接口

Kotlin 的接口可以既包含抽象方法的声明也包含实现。与抽象类不同的是，接口无法保存状态。它可以有属性但必须声明为抽象或提供访问器实现。

使用关键字 `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
}
```

接口继承

一个接口可以从其他接口派生，从而既提供基类型成员的实现也声明新的函数与属

性。很自然地，实现这样接口的类只需定义所缺少的实现：

```
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}

data class Employee(
    // 不必实现“name”
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```

解决覆盖冲突

实现多个接口时，可能会遇到同一方法继承多个实现的问题。例如

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}
```

上例中，接口 *A* 和 *B* 都定义了方法 *foo()* 和 *bar()*。 两者都实现了 *foo()*, 但是只有 *B* 实现了 *bar()* (*bar()* 在 *A* 中没有标记为抽象， 因为在接口中没有方法体时默认为抽象）。因为 *C* 是一个实现了 *A* 的具体类，所以必须要重写 *bar()* 并实现这个抽象方法。

然而，如果我们从 *A* 和 *B* 派生 *D*，我们需要实现我们从多个接口继承的所有方法，并指明 *D* 应该如何实现它们。这一规则既适用于继承单个实现（*bar()*）的方法也适用

于继承多个实现（*foo()*）的方法。