类：

还是使用class关键字

构造函数：

在Kotlin中的一个类可以有一个主构造函数和一个或多个此构造函数。

主构造函数是类头的一部分：它跟在类名（和可选的类型参数）后：

**class** Person **constructor**(firstName: String) {

}

如果主构造函数没有任何注解或者可见性修饰符，可以省略constructor关键字

主构造函数不能包含任何的代码。初始化的代码可以放到以init关键字作为前缀的初始化块中：

class InitOrderDemo(name: String) {

val firstProperty = "First property: $name".also(::println)

init {

println("First initializer block that prints ${name}")

}

val secondProperty = "Second property: ${name.length}".also(::println)

init {

println("Second initializer block that prints ${name.length}")

}

}

在实例初始化期间，初始化块按照它们出现在类体中的顺序执行，与属性初始化器交织在一起

注意，主构造的参数可以在初始化块中使用。它们也可以在类体内声明的属性初始化器中使用：

**class** Customer(name: String) {

**val** customerKey = name.toUpperCase()

}

事实上，声明属性以及从主构造函数初始化属性，Kotlin 有简洁的语法：

**class** Person(**val** firstName: String, **val** lastName: String, **var** age: Int) {

// ……

}

与普通属性一样，主构造函数中声明的属性可以是可变的（var）或只读的（val）。

如果构造函数有注解或可见性修饰符，这个constructor关键字是必需的，并且在这些修饰符在它前面：

**class** Customer **public** @Inject **constructor**(name: String) { …… }

次构造器：

前缀有constructor的次构造器：

**class** Person {

**constructor**(parent: Person) {

parent.children.add(**this**)

}

}

如果类有一个主构造函数，每个次构造函数需要委托给主构造函数，可以直接委托或者通过别的次构造函数间接委托。委托到同一个类的另一个构造函数用this关键字即可

**class** Person {

**constructor**(parent: Person) {

parent.children.add(**this**)

}

}

注意：

初始化块中的代码实际上会成为主构造函数的一部分。委托给主构造函数会作为次构造函数的第一条语句，因此所有初始化块中的代码都会在此构造函数体之前执行。即使该类没有主构造函数，这种委托仍会隐式发生，并且仍会执行初始化块：

class Constructors {

init {

println("Init block")

}

constructor(i: Int) {

println("Constructor")

}

}

如果一个非抽象类没有声明任何（主或次）构造函数，它会有一个生成的不带参数的主构造器函数。构造函数的可见性是public。如果不希望有一个公有的构造函数，则需要声明一个带有非默认可见性的空的主构造函数：

**class** DontCreateMe **private** **constructor** () {

}

****注意****：在 JVM 上，如果主构造函数的所有的参数都有默认值，编译器会生成 一个额外的无参构造函数，它将使用默认值。这使得 Kotlin 更易于使用像 Jackson 或者 JPA 这样的通过无参构造函数创建类的实例的库。

**class** Customer(**val** customerName: String = "")

创建类的实例：

**val** invoice = Invoice()

**val** customer = Customer("Joe Smith")

注意：Kotlin并没有new关键字。

继承：

在Kotlin中所有类都有一个共同的超类Any

Any不是java.lang.Object，它除了equals（），hasCode（）和toString（）外没有任何成员

要声明一个显示的超类型，把类型放到类头的冒号之后：

**open** **class** Base(p: Int)

**class** Derived(p: Int) : Base(p)

如果该类有一个主构造函数，其基类可以（并且必须）用基类型的主构造函数参数就地初始化。

如果类没有主构造函数，那么每个次构造函数必须使用super关键字初始化其基类型，或委托给另一个构造函数做到这一地。注意，在这种情况下，不同的次构造函数可以调用基类型额不同的构造函数：

**class** MyView : View {

**constructor**(ctx: Context) : **super**(ctx)

**constructor**(ctx: Context, attrs: AttributeSet) : **super**(ctx, attrs)

}

Open标注与java中的final相反，它允许其他类从这个类继承。默认情况下在Kotlin中所有的类都是final

覆盖方法：

与java不同，Kotlin需要显式标注可覆盖的成员和覆盖后的成员：

**open** **class** Base {

**open** **fun** v() {}

**fun** nv() {}

}**class** Derived() : Base() {

**override** **fun** v() {}

}

如果函数没有标注open，则子类中不允许定义相同签名的函数，不论加不加override。在一个final类中，开放成员是禁止的。

标记为override的成员本身是开放的，也就是说，它可以在子类覆盖。如果想禁止再次覆盖，使用final关键字：

**open** **class** AnotherDerived() : Base() {

**final** **override** **fun** v() {

}

}

覆盖属相：

同样必须以override开头，并且它们必须具有兼容的类型。每个声明的属性可以由具有初始化器的属性或者具有getter方法的属性覆盖

**open** **class** Foo {

**open** **val** x: Int **get**() { …… }

}

**class** Bar1 : Foo() {

**override** **val** x: Int = ……

}

也可以用一个var属性覆盖一个val属性，但反之不可以。因为一个val属性本质上声明了一个getter方法，而将其覆盖为var只是在子类中额外声明一个setter方法。

