Value types, pure functions, constant expressions

Alexander Kuklev[1,2] ‹a@kuklev.com›

[1]Radboud University Nijmegen, Software Science
[2]JetBrains Research

In Kotlin, enums, primitive types (such as `Int`) including `String`, and value classes over them carry value semantics. OpenJDK Project Valhalla introduces value objects that emulate value semantics for non-primitive data. We propose a Kotlin extension properly introducing such objects into the language. Value objects will be required to be hereditarily immutable, self-contained, and devoid of identity besides equality. All fields must be of value types as well. The requirement to be self-contained also imposes restrictions on member functions: they should not be allowed to capture or access any external non-value objects, except those they acquire as parameters and context parameters (receivers). This condition is also known as relative purity, because if such functions only receive value-typed parameters, they are indeed pure.[1]

By enforcing function purity on the type level, we can prevent odd behaviour and eliminate possible vulnerabilities in cases where higher-order functions such as `sortWith(comparator)` rely on the purity of their arguments. We also may optimize performance since pure functions are safe to compute ahead of time, postpone, re-execute, or omit if their result is ignored.

Value types are inherently serializable, and self-contained functions always can be executed at compile time. This way, we can allow constants of non-primitive types and constant expressions containing arbitrary self-contained functions..[2]

1 Records and algebraic data types

Let us use `data` keyword without `class` to define records, i.e. data classes with value semantics. All their fields have to be immutable (hence, `val` keyword can be omitted) and have value types. As opposed to usual data classes, records can be extended, but only by subrecords declared inside their definitions just as in enums. This way we get algebraic data types:

```
data AdtList<T> {

  NonEmptyAdtList<T>(head: T,
                     tail: AdtList<T>) : AdtList<T>
  EmptyAdtList : AdtList<Any?>()
}
```

A record with subclasses, no fields, or all fields given a preset value is a data object with value semantics. This way, algebraic data types effectively subsume enums.

All in all, value types include primitive types, records, algebraic data types and enums, value classes wrapping value types, and immutable arrays of value types.

All member functions of value types must be relatively pure.

2 Self-contained functions

Self-contained functions `f: pure (Xs)→ Y` are functions that are only allowed to invoke, access, or capture external entities that are self-contained constants. They are only pure in the usual sense if they only have value-typed parameters (including context parameters/receiver). We should also allow using the modifier `pure` with function interfaces at the call site:

```
fun <T> Array<T>.sortWith(comparator: pure Comparator<T>)
```

Sometimes it is desirable to allow a fixed list of exceptions to self-containedness:

```
fun <T> Array<T>.sortWith(comparator: pure(Logger::trace) Comparator<T>)
```

[1]Purity is compatible with runtime exceptions and encapsulated mutability (local `vars`).
[2]Partial support for these features is currently being implemented by Ivan Kylchik and Florian Freitag