**Interfaces in Kotlin**

Kotlin interfaces are similar to those of Java 8: they can contain definitions of abstract methods as well as implementations of non-abstract methods (similar to the Java 8 default methods), but

they can’t contain any state.

**interface** Clickable {

**fun** click()

**fun** showOff() = println("I'm clickable!")

}

**class** Button : Clickable {

**override** **fun** click() = println("I was clicked")

}

Unlike Java, using the **override** modifier is mandatory in Kotlin. You can redefine the behavior of the showOff() method, or you can omit it if you’re fine with the default behavior. The Kotlin compiler forces you to provide your own implementation when you implement two interface with same default method definition.

**class** Button : Clickable, Focusable{

**override** **fun** click() = println("I was clicked")

**override** **fun** showOff() {

**super**<Clickable>.showOff()

**super**<Focusable>.showOff()

}

}

**Open, final, and abstract modifiers**

Whereas Java’s classes and methods are open by default, Kotlin’s are final by default.

If you want to allow the creation of subclasses of a class, you need to mark the class with the open modifier. In addition, you need to add the open modifier to every property or method that can be overridden.

**open** **class** RichButton : Clickable { //This class is open: others can inherit

**fun** disable() {}//This function is "final": you can’t override it.

**open** **fun** animate() {}//This function is open: you may override it.

**override** **fun** click() {} //This function overrides an open function and is open as well.

}

Note that if you override a member of a base class or interface, the overriding

member will also be open by default. If you want to change this and forbid the subclasses of your class from overriding your implementation, you can explicitly mark the overriding member as final.

**Open classes and smart casts**

One significant benefit of classes that are final by default is that they enable smart casts in a larger variety of scenarios. Smart cast can only be used with a class property that is a val and that doesn’t have a custom accessor. This requirement means the property has to be final because otherwise a subclass could override the property and define a custom accessor, breaking the key requirement of smart casts.

**abstract** **class** Animated {

**abstract** **fun** animate()

**open** **fun** stopAnimating() {

}

**fun** animateTwice() {

}

}

**Visibility modifiers**

Basically, visibility modifiers in Kotlin are similar to those in Java. You have the same public , protected , and private modifiers. But the default visibility is different: if you omit a modifier, the declaration becomes public .As an alternative, Kotlin offers a new visibility modifier, internal , which means "visible inside a module." A module is a set of Kotlin files compiled together.

In Kotlin, visibility rules are simple, and a protected member is only visible in the class and its subclasses. Also note that extension functions of a class don’t get access to its private or protected members.

One more difference in visibility rules between Kotlin and Java is that an outer class doesn’t see private members of its inner (or nested) classes in Kotlin.

**Inner And nested classes**

**class** Outer {

**class** Inner {

/\* \*/

}

}

The difference is that Kotlin nested classes don’t have access to the outer class instance, unless you

specifically request that.

In Java declaring a nested class as static removes the implicit reference from that class to its enclosing class. A nested class in Kotlin with no explicit modifiers is the same as a static nested

class in Java.

To turn it into an inner class, so that it contains a reference to an outer class, you use the inner modifier.

**class** Outer {

**inner** **class** Inner {

**fun** getOuterReference(): Outer = **this**@Outer

}

}

**Sealed classes**

When you evaluate an expression using the when construct, the Kotlin compiler forces you to check for the default option.

**interface** Expr

**class** Num(**val** value: Int) : Expr

**class** Sum(**val** left: Expr, **val** right: Expr) : Expr

**fun** eval(e: Expr): Int =

**when** (e) {

**is** Num -> e.value

**is** Sum -> eval(e.right) + eval(e.left)

**else** ->

**throw** IllegalArgumentException("Unknown expression")

}

Kotlin provides a solution to this problem: sealed classes. You mark a superclass with the sealed modifier, and that restricts the possibility of creating subclasses. All the direct subclasses must be nested in the superclass:

**sealed** **class** Expr {

**class** Num(**val** value: Int) : Expr()

**class** Sum(**val** left: Expr, **val** right: Expr) : Expr()

}

**fun** eval(e: Expr): Int =

**when** (e) {

**is** Expr.Num -> e.value

**is** Expr.Sum -> eval(e.right) + eval(e.left)

}

Note that the sealed modifier implies that the class is open; you don’t need an explicit open modifier. Sealed classes can’t have inheritors defined outside of the class and a subclass can’t

be made a data class

**primary constructor and initializer blocks**

**class User(val nickname: String)**

"val" means the corresponding property is generated for the constructor parameter.

