Inheritance

All classes in Kotlin have a common superclass Any, that is a default super for a class with no supertypes declared:

**class** Example // Implicitly inherits from Any

Any is not java.lang.Object; in particular, it does not have any members other than equals(), hashCode() and toString(). Please consult the [Java interoperability](https://kotlinlang.org/docs/reference/java-interop.html#object-methods) section for more details.

To declare an explicit supertype, we place the type after a colon in the class header:

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

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

If the class has a primary constructor, the base type can (and must) be initialized right there, using the parameters of the primary constructor.

If the class has no primary constructor, then each secondary constructor has to initialize the base type using the super keyword, or to delegate to another constructor which does that. Note that in this case different secondary constructors can call different constructors of the base type:

**class** MyView : View {

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

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

}

The open annotation on a class is the opposite of Java's final: it allows others to inherit from this class. By default, all classes in Kotlin are final, which corresponds to [Effective Java](http://www.oracle.com/technetwork/java/effectivejava-136174.html), Item 17: *Design and document for inheritance or else prohibit it*.

Overriding Methods

As we mentioned before, we stick to making things explicit in Kotlin. And unlike Java, Kotlin requires explicit annotations for overridable members (we call them *open*) and for overrides:

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

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

**fun** nv() {}

}

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

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

}

The override annotation is required for Derived.v(). If it were missing, the compiler would complain. If there is no open annotation on a function, like Base.nv(), declaring a method with the same signature in a subclass is illegal, either with override or without it. In a final class (e.g. a class with no open annotation), open members are prohibited.

A member marked override is itself open, i.e. it may be overridden in subclasses. If you want to prohibit re-overriding, use final:

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

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

}

# **Interfaces**

Interfaces in Kotlin are very similar to Java 8. They can contain declarations of abstract methods, as well as method implementations. What makes them different from abstract classes is that interfaces cannot store state. They can have properties but these need to be abstract or to provide accessor implementations.

An interface is defined using the keyword interface

**interface** MyInterface {

**fun** bar()

**fun** foo() {

// optional body

}

}

## Implementing Interfaces

A class or object can implement one or more interfaces

**class** Child : MyInterface {

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

// body

}

}

## Properties in Interfaces

You can declare properties in interfaces. A property declared in an interface can either be abstract, or it can provide implementations for accessors. Properties declared in interfaces can't have backing fields, and therefore accessors declared in interfaces can't reference them.

**interface** MyInterface {

**val** prop: Int // abstract

**val** propertyWithImplementation: String

**get**() = "foo"

**fun** foo() {

print(prop)

}

}

**class** Child : MyInterface {

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

}

## Resolving overriding conflicts

When we declare many types in our supertype list, it may appear that we inherit more than one implementation of the same method. For example

**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()

}

}

Interfaces A and B both declare functions foo() and bar(). Both of them implement foo(), but only Bimplements bar() (bar() is not marked abstract in A, because this is the default for interfaces, if the function has no body). Now, if we derive a concrete class C from A, we, obviously, have to override bar() and provide an implementation.

However, if we derive D from A and B, we need to implement all the methods which we have inherited from multiple interfaces, and to specify how exactly D should implement them. This rule applies both to methods for which we've inherited a single implementation (bar()) and multiple implementations (foo()).