Introduction to Class Hierarchies and Inheritance

Classes are more like social creatures than hermits. Classes relate to and build on top of the abstractions defined in other classes. To build complex applications, it should be easy to create hierarchies of abstractions. Kotlin does that well—you can create interfaces, define nested and inner classes, and also use inheritance.

Being a statically typed language, Kotlin promotes design by contract, where interfaces serve as specifications and classes as implementors of those contracts. You can also reuse implementations, in addition to specifications, by creating abstract classes.

From the safety point of view, classes are `final` by default and you have to explicitly annotate them as `open` to serve as a base class. Further, Kotlin doesn't work in a binary state of inheritable vs. non-inheritable. You can define classes as `sealed` and thus state which specific classes may extend from those. This gives you the capability to model closed sets of classes, to create what are called algebraic data types in type theory—a powerful idea and something that's not currently possible in Java.

We'll start this chapter with creating interfaces and abstract classes. Then we'll look at creating nested and inner classes—good design options when two classes are closely related to each other. We'll then follow that with how to use inheritance if we want to substitute the objects of one class where instances of another are expected. Then you'll learn how to restrict subclasses using `sealed classes. Finally, we'll look at enums, which are a way to create classes that work together to represent multiple values of a single abstraction.
