Startup and dependency injection

Alexander Kuklev[1,2] ‹a@kuklev.com›

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

Presently, the entry point of a Kotlin application is always a static function named `main()`. We propose introducing an annotation that allows a class constructor to serve as an entry point:

```
class OurApp : Application {
  @Main constructor(varargs args: String) { … }
  // also allow keys and flags in addition to positional arguments, see below
}
```

At startup, applications typically couple external components, naïvely implemented as singletons:

```
object Database : DbConnection("jdbc:mysql://user:pass@localhost:3306/ourApp")
```

Global singletons are visible to each other and initialized when first used, so no dependency injection is required. While this approach is unbeatably concise, it has serious drawbacks:

- parameters must be known in advance, ruling out config files and command-line arguments;
- tight coupling hinders unit testing and reusability;
- initialization happens in an uncontrolled manner.

To address these issues, let us introduce a syntax to state component dependencies explicitly:

```
// Declaration syntax: init Component(Dependencies) { initializer }
init ConfigPath() = "./etc/config.yaml"

init Config(ConfigPath) = Yaml.fromFile(ConfigPath)

init Database(Config) = DbConnection(Config.dbConnString)

// Application class with explicit overridable dependencies:
class OurApp(params) init(Config, Database) : Application { … }
```

Under the hood, dependencies are additional constructor parameters of `OurApp` with default values which allows overriding them if necessary. Their initialization precedes superclass constructors. Explicit dependencies (`Config`, `Database`) are `protected val` arguments accessible as fields of the class after its construction, while implicit transitive dependencies (`ConfigPath`) are not. Yet they are initialized in the same phase and can also be overridden:

```
class UnitTests {
  val app = OurApp(params,
                   ConfigPath = "tests/config.yaml",
                   Database = MockDb)
  ... tests
}
```

Very often, default components can be overridden using command-line arguments, so let us also introduce syntax for optional overriding:

```
@Main class OurApp(@Flag("-V", "verbose mode") val verbose: Boolean,
                   @Key("-C", "config file path") val path: String?)
      init(ConfigPath = path if path != null,
           Config, Database) : Application {

  ...
}
```