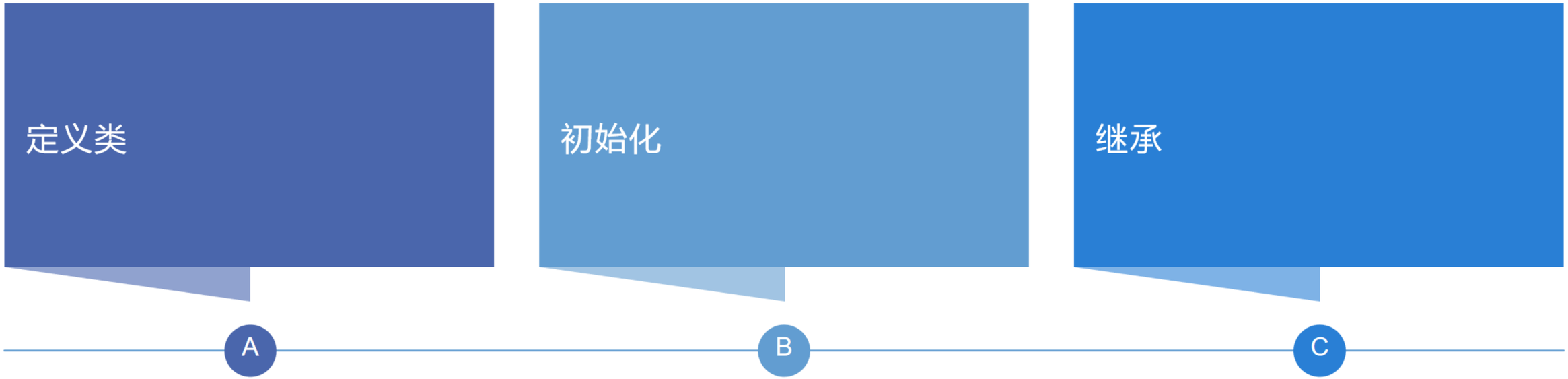
彻底掌握Kotlin



定义类

field

- 针对你定义的每一个属性，Kotlin都会产生一个field、一个getter、以及一个setter，field用来存储属性数据，你不能直接定义field，Kotlin会封装field，保护它里面的数据，只暴露给getter和setter使用。属性的getter方法决定你如何读取属性值，每个属性都有getter方法，setter方法决定你如何给属性赋值，所以只有可变属性才会有setter方法，尽管Kotlin会自动提供默认的getter和setter方法，但在需要控制如何读写属性数据时，你也可以自定义他们。

field

- 定义一个Player类

```
1   import kotlin.math.absoluteValue
2
3   class Player{
4       var name = "abc"
5           get() = field.capitalize()
6           set(value) {
7               field = value.trim()
8           }
9
10      var age = 10
11          get() = field.absoluteValue
12          private set(value) {
13              field = value.absoluteValue
14          }
15  }
```

计算属性

- 计算属性是通过一个覆盖的get或set运算符来定义，这时field就不需要了。

```
1    class Player{
2        val rolledValue
3            get() = (1..6).shuffled().first()
4    }
```

防范竞态条件

- 如果一个类属性既可空又可变，那么引用它之前你必须保证它非空，一个办法是用also标准函数。

```
1  class Player{
2      var words:String? = "hello"
3
4      fun saySomething(){
5          words?.also { it: String
6              println("Hello ${it.toUpperCase()}")
7          }
8      }
9  }
```

初始化

主构造函数

- 我们在Player类的定义头中定义一个主构造函数，使用临时变量为Player的各个属性提供初始值，在Kotlin中，为便于识别，临时变量（包括仅引用一次的参数），通常都会以下划线开头的名字命名。

```
class Player(
    _name: String,
    _age: Int,
    _isNormal: Boolean
) {
    var name = _name
        get() = field.capitalize()
        private set(value){
            field = value.trim()
        }

    var age = _age
    var isNormal = _isNormal
}

fun main() {
    var player = Player( name: "Jack", age: 20, isNormal: true)
}
```

在主构造函数里定义属性

- Kotlin允许你不使用临时变量赋值，而是直接用一个定义同时指定参数和类属性，通常，我们更喜欢用这种方式定义类属性，因为他会减少重复代码。

```
class Player2(
        _name: String,
        var age: Int,
        val isNormal: Boolean
) {
        var name = _name
            get() = field.capitalize()
            private set(value) {
                field = value.trim()
            }
}
```

