Sealed Classes

Sealed Classes

Sealed classes are used for representing restricted class hierarchies

Sealed Classes

Sealed classes are used for representing restricted class hierarchies, when a value can have one of the types from a limited set, but cannot have any other type. They are, in a sense, an extension of enum classes: the set of values for an enum type is also restricted, but each enum constant exists only as a single instance, whereas a subclass of a sealed class can have multiple instances which can contain state.

To declare a sealed class, you put the `sealed` modifier before the name of the class. A sealed class can have subclasses, but all of them must be declared in the same file as the sealed class itself. (Before Kotlin 1.1, the rules were even more strict: classes had to be nested inside the declaration of the sealed class).

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()
```

A sealed class is abstract by itself, it cannot be instantiated directly and can have `abstract` members.

Sealed classes are not allowed to have non-`private` constructors (their constructors are `private` by default).

Note that classes which extend subclasses of a sealed class (indirect inheritors) can be placed anywhere, not necessarily in the same file.

The key benefit of using sealed classes comes into play when you use them in a when expression. If it's possible to verify that the statement covers all cases, you don't need to add an `else` clause to the statement. However, this works only if you use `when` as an expression (using the result) and not as a statement.

```
fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```