Fantom

Programming language for JVM, CLR, and JS

2012, Kamil Toman

http://fantomery.org/katox/fantom-en.pdf

What Is Fantom

- Language for multiple platforms
- Object oriented
- Functional
- Balance of static and dynamic typing
- Well-known "c-like" syntax

- Opensource (AFL 3.0)

Productivity

- Literals
- Type inference
- Functions
- Closures
- Mixins
- Integration (FFI)
- DSL support

Literals

- Lists
  [,]
  [10, 20, 30]
  ["a", "b", ["c", "d"], "e"]
- Maps
  [:]
  ["x" : 10.0d, "y" : 3.14f]
  [10 : "x", 20 : ["p", "q", "r"]]
- Types (introspection)
  sys::Str#, Str#
- Slots (introspection)
  Str#plus, Int#plusDecimal, Uuid#fromStr

Literals (2)

- Uri
  `http://fantom.org`
  `http://myserver.com/edit?n=Ron&s=Smith`
  `/home/fantom/checkitout.fan`
- Duration
  1ns, 1ms, 1sec, 1min, 1hr, 1day
- Range
  [10..20, 30..<40, -3..-1]

Type Inference

- u := Uuid()
  // u.typeof -> sys::Uuid
- s := "some string"
  // s.typeof -> sys::Str
- list := [10, 20, 30]
  // list.typeof -> sys::Int[]
  listObj := ["a", 1]
  // listObj.typeof -> sys::Obj[]
- map := ["x" : 10, "y" : 3]
  // map.typeof -> [sys::Str : sys::Int]
- mapNum := ["x" : 10.0d, "y" : 3.14f]
  // mapNum.typeof -> [sys::Str : sys::Num]

Functions and Closures

- Functions - signature |A a, B b, ..., H h -> R|
  ```
  double := |Int a -> Int| { 2 * a }
  v := |->| {}
  ```
- Methods - just functions wrappers
  ```
  |Int a, Int b ->Int| plus := Int#plus.func
  ```
- Closures - expressions that return functions
  ```
  Str prefix := ""
  print := |Obj o -> Str| { prefix + o.toStr }
  ```
- Bind - bind parameters of existing functions
  ```
  op := |Method m, Int a, Int b| { m.callOn(a, [b]) }
  mul := opFunc.bind([Int#mult])          // |Int, Int->Int|
  plus2 := opFunc.bind([Int#plus, 2])     // |Int -> Int|
  ```

Mixins

```
mixin HasColor {
  abstract Str color
  virtual Void sayIt() { echo ("My favourite $color") } }

mixin HasAge {
  abstract Date produced
  virtual Bool isVeteran() { return produced < Date.fromStr("1919-01-01") } }

class Car : HasColor, HasAge {
  override Str color := "evil grey"
  override Date produced := Date.today
  override Void sayIt()   {  echo("My ${isVeteran ? "veteran" : color} car
produced ${produced.year}.")   } }

Car().sayIt     // -> My evil grey car produced 2012.
```

Java FFI

- Java
  - package => Fantom pod
  - class => Fantom class
  - interface => Fantom mixin
  - field => Fantom field
  - method => Fantom method

```
using [java] javax.swing
using [java] java.util::Map$Entry as Entry
f := JFrame(...)
```

- There are some limitations of Fantom FFI. What's not supported
  - overloaded methods, primitive multidimensional arrays, > 1 level deep inheritance from java classes

Javascript FFI

- source code Fantom -> Js generation (no bytecode)

```
@Js
class GonnaBeJs
{
  Void sayHi() { Win.cur.alert("Hello!") }
}
```

- native peer Js -> Fantom

```
 // Fantom
class Foo  {
    native Str? f
}

 // Javascript
fan.mypod.FooPeer.prototype.m_f = "";
fan.mypod.FooPeer.prototype.f   = function(t) { return this.m_f; }
fan.mypod.FooPeer.prototype.f$  = function(t, v) { this.m_f = v; }
```

DSL Support

- Ability to integrate custom language or AST modifications of Fantom code
- Syntax - DslType <|...|>

```
echo(Str<|A towel, it says, is about the most massively useful thing an interstellar hitchhiker can have.|>)

Regex<|^[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}$|>

@Select
User getUser(Int id){
  one(sql<|
    select u.userid, u.name
    from User u,Company c
    where u.id = #{id} and u.companid  = c.id and c.isdeleted = 0
   |>)
}
```

Practical Shortcuts

- Dynamic typing
  ```
  obj->foo   // obj.trap("foo", [,])
  ```
- Implicit upcast
  ```
  Derived derived := base
  ```
- Optional return for simple expressions
  ```
  files.sort |a, b| { a.modified <=> b.modified }
  ```
- isnot keyword
  ```
  alwaysFalse := "a" isnot Str
  ```
- Optional parenthesis for functions of arity 0
  ```
  yes := "yes".capitalize
  ```

Practical Shortcuts (2)

- Optional function calls
  tel := contact?.phone?.mobile
- Elvis operator
  x ?: def // x == null ? def : x
- it-blocks
  `file.txt`.toFile.eachLine { echo (it) }
  // `file.txt`.toFile().eachLine(|line| { echo (line) })
- literals for data
  Date.today - 1day // yesterday
- string interpolation
  res.headers["Authorization"]="Basic " + "$user:$pass".
  toBuf.toBase64

