Kotlin – Class and Objects

Tushar B. Kute,
http://tusharkute.com

What is Object Oriented Programming?

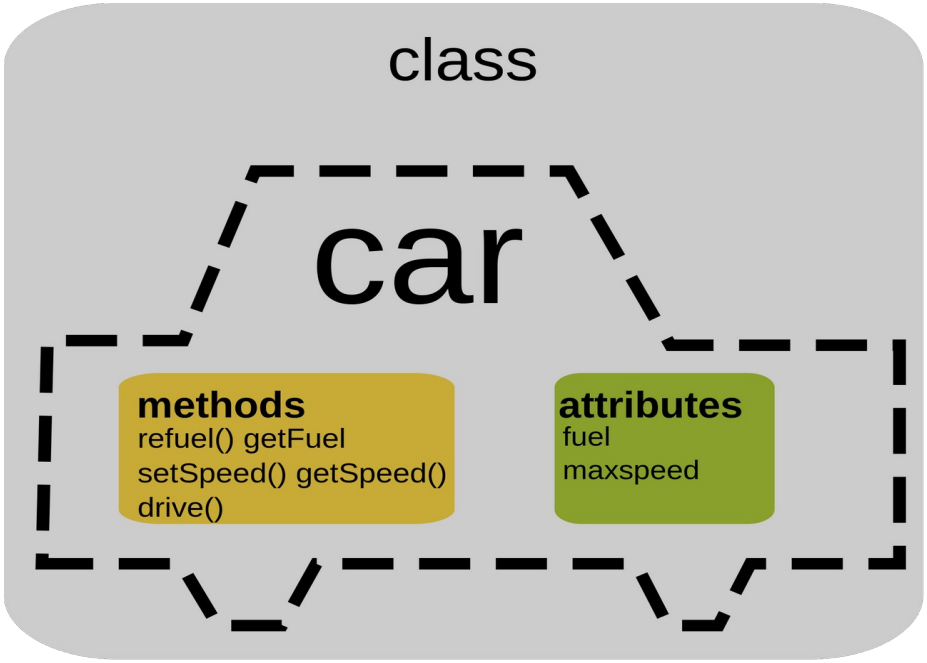
- Object-oriented Programming, or OOP for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.
- For instance, an object could represent a person with a name property, age, address, etc., with behaviors like walking, talking, breathing, and running.
- Or an email with properties like recipient list, subject, body, etc., and behaviors like adding attachments and sending.

What is Object Oriented Programming?

- Put another way, object-oriented programming is an approach for modeling concrete, real-world things like cars as well as relations between things like companies and employees, students and teachers, etc.
- OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

Class

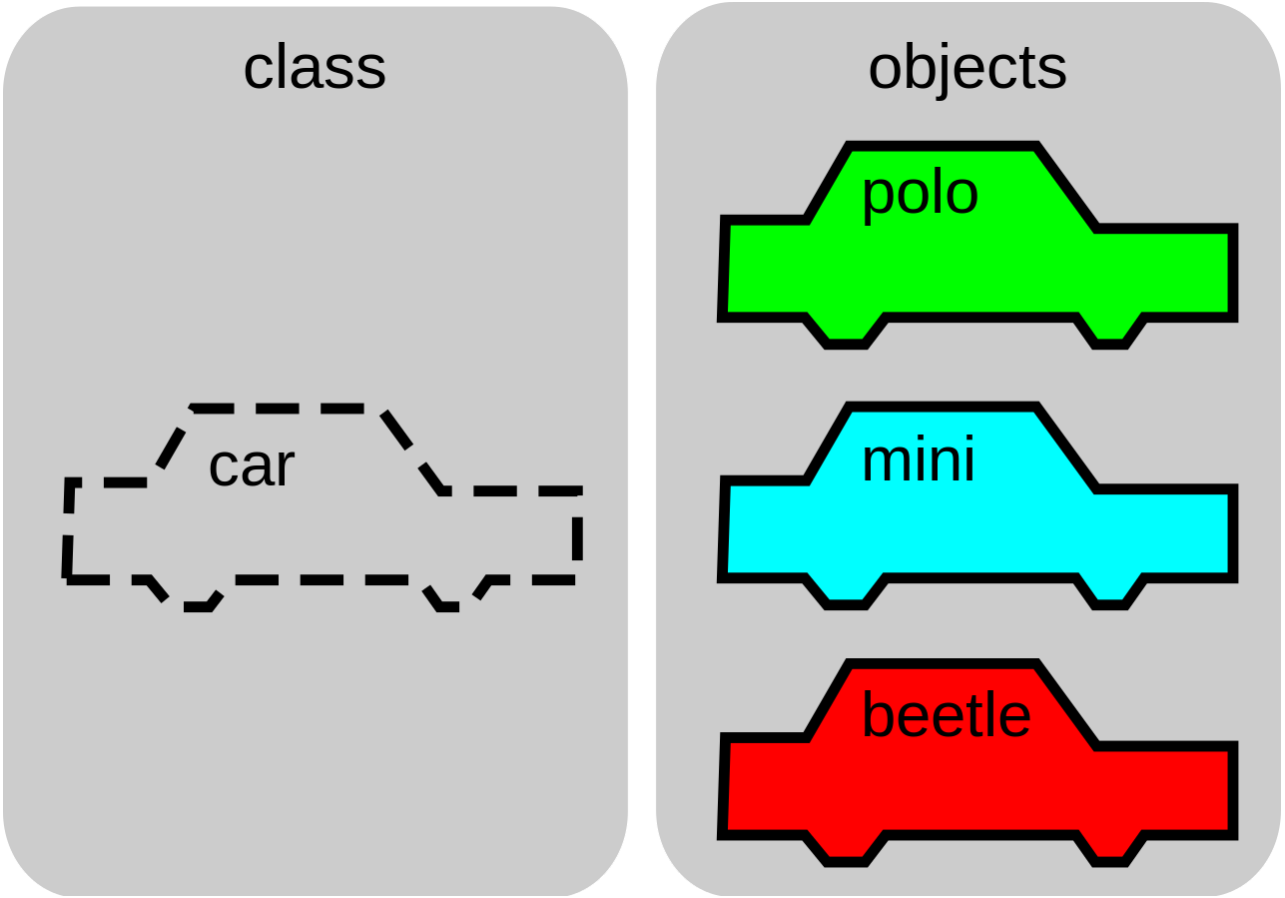
Classes are used to create new user-defined data structures that contain arbitrary information about something. In the case of an car, we could create an Car() class to track properties about the Car like the fuel and maxspeed.



