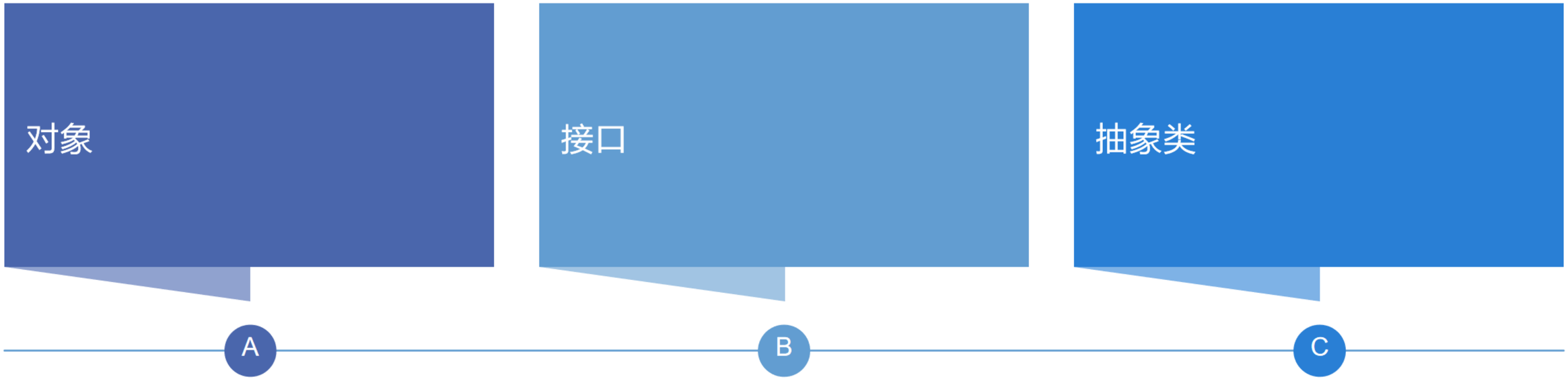
彻底掌握Kotlin



对象

object关键字

- 使用object关键字，你可以定义一个只能产生一个实例的类-单例
- 使用object关键字有三种方式
  - 对象声明
  - 对象表达式
  - 伴生对象

对象声明

- 对象声明有利于组织代码和管理状态，尤其是管理整个应用运行生命周期内的某些一致性状态。

```
1  object ApplicationConfig{
2      init {
3          println("loading config...")
4      }
5      fun setSomething(){
6          println("setSomething")
7      }
8  }
9  fun main() {
10     ApplicationConfig.setSomething()
11     println(ApplicationConfig)
12     println(ApplicationConfig)
13 }
```

对象表达式

- 有时候你不一定非要定义一个新的命名类不可，也许你需要某个现有类的一种变体实例，但只需用一次就行了，事实上，对于这种用完就丢的类实例，连命名都可以省了。这个对象表达式是XX的子类，这个匿名类依然遵循object关键字的一个规则，即一旦实例化，该匿名类只能有唯一一个实例存在。

```
1  open class Player {
2      open fun load() = "loading nothing."
3  }
4
5  fun main() {
6      val p = object : Player() {
7          override fun load() = "anonymous class load..."
8      }
9      println(p.load())
10 }
```

伴生对象

- 如果你想将某个对象的初始化和一个类实例捆绑在一起，可以考虑使用伴生对象，使用companion修饰符，你可以在一个类定义里声明一个伴生对象，**一个类里只能有一个伴生对象**。

```
1   import java.io.File
2
3   open class ConfigMap {
4       //只有初始化ConfigMap类或调用load函数时，伴生对象的内容才会载入。
5       //而且无论实例化ConfigMap类多少次，这个伴生对象始终只有一个实例存在。
6       companion object {
7           private const val PATH = "xxx"
8
9           fun load() = File(PATH).readBytes()
10      }
11  }
```

嵌套类

- 如果一个类只对另一个类有用，那么将其嵌入到该类中并使这两个类保持在一起是合乎逻辑的，可以使用嵌套类。

```
1   class Player2() {
2   //object Player2
3       class Equipment(var name: String) {
4           fun show() = println("equipment:$name")
5       }
6       fun battle() {
7           Equipment( name: "sharp knife").show()
8       }
9   }
10
11  fun main() {
12      Player2.Equipment( name: "AK47").show();
13  }
```