Declarative Style

```
win := Window
    {
      size = Size(300,200)
      Label {
          text = "Hello world"
          halign=Halign.center },
    }.open


createTable("user") {
     createColumn("user_id", integer) { autoIncrement; primaryKey }
     createColumn("login", varchar(64)) { notNull; unique }
     createColumn("pwd", varchar(48)) { comment("Password hash"); notNull }
     createColumn("role_id", integer) { notNull }
     index("idx_login", ["login"])
     foreignKey("fk_role", ["role_id"], references("role", ["role_id"]))
}
```

Declarative Style (2)

```
class Person {
  Str? name
  Str[]? emails
}
phoneBook := [
    Person
    {
      name = "Fantom";  emails = [ "fantom@opera.org", "fantom@gmail.com" ]
    },
    Person
    {
      name = "Nobody";  emails = [ "nobody@nowhere.org", "noob@gmail.com" ]
    }
]
```

Functional Style

```
["ox", "cat", "deer", "whale"].map { it.size }  // -> [2, 3, 4, 5]
[2,3,4].all { it > 1 }                  // -> true
[2,3,4].any { it == 4 }                 // -> true

["hello", 25, 3.14d, Time.now].findType(Num#) // -> [25, 3.14]
[1, 1, 2, 3, 2, 1, 1].findAll |i| { i.isEven }    // -> [2, 2]

[1, 1, 2, 3, 2, 1, 1].reduce([:]) |[Int:Int]r, Int v->[Int:Int]| { return r[v]=r.get(v,0)+1 }
// -> [1:4, 2:2, 3:1]

phoneBook.findAll | Person p -> Bool | {  p.name == "Fantom" }
            .map | Person p -> Str[] | { p.emails }
            .flatten
            .exclude | Str email -> Bool | { s.endsWith("gmail.com") }
            .each { echo (it) }
// -> fantom@opera.org
```

Safety

- Null & Not-Null types
- Const classes and fields
- Guaranteed functions/closures with no mutations (thread safe)
- Actor system (no shared state)
  - Unsafe wrapper - break your rules whenever you want to save some time and shoot yourself into foot

Null and Not-Null types

- null assignment must be explicitly allowed in the code
- Default for types is **not** nullable version
- Automatic conversion
  - apparent bugs -> compile error
  - could work -> (maybe) runtime error

Null and Not-Null types (2)

```
class Nullable {                                    // © Xored, Inc.
    Void main() {
        Int? nullable := null
        Int nonNullable := 0
        nonNullable = nullable                      // runtime err
        nonNullable = null                          // compile err
        do3(null)                                   // compile err
        do3(nullable)                               // runtime err
    }
    Int do1(Int? arg) { null }      // compile err
    Int do2(Int? arg) { arg }       // runtime err
    Int? do3(Int arg) { arg }
}
```

Const

- Const classes guarantee their internal state won't change
- Const fields must be initialized on object construction
- Const fields must be const types or their values must be converted by calling toImmutable for List, Map, Func

Const

```
class NonConst {                    // © Xored, Inc.
        const Int constField
        Int nonConstField
        Void mutate()  {
                constField = 4      // compile err
                nonConstField = 4   // ok
        }
}
class Const {
        new make(Int i, Str[] list) {
                // implicit call of list.toImmutable
                this.list = list
        }
        const Str[] list
}
```

Actor System

- Concurrency is not handled by thread but by Actors
- Actor is a const class extending concurrent::Actor
- Actors exchange immutable messages, i.e.
  a. serialized content
  b. constant content (allows to pass by reference)
- Actors typically contain private data, which are mutable (thread local analogy)

```
// asynchronous incrementation of number send (blocking send operation)
actor:=Actor(ActorPool()) | Int msg -> Int | { msg + 1 }
5.times { echo(actor.send(it).get) }
```

Tooling and Build System

- Out-of-the-box build system using Fantom
  - handles standard setup with no additional code
  - easy to understand (simple implementation)
  - module dependencies and version checking
- Bundled simple testing framework **fant**
  - fits into default build structure
  - pretty standard lifecycle (junit like)
  - works also for javascript targeted code

```
using build
class Build : BuildPod {
  new make()   {
    podName = "mongo";  summary = "Interface to MongoDB (http://www.mongodb.com)"
    depends = ["sys 1.0", "inet 1.0", "concurrent 1.0"]
    srcDirs = [`test/`, `fan/`, `fan/gridfs/`, `fan/bson/`]
    docSrc  = true   } }
```

Modularity - Fantom repo & pods

```
// start all non-abstract MyService services in project CoolProj
using fanr
repo:=Repo.makeForUri(`http://my-fantom-repo/`)
pods:=repo.query(Str<|"*proj.name=="CoolProj"|>)
pods.each |pod|
{
  echo("$p.name$p.version$p.depends")
  pod.types.each |type|
  {
    if (type.fits(MyService#) && !type.isAbstract)
      type.make->start
  }
}
```

References

- Open source & docs
  - http://fantom.org/
  - http://www.talesframework.org/
  - http://www.fanzy.net/
  - http://langref.org/
  - http://rosettacode.org/wiki/Category:Fantom
- Commercial products
  - http://skyfoundry.com/skyspark/
  - http://www.xored.com/products/f4/
  - http://www.kloudo.com/
  - http://www.cull.io/