Inheritance

We'll cover the following

When you use inheritance, you'll feel the extra layer of safety and protection that Kotlin provides. Since inheritance is one of the misused concepts in OO programming, Kotlin helps you make sure that your intentions are laid out very explicitly to the users of your classes.

Inheritance in Kotlin

Kotlin doesn't want classes to accidentally serve as a base class. As an author of a class, you have to provide explicit permission for your class to be used as a base class. Likewise, when writing a method, you have to tell Kotlin that it's OK for a derived class to override that method. Let's take a look at how Kotlin provides this safety net.

Unlike interfaces, classes in Kotlin are `final` by default—that is, you can't inherit from them. Only classes marked `open` may be inherited from. Only open methods of an open class may be overridden in a derived class and have to be marked with `override` in the derived. A method that isn't marked `open` or `override` can't be overridden. An overriding method may be marked `final override` to prevent a subclass from further overriding that method.

You may override a property, either defined within a class or within the parameter list of a constructor. A `val` property in the base may be overridden with a `val` or `var` in the derived. But a `var` property in the base may be overridden only using `var` in the derived. The reason for this restriction is that `val` only has a getter and you may add a setter in the derived by overriding with `var`. But you shouldn't

attempt to withdraw the setter that's for a base's `var` by overriding with a `val` in the derived.

Creating a base class

All these concepts will take shape in the next example. The `Vehicle` class that follows is marked as `open` and so can serve as a base class.

```
open class Vehicle(val year: Int, open var color: String) {
  open val km = 0

  final override fun toString() = "year: $year, Color: $color, KM: $km"

  fun repaint(newColor: String) {
    color = newColor
  }
}
```

inheritance.kts

The class takes two parameters in the constructor: the first defines a property that can't be overridden in any derived class of `Vehicle`, and the second is a property that may be overridden since it's marked as `open`. The property `km` defined within the class may also be overridden in a derived class. This class overrides the `toString()` method of its own base class `Any`, but prohibits any inheriting class from overriding that method. The method `repaint()` is final since it's not marked as `open`.

Creating a derived class

Next we'll create a derived class of Vehicle:

```
open class Car(year: Int, color: String) : Vehicle(year, color) {
  override var km: Int = 0
    set(value) {
      if (value < 1) {
        throw RuntimeException("can't set negative value")
      }

      field = value
    }

  fun drive(distance: Int) {
    km += distance
  }
}
```

inheritance.kts

The class `Car` derives from `Vehicle` and at the same time can serve as a base class

for any class that likes to extend it. The parameters of the constructor of `Car` are passed as arguments to the constructor of `Vehicle`. The colon notation is used to express inheritance of a class from another class, much like how inheritance from an interface was specified. Unlike Java, Kotlin doesn't distinguish between `implements` and `extends`—it's just inheritance.

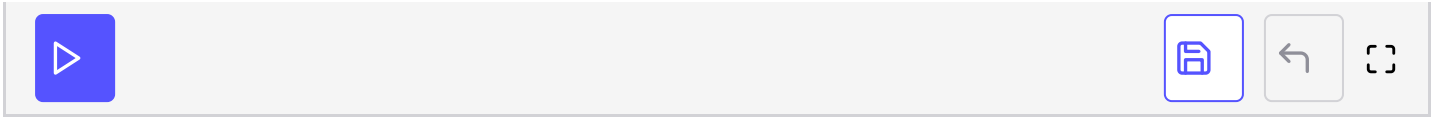
The class overrides the `km` property, provides a setter that checks the value to be greater than zero, and sets the acceptable value into the backing field for this property. There's no explicit getter, and the value in the backing field will be returned automatically when requested. The `drive()` method modifies the `km` property stored in `Car` and is `final` since it isn't marked `open`.

Let's create an instance of the `Car` class to study its behavior.

```
val car = Car(2019, "Orange")
println(car.year)  // 2019
println(car.color) // Orange

car.drive(10)
println(car) // year: 2019, Color: Orange, KM: 10

try {
  car.drive(-30)
} catch(ex: RuntimeException) {
  println(ex.message) // can't set negative value
}
```



inheritance.kts

The instance of `Car` takes values for `year` and `color` properties. Both these properties are passed on to the base and stored there. When `drive()` is called the first time, the `km` property stored in `Car`, and not the one in `Vehicle`, is modified. This value is displayed in the implicit call to `toString()` within `println(car)`.

Even though the `toString()` method is implemented in `Vehicle` and not in `Car`, when the `km` property is accessed within the `toString()` method, the overridden implementation in `Car` is used due to polymorphism. The second call to `drive()` fails since the value set into the `km` property can't be less than `1`.

Extending the class

We may derive further from `Car` if we like. The `FamilyCar` class that follows

extends `Car`:

```
class FamilyCar(year: Int, color: String) : Car(year, color) {
  override var color: String
    get() = super.color
    set(value) {
      if (value.isEmpty()) {
        throw RuntimeException("Color required")
      }

      super.color = value
    }
}
```

inheritance.kts

Unlike the `km` property in `Car`, which kept its value locally in its backing field, the `FamilyCar` doesn't store the value of `color` locally. Instead, by overriding both getter and setter, it fetches and forwards the value in these methods, respectively, from the base class. Since `Car` doesn't override `color`, the `FamilyCar` uses the `color` from `Vehicle`. But if the value set is empty, the change won't be accepted due to the overridden setter.

The constructor of `FamilyCar` passes the values to the base, but due to polymorphism the getter and setter in the derived will be used appropriately.

Let's use an instance of `FamilyCar` to see its behavior.

```
val familyCar = FamilyCar(2019, "Green")

println(familyCar.color) //Green

try {
  familyCar.repaint("")
} catch(ex: RuntimeException) {
  println(ex.message) // Color required
}
```

inheritance.kts

Even though `color` is stored within the `Vehicle`, the instance of `FamilyCar` takes over the validation of the property's value and doesn't permit a blank color.

In addition to the reasonable restrictions Kotlin places, it also ensures that when overriding, you may be more generous with the access restriction, but not stricter.

For example, you may make a `private` or `protected` member `public` in the derived, but you can't make a `public` member of base `protected` in the derived.

QUIZ

Q Which classes in Kotlin support inheritance?

Retake Quiz

We've seen Kotlin's support for inheritance—any class that's open may be used as a base class. Sometimes, though, we may want to restrict the derived classes to some

base class. Sometimes, though, we may want to restrict the derived classes to some

select classes. Kotlin provides sealed classes for that purpose, which we'll cover in the next lesson.