数据类

- 数据类，是专门设计用来存储数据的类
- 数据类提供了toString的个性化实现
- ==符号默认情况下，比较对象就是比较它们的引用值，数据类提供了equals和hash Code的个性化实现

```
1   data class Coordinate(var x: Int, var y: Int) {
2       //坐标值是否是正值
3       val isInBounds = x >= 0 && y >= 0
4   }
5   fun main() {
6       println(Coordinate( x: 1,  y: 5))
7       println(Coordinate( x: 1,  y: 5) == Coordinate( x: 1,  y: 5))
8   }
9
```

copy

- 除了重写Any类的部分函数，提供更好用的默认实现外，数据类还提供了一个函数，它可以用来方便地复制一个对象。假设你想创建一个Student实例，除了name属性，它拥有和另一个现有Student实例完全一样的属性值，如果Student是个数据类，那么复制现有Student实例就很简单了，只要调用copy函数，给想修改的属性传入值参就可以了。

```
data class Student(var name: String, val age: Int) {
    var score = 10
    private val hobby = "music"
    val subject:String
    init {
        println("initializing student")
        subject = "math"
    }
    constructor(_name: String):this(_name, age: 10){
        score = 20
    }
    override fun toString(): String {
        return "Student(name='$name', age=$age, " +
                "score=$score, hobby='$hobby', " +
                "subject='$subject')"
    }
}
```

```
fun main(args: Array<String>) {
    val s = Student( name: "jack")
    val copy : Student = s.copy( name: "Rose")
    println(copy)
}
```

解构声明

- 解构声明的后台实现就是声明component1、component2等若干个组件函数，让每个函数负责管理你想返回的一个属性数据，如果你定义一个数据类，它会自动为所有定义在主构造函数的属性添加对应的组件函数。

```
1  class PlayerScore(val experience: Int, val level: Int) {
2      operator fun component1() = experience
3      operator fun component2() = level
4  }
5
6  fun main() {
7      val (x: Int, y: Int) = PlayerScore( experience: 10, level: 4)
8  }
```

```
val (x: Int, y: Int) = Coordinate( x: 1, y: 5)
```

使用数据类的条件

- 正是因为上述这些特性，你才倾向于用数据类来表示存储数据的简单对象，对于那些经常需要比较、复制或打印自身内容的类，数据类尤其适合它们。然而，一个类要成为数据类，也要符合一定条件。总结下来，主要有三个方面：
  - 数据类必须有至少带一个参数的主构造函数
  - 数据类主构造函数的参数必须是val或var
  - 数据类不能使用abstract、open、sealed和inner修饰符

运算符重载

- 如果要将内置运算符应用在自定义类身上，你必须重写运算符函数，告诉编译器该如何操作自定义类。

```
1   data class Coordinate2(var x: Int, var y: Int) {
2       val isInBounds = x >= 0 && y >= 0
3
4       operator fun plus(other:Coordinate2) =
5           Coordinate2( x: x + other.x,  y: y + other.y)
6   }
7
8   fun main() {
9       val c1 = Coordinate2( x: 5,  y: 6)
10      val c2 = Coordinate2( x: 10,  y: 20)
11      println(c1 + c2)
12  }
```

运算符重载

- 常见操作符

操作符	函数名	作用
+	plus	把一个对象添加到另一个对象里
+=	plusAssign	把一个对象添加到另一个对象里，然后将结果赋值给第一个对象
==	equals	如果两个对象相等，则返回true，否则返回false
>	compareTo	如果左边的对象大于右边的对象，则返回true，否则返回false
[ ]	get	返回集合中指定位置的元素
..	rangeTo	创建一个range对象
in	contains	如果对象包含在集合里，则返回true

枚举类

- 枚举类，用来定义常量集合的一种特殊类。

```
1   enum class Direction {
2       EAST,
3       WEST,
4       SOUTH,
5       NORTH
6   }
7
8   fun main() {
9       println(Direction.EAST)
10  }
```