次构造函数

- 有主就有次，对应主构造函数的是次构造函数，我们可以定义多个次构造函数来配置不同的参数组合。

```
12      constructor(name: String) : this(name,
13                                       age = 100,
14                                       isNormal = false)
15
```

次构造函数

- 使用次构造函数，定义初始化代码逻辑。

```
16      constructor(name: String, age: Int) : this(name,
17                                                 age,
18                                                 isNormal = false) {
19          this.name = name.toUpperCase()
20      }
```

默认参数

- 定义构造函数时，可以给构造函数参数指定默认值，如果用户调用时不提供值参，就使用这个默认值。

```
class Player3(
        _name: String,
        var age: Int = 20,
        private val isNormal: Boolean
) {
        var name = _name
            get() = field.capitalize()
            private set(value) {
                field = value.trim()
            }
}
```

初始化块

- 初始化块可以设置变量或值，以及执行有效性检查，如检查传给某构造函数的值是否有效，初始化块代码会在构造类实例时执行。

```
12      init {
13          require( value: age > 0){"age must be positive."}
14          require(name.isNotBlank()) {"player must have a name."}
15      }
```

初始化顺序

- 主构造函数里声明的属性
- 类级别的属性赋值
- init初始化块里的属性赋值和函数调用
- 次构造函数里的属性赋值和函数调用

初始化顺序

```
1  class Student(_name: String, val age: Int) {
2      var name = _name
3      var score = 10
4      private val hobby = "music"
5      val subject:String
6
7      init {
8          println("initializing student")
9          subject = "math"
10     }
11
12     constructor(_name: String):this(_name, age: 10){
13         score = 20
14     }
15 }
```

①

②

③

④

```
67     public Student(@NotNull String _name, int age) {
68         Intrinsics.checkParameterIsNotNull(_name, "_name");
69         super();
70         this.age = age;
71         this.name = _name;
72         this.score = 10;
73         this.hobby = "music";
74         String var3 = "initializing student";
75         boolean var4 = false;
76         System.out.println(var3);
77         this.subject = "math";
78     }
79
80     public Student(@NotNull String _name) {
81         Intrinsics.checkParameterIsNotNull(_name, "_name");
82         this(_name, 10);
83         this.score = 20;
84     }
```

延迟初始化

- 使用lateinit关键字相当于做了一个约定：在用它之前负责初始化
- 只要无法确认lateinit变量是否完成初始化，可以执行isInitialized检查

```
1  class Player4 {
2      lateinit var equipment:String
3      fun ready(){
4          equipment = "sharp knife"
5      }
6      fun battle(){
7          if (::equipment.isInitialized) println(equipment)
8      }
9  }
```

惰性初始化

- 延迟初始化并不是推后初始化的唯一方式，你也可以暂时不初始化某个变量，直到首次使用它，这个叫作惰性初始化。

```
class Player5(_name: String) {
    var name = _name
    val config by lazy {loadCofig()}
    private fun loadCofig():String{
        println("loading...")
        return "xxx"
    }
}
```

初始化陷阱一

- 在使用初始化块时，顺序非常重要，你必须保证块中的所有属性已完成初始化。

```
1  class Player6(){
2      init {
3          val bloodBonus:Int = blood.times( other: 4)
4      }
5      val blood = 100
6  }
```

初始化陷阱二

- 这段代码编译没有问题，因为编译器看到name属性已经在init块里初始化了，但代码一运行，就会抛出空指针异常，因为name属性还没赋值，firstLetter函数就应用它了。

```
1  class Player7(){
2      val name:String
3      private fun firstLetter() = name[0]
4
5      init {
6          println(firstLetter())
7          name = "Jack"
8      }
9  }
```

初始化陷阱三

- 因为编译器看到所有属性都初始化了，所以代码编译没问题，但运行结果却是null，问题出在哪？在用initPlayerName函数初始化playerName时，name属性还未完成初始化。

```
class Player8(_name:String){
    val playerName:String = initPlayerName()
    val name:String = _name

    private fun initPlayerName() = name
}
```

继承

继承

- 类默认都是封闭的，要让某个类开放继承，必须使用open关键字修饰它。

```
1  open class Product(val name:String){
2      fun description() = "Product: $name"
3
4      fun load() = "Nothing..."
5  }
6
7  class LuxuryProduct : Product( name: "Luxury")
```

