When to Choose Delegation over Inheritance?

We'll cover the following

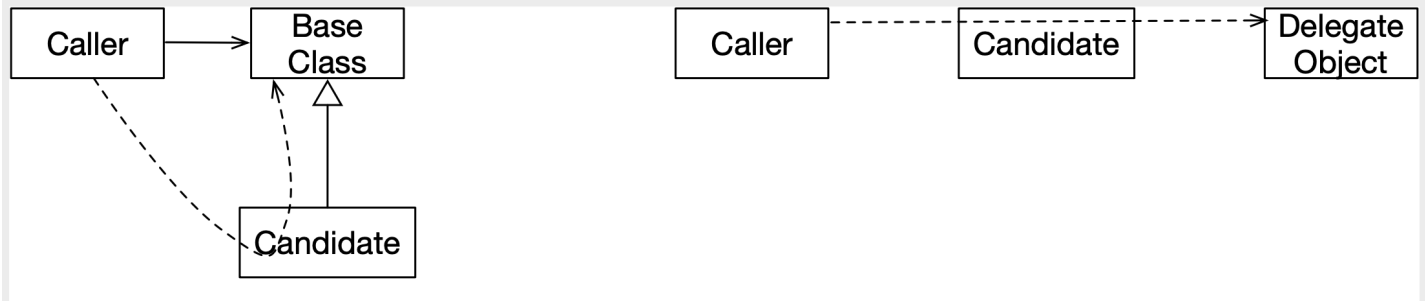
How to choose

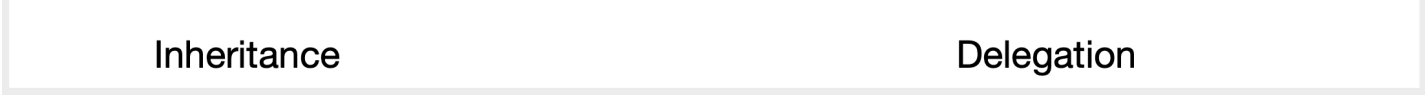
Both delegation and inheritance are useful, but you have to decide when one is a better choice than the other. Inheritance is common, highly used, and is a first-class feature in OO languages. Though delegation is more flexible, there's no special support for it in many OO languages. We often shy away from delegation since it takes more effort to implement than inheritance. Kotlin supports both inheritance and delegation, so we can freely choose between them based on the problem at hand. The following rules will serve you well in deciding which one to choose:

- Use inheritance if you want an object of a class to be used in place of an object of another class.
- Use delegation if you want an object of a class to simply make use of an object of another class.

Differences explained with an example

You can see the two design choices in the following figure—a `Candidate` class using inheritance on the left and delegation on the right.





When a `Candidate` class inherits from the `BaseClass`—like in the design on the left in the figure—an instance of the `Candidate` class carries an instance of the `BaseClass` within. Not only is that base instance (`BaseClass`) inseparable from the derived instance (`Candidate`) but its class is set in stone. The `Caller`, which references an instance of the base class at compile time, may actually be using an instance of derived at runtime without knowing. We celebrate this as the charm of polymorphism that's enabled through inheritance in statically typed OO languages. But achieving this behavior to freely interchange instances of derived—where an instance of base is expected—can be a slippery slope, as cautioned by Liskov's Substitution Principle (LSP)—see Agile Software Development, Principles, Patterns, and Practices. The issue is that the creator of the derived class must make sure that the implementation that overrides the base methods maintains the external observable behavior of the base class. In short, using inheritance greatly limits the degree of freedom for the designer of the derived class.

When a `Candidate` class delegates to a `DelegateObject`—like in the design on the right in the figure—an instance of the `Candidate` class holds a reference to a delegate. The actual class of the delegate may vary, within reason. In some languages that support delegation, like Groovy, Ruby, and JavaScript, you may even change the delegated object on a `Candidate` instance at runtime. Unlike in inheritance, the instances aren't inseparable, and that offers greater flexibility. The `Caller` sends its call to the `Candidate`, which then forwards the call, as appropriate, to the delegate.

Choose wisely

With these two distinctive design choices being available, programmers have to choose wisely between the two. Use inheritance to specialize the implementation of a class and to substitute an instance of one class in place of another—that is, to design the kind-of relationship, like in `Dog` is a kind of `Animal`. To merely reuse an object in the implementation of another, use delegation, like a `Manager` has an `Assistant`, to do part of the work.

If delegation is the right choice for your design, you'll have to write a lot of duplicated code in languages like Java. That leads to bloated code that can be hard to maintain. Kotlin uses a better, declarative approach to delegation—you concisely convey your intent, and the compiler runs off to generate the necessary

code.

In the next lesson, we'll cover how to design delegates in Kotlin.