**class User constructor(\_nickname: String) {**

**val** nickname: String

**init** {

nickname = \_nickname

}

}

**class User(\_nickname: String) {**

**val** nickname = \_nickname

}

Above all are same

You can declare default values for constructor arguments just as you can for function arguments:

**class** User(**val** nickname: String,

**val** isSubscribed: Boolean = **true**)

**val** ***alice*** = User("Alice")

**val** ***bob*** = User(nickname = "Bob", isSubscribed = **false**)

If your class has a superclass, the primary constructor also needs to initialize the superclass. You can do so by providing the superclass constructor parameters after the superclass reference in the base class list:

**open** **class** User(**val** nickname: String) { }

**class** TwitterUser(nickname: String) : User(nickname) { }

Here’s how you make the primary constructor private:

**class** Secretive **private** **constructor**() {}

or

**class** Secretive {

**private** **constructor**()

}

**Secondary constructors**

**open** **class** View {

**constructor**(ctx: Context) {

}

**constructor**(ctx: Context, attr: AttributeSet) {

}

}

secondary constructor is introduced using the **constructor** keyword. You can declare as many secondary constructors as you need.

Here you define two constructors, each of which calls the corresponding constructor of the superclass using the **super**() keyword.

**class** MyButton : View {

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

}

**constructor**(ctx: Context, attr: AttributeSet): **super**(ctx, attr) {

}

}

Just as in Java, you also have an option to call another constructor of your own class from a constructor, using the **this**() keyword.

**class** MyButton : View {

**constructor**(ctx: Context) : **this**(ctx, MY\_STYLE) {

}

**constructor**(ctx: Context, attr: AttributeSet) : **super**(ctx, attr) {

}

}

**Implementing properties declared in interfaces**

In Kotlin, an interface can contain abstract property declarations.

**interface** User {

**val** nickname: String

}

This means classes implementing the User interface need to provide a way to obtain the value of nickname . The interface doesn’t specify whether the value should be stored in a backing field or obtained through a getter. Therefore, the interface itself doesn’t contain any state, and only classes implementing the interface may store the value if they need to.

**class** PrivateUser(**override** **val** nickname: String) : User

For PrivateUser, you use the concise syntax to declare a property directly in the primary constructor. This property implements the abstract property from User, so you mark it as **override** .

**class** SubscribingUser(**val** email: String) : User {

**override** **val** nickname: String

**get**() = email.substringBefore('@')

}

**class** FacebookUser(**val** accountId: Int) : User {

**override** **val** nickname = getFacebookName(accountId)

}

In addition to abstract property declarations, an interface can contain properties with getters and setters, as long as they don’t reference a backing field. Property doesn’t have a backing field: the result value is computed on each access.

**interface** User {

**val** email: String

**val** nickname: String

**get**() = email.substringBefore('@')

}

This interface contains the abstract property email, as well as the nickname property with a custom getter. The first property must be overridden in subclasses, whereas the second one can be inherited.

**Accessing a backing field from a getter or setter**

**class** User(**val** name: String) {

**var** address: String = "unspecified"

**set**(value: String) {

field = value

}

**get**() = field

}

The difference is between making a property that has a backing field and one that doesn’t. The way you access it doesn’t depend on whether the property has a backing field. The compiler will generate the backing field for the property if you either reference it explicitly or use the default accessor implementation. If you provide custom accessor implementations that don’t use field, the backing field won’t be present.

The accessor’s visibility by default is the same as the property’s. But you can change thisif you need to, by putting a visibility modifier before the get or set keyword.