Objects

- While the class is the blueprint, an instance is a copy of the class with actual values, literally an object belonging to a specific class.
- Put another way, a class is like a form or questionnaire.
- It defines the needed information. After you fill out the form, your specific copy is an instance of the class; it contains actual information relevant to you.

Objects



OOP Features

- Reduction in the complexity
- Importance of data
- Creation of new data structure
- Data Hiding
- Characterization of the objects
- Communication among objects
- Extensibility
- Bottom-up programming approach

OOP Concepts

- Class
- Object
- Data Hiding / abstraction / encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding

Kotlin Class

```
class ClassName { // Class Header
    //
    // Variables or data members
    // Member functions or Methods
    //
    ...
    ...
}
```

Creating objects

The objects are created from the Kotlin class and they share the common properties and behaviours defined by a class in form of data members (properties) and member functions (behaviours) respectively.

The syntax to declare an object of a class is:

```
var varName = ClassName()
```

Accessing Variables

We can access the properties and methods of a class using the . (dot) operator as shown below:

```
var varName = ClassName()

varName.property = <Value>
varName.functionName()
```

Class

- Examples

Constructor

- A Kotlin constructor is a special member function in a class that is invoked when an object is instantiated. Whenever an object is created, the defined constructor is called automatically which is used to initialize the properties of the class.
- Every Kotlin class needs to have a constructor and if we do not define it, then the compiler generates a default constructor.
- A Kotlin class can have following two type of constructors:
  - Primary Constructor
  - Second Constructors

Primary Constructor

- The primary constructor can be declared at class header level as shown in the following example.

```
class Person constructor(val firstName: String, val age: Int) {
  // class body
}
```

- The constructor keyword can be omitted if there is no annotations or access modifiers specified like public, private or protected..

```
class Person (val firstName: String, val age: Int) {
  // class body
}
```

Initializer Block

- The primary constructor cannot contain any code. Initialization code can be placed in initializer blocks prefixed with the init keyword.
- There could be more than one init blocks and during the initialization of an instance, the initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers

Inheritance

- Inheritance can be defined as the process where one class acquires the members (methods and properties) of another class. With the use of inheritance the information is made manageable in a hierarchical order.
-
- A class which inherits the members of other class is known as subclass (derived class or child class) and the class whose members are being inherited is known as superclass (base class or parent class).
-
- Inheritance is one of the key features of object-oriented programming which allows user to create a new class from an existing class. Inheritance we can inherit all the features from the base class and can have additional features of its own as well.

Inheritance : Example

```
open class ABC {
    fun think () {
        println("Hello World")
    }
}
class BCD: ABC(){
}

fun main() {
    var  a = BCD()
    a.think()
}
```

Overriding

- Overriding in Kotlin is a feature of object-oriented programming that allows a subclass (derived class) to provide a specific implementation for a method or property that is already defined in its superclass (base class).
- This enables polymorphism, where objects can be treated as instances of their superclass while still executing subclass-specific behavior.

Overriding

```
open class Animal {
    open fun sound() {
        println("Animal makes a sound")
    }
}
class Dog : Animal() {
    override fun sound() {
        super.sound() // Call the superclass's sound method
        println("Dog barks")
    }
}
fun main() {
    val myDog: Animal = Dog()
    myDog.sound() // Output: Animal makes a sound \n Dog barks
}
```

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 the keyword override, and they must have a compatible type.
- Example.