函数重载

- 父类的函数也要以open关键字修饰，子类才能覆盖它。

```
1  open class Product(val name:String){
2      fun description() = "Product: $name"
3
4      open fun load() = "Nothing..."
5  }
6
7  class LuxuryProduct : Product( name: "Luxury"){
8
9      override fun load() = "LuxuryProduct loading..."
10 }
```

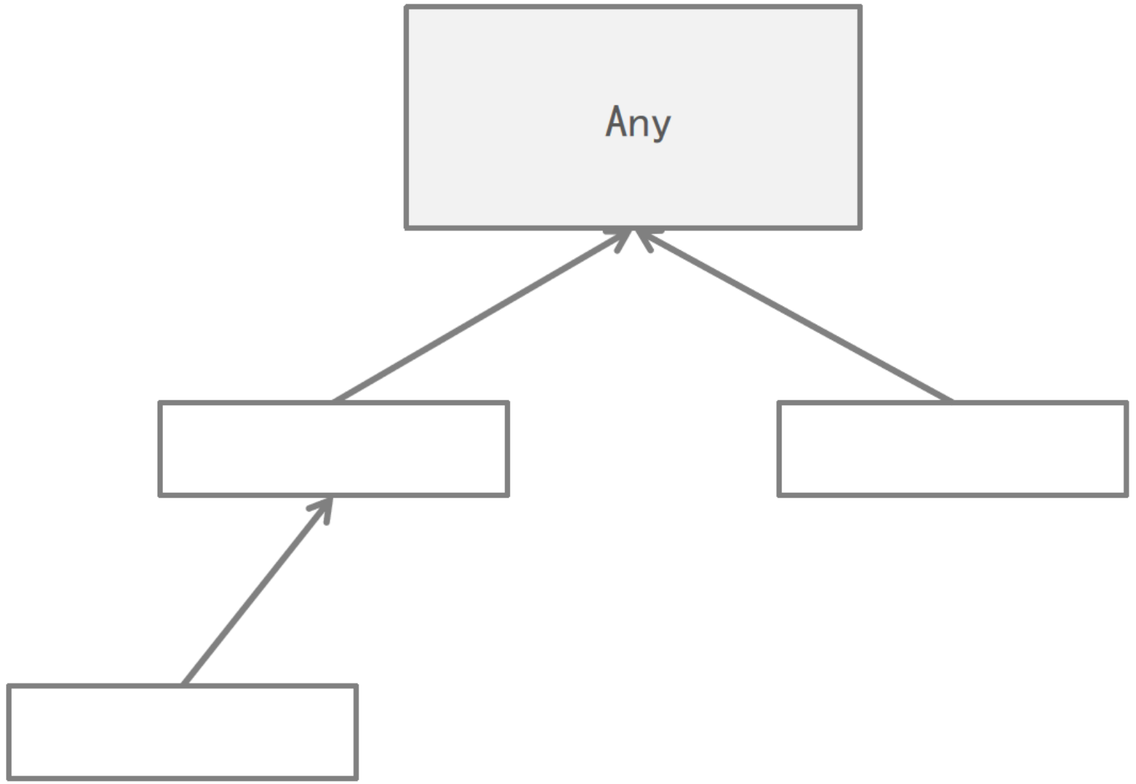
类型检测

- Kotlin的is运算符是个不错的工具，可以用来检查某个对象的类型。

```
10  fun main() {
11      val p = LuxuryProduct()
12      println(p is LuxuryProduct)
13      println(p is Product)
14  }
```

Kotlin层次

- 无须在代码里显示指定，每一个类都会继承一个共同的叫作Any的超类。



类型转换

- as操作符声明，这是一个类型转换。

```
12    fun sale(p: Product) {
13        println(p.load())
14    }
15
16    fun main() {
17        val p = LuxuryProduct()
18        sale((p as Product))
19    }
```

智能类型转换

- Kotlin编译器很聪明，只要能确定any is 父类条件检查属实，它就会将any当做子类类型对待，因此，编译器允许你不经类型转换直接使用。

```
12    fun sale(p: Product) {
13        println(p.load())
14    }
15
16    fun main() {
17        val p = LuxuryProduct()
18        //sale(p as Product)
19        sale(p)
20    }
```