枚举类

- 枚举类也可以定义函数

```
1   //给枚举类添加一个主构造函数，
2   enum class Direction2(private val coordinate: Coordinate) {
3       //因为枚举类的构造函数带参数，所以定义每个枚举常量时，
4       //都要传入Coordinate对象，调用构造函数
5       EAST(Coordinate( x: 5,  y: -1)),
6       WEST(Coordinate( x: 1,  y: 0)),
7       SOUTH(Coordinate( x: 0,  y: 1)),
8       NORTH(Coordinate( x: -1,  y: 0));
9
10      fun updateCoordinate(playerCoordinate: Coordinate) =
11          Coordinate( x: playerCoordinate.x + coordinate.x,
12              y: playerCoordinate.y + coordinate.y)
13  }
14
15  fun main() {
16      //调用函数时，使用的是枚举常量，所以应该这样调用
17      println(Direction2.EAST.updateCoordinate(Coordinate( x: 10, y: 20)))
18  }
```

代数数据类型

- 可以用来表示一组子类型的闭集，枚举类就是一种简单的ADT。

```
enum class LicenseStatus {
    UNQUALIFIED,
    LEARNING,
    QUALIFIED;
}

class Driver(var status: LicenseStatus) {
    fun checkLicense(): String {
        //不用使用else语句，且编译器会帮你检查代码处理是否有遗漏
        return when (status) {
            LicenseStatus.UNQUALIFIED -> "没资格"
            LicenseStatus.LEARNING -> "在学"
            LicenseStatus.QUALIFIED -> "有资格"
        }
    }
}
```

密封类

- 对于更复杂的ADT，你可以使用Kotlin的密封类（sealed class）来实现更复杂的定义，

  密封类可以用来定义一个类似于枚举类的ADT，但你可以更灵活地控制某个子类型。

- 密封类可以有若干个子类，要继承密封类，这些子类必须和它定义在同一个文件里。

```kotlin
sealed class LicenseStatus2 {
    object UnQualified : LicenseStatus2()
    object Learning: LicenseStatus2()
    class Qualified(val licenseId: String) : LicenseStatus2()
}

class Driver2(var status: LicenseStatus2) {
    fun checkLicense(): String {
        return when (status) {
            is LicenseStatus2.UnQualified -> "没资格"
            is LicenseStatus2.Learning -> "在学"
            is LicenseStatus2.Qualified ->
                "有资格，驾驶证编号：" +
                        "${(this.status as LicenseStatus2.Qualified).licenseId}"
        }
    }
}
```

接口

接口定义

- Kotlin规定所有的接口属性和函数实现都要使用override关键字，接口中定义的函数并不需要open关键字修饰，他们默认就是open的。

```
interface Movable {
    val maxSpeed: Int
    var wheels: Int

    fun move(movable: Movable): String
}
class Car(_name: String, override var wheels: Int = 4) :Movable {

    override val maxSpeed: Int
        get() = TODO( reason: "not implemented")

    override fun move(movable: Movable): String {
        TODO( reason: "not implemented")
    }
}
```

默认实现

- 只要你愿意，你可以在接口里提供默认属性的getter方法和函数实现。

```
interface Movable {
    val maxSpeed: Int
        get() = (0..4).shuffled().last()
    var wheels: Int

    fun move(movable: Movable): String
}

class Car(
    _name: String,
    override var wheels: Int = 4
) :Movable {
    override var maxSpeed: Int
        get() = super.maxSpeed
        set(value) {}

    override fun move(movable: Movable): String {
        TODO( reason: "not implemented")
    }
}
```

抽象类

抽象类

- 要定义一个抽象类，你需要在定义之前加上abstract关键字，除了具体的函数实现，抽象类也可以包含抽象函数—只有定义，没有函数实现。

```
1  abstract class Gun(
2      val range: Int
3  ) {
4      abstract fun trigger(): String
5  }
6  class AK47(
7      _price: Int
8  ) : Gun(range = 100) {
9
10     override fun trigger(): String {
11         return "AK47 shooting..."
12     }
13 }
14 fun main() {
15     val g:Gun = AK47( price: 500)
16     println(g.trigger())
17 }
```