Derived Class Initialization Order

- When we create an object of a derived class then constructor initialization starts from the base class.
- Which means first of all base class properties will be initialized, after that any derived class instructor will be called and same applies to any further derived classes.
- This means that when the base class constructor is executed, the properties declared or overridden in the derived class have not yet been initialized.
- Example

Access Super Class Members

- Code in a derived class can call its superclass functions and properties directly using the super keyword.
- Example.

Abstract Class

- A Kotlin abstract class is similar to Java abstract class which can not be instantiated. This means we cannot create objects of an abstract class. However, we can inherit subclasses from a Kotlin abstract class.
- A Kotlin abstract class is declared using the abstract keyword in front of class name.
- The properties and methods of an abstract class are non-abstract unless we explictly use abstract keyword to make them abstract.
- If we want to override these members in the child class then we just need to use override keyword infront of them in the child class.

Interface

- The interface works exactly similar to Java 8, which means they can contain method implementation as well as abstract methods declaration.
- An interface can be implemented by a class in order to use its defined functionality.
- The keyword interface is used to define an interface in Kotlin as shown in the following piece of code.

Interface

```
interface ExampleInterface {
    var myVar: String      // abstract property
    fun absMethod()        // abstract method
    fun sayHello() = "Hello there" // method
                with default implementation
}
```

Interface

Example

Visibility Control

- The Kotlin visibility modifiers are the keywords that set the visibility of classes, objects, interface, constructors, functions as well as properties and their setters.
- Though getters always have the same visibility as their properties, so we can not set their visibility.

Visibility Control

- There are four visibility modifiers in Kotlin:
  - public
  - private
  - protected
  - internal
- The default visibility is public. These modifiers can be used at multiple places such as class header or method body.

Public Modifier

- Public modifier is accessible from anywhere in the project workspace. If no access modifier is specified, then by default it will be in the public scope. In all our previous examples, we have not mentioned any modifier, hence, all of them are in the public scope.
- Following is an example to understand more on how to declare a public variable or method.

```
class publicExample {
    val i = 1
    fun doSomething() {
    }
}
```

Private Modifier

- The classes, methods, packages and other properties can be declared with a private modifier. This modifier has almost the exact opposite meaning of public which means a private member can not be accessed outside of its scope.
- Once anything is declared as private, then it can be accessible within its immediate scope only. For instance, a private package can be accessible within that specific file. A private class or interface can be accessible only by its data members, etc.

```
private class privateExample {
    private val i = 1
    private val doSomething() {
    }
}
```

Private Modifier

- The classes, methods, packages and other properties can be declared with a private modifier. This modifier has almost the exact opposite meaning of public which means a private member can not be accessed outside of its scope.
- Once anything is declared as private, then it can be accessible within its immediate scope only. For instance, a private package can be accessible within that specific file. A private class or interface can be accessible only by its data members, etc.

```
private class privateExample {
    private val i = 1
    private val doSomething() {
    }
}
```

Private Modifier

- Protected is another access modifier for Kotlin, which is currently not available for top level declaration like any package cannot be protected. A protected class or interface or properties or function is visible to the class itself and it's subclasses only.

```
package one;
class A() {
   protected val i = 1
}
class B : A() {
   fun getValue() : Int {
      return i
   }
}
```

Internal Modifier

- Internal is a newly added modifier in Kotlin. If anything is marked as internal, then the specific field will marked as the internal field. An Internal package is visible only inside the module under which it is implemented. An internal class interface is visible only by other class present inside the same package or the module.

```
package one
internal class InternalExample {
}
class publicExample{
    internal val i = 1
    internal fun doSomething() {
    }
}
```

Visibility Control

- There are four visibility modifiers in Kotlin:
  - public
  - private
  - protected
  - internal
- The default visibility is public. These modifiers can be used at multiple places such as class header or method body.

Data Class

- There are following conditions for a Kotlin class to be defined as a Data Class:
  - 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.
  - The class may extend other classes or implement interfaces. If you are using Kotlin version before 1.1, the class can only implement interfaces.

Data Class

- It's simple to define a Kotlin Data Class. If a Kotlin class is marked with data keyword then it becomes a data class. For example:

```
data class Book(val name: String, val
publisher: String, var reviewScore: Int)
```

- Good thing about Kotlin Data Class is that when you declare a Kotlin Data Class, the compiler generates Constructor, toString(), equals(), hashCode(), and additional copy() and componentN() functions automatically.

Sealed Class

- Sealed allows the developers to maintain a data type of a predefined type.
- To make a sealed class, we need to use the keyword sealed as a modifier of that class.
- A sealed class can have its own subclass but all those subclasses need to be declared inside the same Kotlin file along with the sealed class.

Summary

- List: A list is an ordered collection of elements where duplicates are allowed and each element has a specific index.
- Set: A set is a collection of unique elements that are not stored in any specific order.
- Map: A map is a collection of key-value pairs where each key is unique and maps to a single value.

Thank you

@mitu_skillologies

@mITuSkillologies

@mitu_group

@mitu-skillologies

@MITUSkillologies

kaggle

@mituskillologies

Web Resources
https://mitu.co.in
http://tusharkute.com

@mituskillologies

contact@mitu.co.in

tushar@tusharkute.com