Programming with Types

Kevin Jones

@kevinrjones www.rocksolidknowledge.com

Types

- Interfaces
- Classes
- Inheritance
- Construction
- Data Classes

Interfaces

Basic definitions

Default Methods

Properties

Interfaces

```
interface Time {
    fun setTime(hours: Int, mins: Int = 0, secs: Int = 0)
    fun now() : Time = // return current time
}
```

What About Multiple Implementation?

Must override the method

Interfaces

```
interface A { fun doSomething() ={} }
interface B { fun doSomething() ={} }
class Foo : A, B {
    override fun doSomething() = {
        super<A>.doSomething()
    }
}
```

Classes

'final' by default

'abstract' classes

Modifiers

Sealed classes

Constructors

Data classes

Classes

final by default

```
class Student : Person {
    fun getClasses() {}
    //...
}
```

- Class is *final* by default
- Methods are *final* by default

```
open class Student : Person{
    open fun getClasses() {}
    //...
}
```

- Use *open* to show class can be derived from
- Use *open* to show function can be overridden

Classes

Can be abstract

```
abstract class Person{
    abstract fun getName()
    open fun workHard(){}
    fun goOnHoliday(){}
}
```

- Class is abstract
- getName must be implemented
- workHard may be overridden
- goOnHoliday cannot be overriden

Classes

Everything is public by default

No 'package-private'

But does have 'internal'

Sealed Classes

Used to restrict class Heirarchies

'Enums on steroids'

Sealed Classes

```
sealed class Event {
    class SendEvent(id: Int, to: String) : Event()
    class ReceiveEvent(id: Int, from: String) : Event()
}
```

Using Sealed Classes

```
fun handleEvent(e:Event) =
    when(e){
        is SendEvent -> print(e.to)
        is ReceiveEvent -> print(e.from)
    }
```

Classes

Constructing Instances

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



val kevin = Person("Kevin")
```

- Specify primary construcion parameters on class definition

Alternative Primary Constructor Usage

```
open class Person(name: String) {
    val name: String
    init {
        this.name = name
    }
}
open class Person(_name: String) {
    val name = _name
}
```

Classes

Secondary Constructors

Secondary Constructor Usage

```
open class Person(name: String) {
    constructor(name: String, age: Int) : this(name)
}
```

Classes

Secondary Constructors

Prefer default values

Primary Constructor Usage

```
open class Person(name: String, age = 0) {
}
```

Classes

Calling superclass constructors

Calling Superclass Constructors

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


class Student: Person {
    constructor(name: String) : super(name)
}
```

Classes

Using default constructors

```
open class Person


class Student : Person()
```

- Default constructor is generated
- Must call explicitly

Classes

Private constructors are supported

Usually used to inhibit construction

- eg to create a singleton

In Kotlin there is usually a better way

Data Classes

Provide a convenient way to override equals, hashCode and toString

Typically immutable classes

Kotlin also generates 'copy' method

Using Data Classes

```
data class Meeting(val name:String, val location:String)
val aMeeting = Meeting("A Meeting", "London")
val anotherMeeting = aMeeting.copy(location = "New York")
```

Summary

- Interfaces can have default methods
- Classes similar in many ways to Java
- Constructors part of class definition
- Constructors take default parameters
- 'final' by default
- 'sealed' restricts type hierarchy
- Data classes simplify some basics