**Universal object methods**

**TOSTRING()**

**class** Client(**val** name: String, **val** postalCode: Int) {

**override** **fun** toString() = "Client(name=$name, postalCode=$postalCode)"

}

**EQUALS()**

**class** Client(**val** name: String, **val** postalCode: Int) {

**override** **fun** equals(other: Any?): Boolean {

**if** (other == **null** || other **!is** Client)

**return** **false**

**return** name == other.name && postalCode == other.postalCode

}

}

"Any" is the analogue of java.lang.Object: a superclass of all classes in Kotlin. The nullable type "Any?" means "other" can be null.

The **is** check in Kotlin is the analogue of instanceof in Java.

**HASHCODE()**

**class** Client(**val** name: String, **val** postalCode: Int) {

**override** **fun** hashCode(): Int = name.hashCode() \* 31 + postalCode

}

**Data classes**

**data** **class** Client(**val** name: String, **val** postalCode: Int)

If you add the modifier data to your class, all the necessary methods are automatically generated

for you.

**COPY()**

Here’s what the copy() method would look like if you implemented it manually:

**class** Client(**val** name: String, **val** postalCode: Int) {

**fun** copy(name: String = **this**.name,

postalCode: Int = **this**.postalCode) = Client(name, postalCode)

}

**Object keyword**

The keyword defines a class and creates an instance of that class at the same time.

**object** Payroll {

**val** allEmployees = arrayListOf<Person>()

**fun** calculateSalary() {

**for** (person **in** allEmployees) {

}

}

}

Just like a class, an object declaration can contain declarations of properties, methods, initializer blocks, and so on. The only things that aren’t allowed are constructors (either primary or secondary)

Payroll.calculateSalary()

Payroll.allEmployees.add(Person(...))

**singletons**

**object** declaration is a way to define a singleton.

**object** CaseInsensitiveFileComparator : Comparator<File> {

**override** **fun** compare(file1: File, file2: File): Int {

**return** file1.getPath().compareTo(file2.getPath(),

ignoreCase = **true**)

}

}

**Companion objects**

Classes in Kotlin can’t have static members; Java’s static keyword isn’t part of the Kotlin language. As a replacement, Kotlin relies on package-level functions and object declarations .In most cases, it’s recommended that you use package-level functions. But package-level functions can’t access private members of a class. Thus, if you need to write a function that can be called without having a class instance but needs access to the internals of a class, you can write it as a member of an object declaration inside that class.

One of the objects defined in a class can be marked with a special keyword, companion . If you do that, you gain the ability to access the methods and properties of that object directly through the name of the containing class, without specifying the name of the object explicitly.

**class** A {

**companion** **object** {

**fun** bar() {

println("Companion object called")

}

}

}

A.bar()

The companion object has access to all private members of the class, and it’s an ideal candidate to implement the factory pattern.

In most cases, you refer to the companion object through the name of its containing

class, so you don’t need to worry about its name. But you can specify it if needed

**class** Person(**val** name: String) {

**companion** **object** Loader {

**fun** fromJSON(jsonText: String): Person = ...

}

}

Person.Loader.fromJSON("{name: 'Dmitry'}")

**Companion object implementing an interface**

**interface** JSONFactory<T> {

**fun** fromJSON(jsonText: String): T

}

**class** Person(**val** name: String) {

**companion** **object** : JSONFactory<Person> {

**override** **fun** fromJSON(jsonText: String): Person = ...

}

}

**COMPANION-OBJECT EXTENSIONS**

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

**companion** **object** {}

}

**fun** Person.Companion.fromJSON(json: String): Person {

....

}

**val** p = Person.fromJSON(json)

**Anonymous inner classes**

**val** listener = **object** : MouseAdapter() {

**override** **fun** mouseClicked(e: MouseEvent) {

...

}

**override** **fun** mouseEntered(e: MouseEvent) {

...

}

}

Declares an anonymous object extending MouseAdapter, Overrides MouseAdapter methods

Just as with Java’s anonymous classes, code in an object expression can access the variables in the function where it was created. But unlike in Java, this isn’t restricted to final variables; you can also modify the values of variables from within an object expression.