compiler will “inline” the property where it was used. This is similar to inline functions, where the compiler inserts the contents of the function into where it was called.