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 the main function. We propose introducing `@main` annotation that allows using a class constructor as the entry point:

```
class MyApp {
  @main constructor(args : Array<String>) {
    ...
  }
}
```

Application startup often requires initialisation of external services and components configurable via command-line arguments. With our proposals and kotlin-argparser it looks as neat as this:

```
class OurApp(args : Args)
      init(override ConfigPath = args.configPath if args.configPath != null,
           Config, Database) : Application {

    @main constructor(args : Array<String>) : this(parseArgs<Args>(args)) { … }

    private class Args(p : ArgsParser) {
      val configPath by p.storing("-C", "--config", "config file path")
    }
    ...
}
```

The most straightforward way is to frame external services and components as global 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.

These issues are solved by introducing the application class which initialises necessary singletons in the right order according to their dependencies (which may include simultaneous initialisation) and interconnects them. We believe that the language should provide a specialised syntax to make this approach a drop-in replacement for the naïve one with no syntactic penalty, e.g.

```
init ConfigPath() = "./etc/config.yaml"       // init Foo(Dependencies) {initialize}

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

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

class OurApp(params) init(Config, Database) { … }    // class Bar init(Dependencies)
```

Under the hood, dependency initialization works via default constructor's default arguments, so it precedes superclass constructors. Explicit dependencies (`Config`, `Database`) are available as protected immutable properties of `OurApp`. Implicit dependencies (`ConfigPath`) are not accessible by name, but are initialized at the same time and can be overridden exactly as the explicit ones:

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