可以在主构造函数中使用override关键字作为属性声明的一部分：

**interface** Foo {

**val** count: Int

}

**class** Bar1(**override** **val** count: Int) : Foo

**class** Bar2 : Foo {

**override** **var** count: Int = 0

}

调用超类的方法和成员变量，还是用super

在一个内部类中访问外部类的超类，可以通过由外部类名限定的 super 关键字来实现：super@Outer：

**class** Bar : Foo() {

**override** **fun** f() { /\* …… \*/ }

**override** **val** x: Int **get**() = 0

**inner** **class** Baz {

**fun** g() {

**super**@Bar.f() // 调用 Foo 实现的 f()

println(**super**@Bar.x) // 使用 Foo 实现的 x 的 getter

}

}

}

覆盖规则：

在kotlin中，如果一个类从它的直接超类继承相同成员的多个实现，它必须覆盖这个成员并提供自己的实现。为了表示采用从哪个超类型继承的实现，使用由尖括号中超类型名限定的super

**open** **class** A {

**open** **fun** f() {

print("A")

}

**fun** a() {

print("a")

}

}

**interface** B {

**fun** f() {

print("B")

} // 接口成员默认就是“open”的

**fun** b() {

print("b")

}

}

**class** C() : A(), B {

// 编译器要求覆盖 f()：

**override** **fun** f() {

**super**<A>.f() // 调用 A.f()

**super**<B>.f() // 调用 B.f()

}

}

抽象类：

类和其中的某些成员可以声明为abstract。不需要用open标注一个抽象类或者函数。

**open** **class** Base {

**open** **fun** f() {}

}

**abstract** **class** Derived : Base() {

**override** **abstract** **fun** f()

}

## 伴生对象：

在Kotlin中类没有静态方法。

如果在你的类内声明了一个[伴生对象](http://www.kotlincn.net/docs/reference/object-declarations.html" \l "伴生对象)， 你就可以使用像在 Java/C# 中调用静态方法相同的语法来调用其成员，只使用类名作为限定符。

属性和字段：

声明属性：

Kotlin的类可以有属性，属性可以用关键字var声明为可变的，val为只读

编译期常量：

用const修饰符标记为编译期常量

- 位于顶层或者是object的一个成员

- 用String或原生类型值初始化

延迟初始化属性与变量：

有时声明一个非空类型的属性必须在构造函数中初始化。但是有时候我不想这样做，我可能需要在另外一个方法中进行初始化，这个时候就可以用lateinit修饰符标记该属性：

**public** **class** MyTest {

**lateinit** **var** subject: TestSubject

@SetUp **fun** setup() {

subject = TestSubject()

}

@Test **fun** test() {

subject.method() // 直接解引用

}

}

该修饰符只能用在类体中的属性。

### 检测一个 lateinit var 是否已初始化（自 1.2 起）

要检测一个 lateinit var 是否已经初始化过，请在[该属性的引用](http://www.kotlincn.net/docs/reference/reflection.html" \l "属性引用)上使用 .isInitialized：

**if** (foo::bar.isInitialized) {

println(foo.bar)

}

此检测仅对可词法级访问的属性可用，即声明位于同一个类型内、位于其中一个外围类型中或者位于相同文件的顶层的属性。

接口：

还是使用interface来定义接口：

**interface** MyInterface {

**fun** bar()

**fun** foo() {

// 可选的方法体

}

}

## 实现接口

一个类或者对象可以实现一个或多个接口。

**class** Child : MyInterface {

**override** **fun** bar() {

// 方法体

}

}

## 接口中的属性

你可以在接口中定义属性。在接口中声明的属性要么是抽象的，要么提供访问器的实现。在接口中声明的属性不能有幕后字段（backing field），因此接口中声明的访问器不能引用它们。

**interface** MyInterface {

**val** prop: Int // 抽象的

**val** propertyWithImplementation: String

**get**() = "foo"

**fun** foo() {

print(prop)

}

}

**class** Child : MyInterface {

**override** **val** prop: Int = 29

}

解决覆盖冲突，仍然用super<>