MOBILE APPLICATION DEVELOPMENT

ANDROID (2017)

LECTURE 03: CLASSES, OBJECTS, AND FUNCTIONS

KOTLIN FEATURES: CLASSES

- Kotlin classes are declared with the `class` keyword and a name.
  `class Empty` is a complete class declaration.
- Classes may have primary constructors, declared along with their name. Adding mutability specifiers to constructor parameters causes them to be added to the class as properties.
  `class NotEmpty constructor(val x: String) {}` is also a complete class declaration.
  - This class would have one property, `x`, which is a String.
  - If the primary constructor has default visibility, the word `constructor` can be omitted.
- Classes may have properties outside their primary constructors, which may reference parameters from the primary constructor (note that the class below does NOT have a property called 'name'):

```
class Customer(name: String) {
    val upperCaseName: String = name.toUpperCase()
}
```

KOTLIN FEATURES: CLASSES

- Classes may have secondary constructors, which must call the primary constructor of the class if such a constructor exists:

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.addChild(this) // Assumes the class has an addChild() function.
    }
}
```

  If a class has no primary constructor, secondary constructors do not need to call it:

```
class Person {
    constructor(name: String, parent: Person) {
        parent.addChild(this)
    }
    constructor(name: String) {}
}
```

  Classes are instantiated by calling their constructors like functions: `val dave: Person = Person("Dave")`

KOTLIN FEATURES: CLASS PROPERTIES

- Properties on classes have default getters (and setters if they are mutable), and are accessed by name as though they were fields in Java:
  `person.name` is how the `name` property of a Person would be accessed.

- Getters or setters for class properties can be customized using `get` and `set` functions declared alongside the property. Properties have an automatic backing field which can be referred to in these functions with the `field` keyword:

```
//Inside a class...
var counter: Int = 0 // Initializer values are directly written to backing fields.
    get() = field // Returns the backing field directly. (Don't do this, it's redundant.)
    set(value: Int) {
        if (value >= 0) field = value
    }
```

KOTLIN FEATURES: CLASS MODIFIERS

- Kotlin supports a number of modifiers on classes, including:
  - The `abstract` modifier indicates a class does not have an implementation and must be inherited from and defined to be useful.
  - The `open` modifier means a class may be inherited from (`abstract` implies this).
  - The `inner` modifier means a class is defined inside another class and can access the members of the enclosing class.
  - The `sealed` modifier means that all of the potential subclasses of a class must be defined along with it, making it easy to identify its entire inheritance tree.

KOTLIN FEATURES: INHERITANCE

- Classes marked with `open` may be subclassed, and open members of open classes may be overridden:

```
open class Base {
    open fun overridable() {}
    fun notOverridable() {}
}

class Derived() : Base() {
    override fun overridable() {}
}
```

- Open properties of open classes may also be overridden if new initializers or getters are defined:

```
open class Base {
    open val overridable: Int get() { ... }
}

class Derived() : Base() {
    override var overridable: Int = 42 // Can change val properties to var in overrides, but not vice-versa.
}
```

KOTLIN FEATURES: INTERFACES

- Interfaces in Kotlin may specify abstract functions and/or properties, and may optionally specify implementations of those functions/properties:

```
interface MyInterface {
    val property: String
    val definedProperty: String get() = "DEFINED"
    fun abstract()
    fun defined(): String = property
}

class implementer: MyInterface {
    override val property: String = "Overridden"
    override fun abstract() {}
}
```

KOTLIN FEATURES: DATA CLASSES

- A class which does nothing except hold data can be simply defined as a `data class` in Kotlin:

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

- Kotlin will automatically generate a variety of functions for data classes, such as `toString()` functions, `equals()` functions, and other basic capabilities which makes using such classes easier.

- Data classes must hold some data, so the constructor must have at least one parameter/property, the mutability of all constructor properties must be specified, and the class may not be marked `abstract`, `open`, `inner`, or `sealed`.

KOTLIN FEATURES: ENUM CLASSES

- Kotlin enums are algebraic data types, and can contain complex types as their cases (or simple types, as in other languages like Java):

```
enum class Direction { // Simple, Java-esque enum class.
   NORTH, SOUTH, WEST, EAST
}

enum class ProtocolState { // Enum class whose cases have the function 'signal'.
    WAITING {
        override fun signal() = TALKING
    },
    TALKING {
        override fun signal() = WAITING
    };
    abstract fun signal(): ProtocolState
}
```

KOTLIN FEATURES: INFIX FUNCTIONS AND LAMBDAS

- Functions in Kotlin may be defined as `infix`, meaning that they are called as an infix operator:

```
open class Number(val n: Int) {
    infix fun add(number: Number): Number = Number(this.n + number.n)
}

val sum: Number = Number(1) add Number(1) // Adds two Numbers and stores the result in 'sum'.
```

- Lambdas are unnamed functions which Kotlin can use in place of 'regular' functions:

```
ints.map { value -> value * 2 } // Everything after the word 'map' is a lambda.

ints.map { it -> it.toString() + { " Hi" }() } // Lambdas can call lambdas and functions.
```

KOTLIN FEATURES: OBJECT EXPRESSIONS

- Kotlin allows the creation of unnamed classes using object expressions:

```
fun foo() {
    val adHoc = object {
        var x: Int = 0
        var y: Int = 0
    }
    print(adHoc.x + adHoc.y)
}
```

- Objects cannot be used as types outside of private or local scopes, and trying to use them in another scope by returning them from a function creates an instance of type Any which is not particularly useful.
- Within private/local scopes, objects are quite useful as parameters to functions which require their inputs to adopt specific interfaces, since objects can implement interfaces.

KOTLIN FEATURES: OBJECT DECLARATIONS

- Named objects are called object declarations, and are Kotlin's equivalent to singletons:

```
object NameDropper {
    fun name(): String = "Name"
}
NameDropper.name() // Calls the 'name' function on the NameDropper singleton object.
```

- Classes can have a special object declared inside them called a companion object:

```
class NameDropper {
    companion object {
        fun name(): String = "Name"
    }
}
NameDropper.name() // Calls the 'name' function on the NameDropper class' companion object.
```

KOTLIN FEATURES: CONSTANTS AND LATE INITIALIZATION

- Compile-time constants may be defined with the `const` keyword at top-level scope or within an object, and must be of a primitive type or String with no custom getters:

```
const val TOP_LEVEL_CONSTANT: String = "Constant"

open class ClassWithConstants {
    companion object {
        const val CONSTANT_IN_OBJECT: Int = 42
    }
}
```

- Non-nullable class properties whose values cannot be known at the time the class is initialized may be declared as `lateinit`, and can be initialized later as long as they are initialized before use:

```
open class Late {
    lateinit var late: Any // No need to initialize this with a default value.
}
```

KOTLIN FEATURES: EXTENSIONS AND OPERATOR OVERLOADING

- Kotlin allows classes to be extended by prefixing function declarations with that class name:

```
open class DoNothing

fun DoNothing.doSomething() = print("Hello") // Adds a function to the DoNothing class.
```

- Kotlin allows for operator overloading by using predefined function names to override common operators, such as overriding the `+` operator for the Counter class below:

```
data class Counter(val dayIndex: Int) {
    operator fun plus(increment: Int): Counter {
        return Counter(dayIndex + increment)
    }
}
```