A Short Introduction to Fantom

Thomas J. W. Clarke

5 May 2014

Contents

1 Introduction **5**
- 1.1 Getting Started . . . 5
- 1.2 Language Features . . . 5
- 1.3 Syntactic Differences from Java . . . 7

2 Fantom Development Tools **8**
- 2.1 Prerequisites . . . 8
- 2.2 Standard Build System . . . 8
- 2.3 F4 Eclipse IDE . . . 10

3 Fantom Types **11**
- 3.1 Type Hierarchy and Fits . . . 11
  - 3.1.1 Nullability . . . 12
  - 3.1.2 Mixins and Inheritance . . . 12
- 3.2 Funcs and over or under binding of parameters . . . 13
  - 3.2.1 Func.bind . . . 13
  - 3.2.2 Differences from Java in Class definitions . . . 13
- 3.3 Type Inference: Implicit and Explicit Type Casting . . . 14
- 3.4 Generic Typing for List, Map . . . 14
- 3.5 Static vs Dynamic Type Checking for List, Map . . . 14
  - 3.5.1 Holes in the static type system . . . 15
- 3.6 Language Support for Dynamic Typing . . . 15

4 Functional Programming and Closures **17**
- 4.1 Functions as First-Class Objects . . . 17
- 4.2 Closures . . . 18
  - 4.2.1 Unbound variables in closures . . . 19
  - 4.2.2 Mutability of closures . . . 19
- 4.3 Functional Programming . . . 19
  - 4.3.1 Map and reduce . . . 19
- 4.4 Summary . . . 19

5 It-Blocks and Declarative Programming **20**
- 5.1 It-Blocks . . . 20
  - 5.1.1 Slot lookup and assignment in it-blocks . . . 20
  - 5.1.2 It-Add Statements (AKA comma operator) . . . 21
- 5.2 With-blocks . . . 21
- 5.3 Declarative Programming Examples . . . 22
- 5.4 Summary . . . 23

6 Concurrency **24**
6.1 Example 1 - Concurrent Computation with Results Collected 24
6.2 Example 2 - Output from a long-running task 27
6.3 Example 3 - Interaction with a GUI 29
6.4 Actors In Detail 32
6.4.1 Exceptions in Actor Code 32

7 Conclusions **33**

Nomenclature

abstract field	field of abstract class with type but without data: this must be provided by a subclass in order for (subclass) instances to be created.
abstract method	method of abstract class with signature but without implementation: this must be provided by a subclass in order for(subclass) instances to be created.
anonymous	without a name (in contrast to a method that will always have a method name)
const field	field with value that is a constant (can be created dynamically in constructor).
cruft	Unnecessary words within source code that do not affect or clarify meaning and bulk out the code describing the algorithm
dynamic typing	Data types are held as a property associated with data values and checked for consistency only when these are used at run-time. No types are decalared in the porgram source.
generic types	AKA polymorphic types. Type annotations that incorporate wildcards allowing arbitrary types to be applied subject to constraints, e.g. types in a given wildcard used within one type signature must match for any specific instance of the signature.
immutable	an expression that cannot be changed. Eitehr const, or result of toImmutable method
JVM	Java Virtual Machine: interpreter and (typically) just in time compiler than can run Java jar files.
List	Type representing orders lists of items implemented as variable-length arrays, accessed by index.
Map	Type representing associated maps (AKA hashes, dictionaries) in which values are stored with an associated key and can be looked up from the key.
mixin	set of abstract fields and (possibly concrete) methods that specify an interface to be satisfied by a subclass. Similar to java interface but more general
narrowing cast	casts that convert a type into a subtype: e.g Num to Int
non-nullable	cannot contain null as a value
nullable	can contain null as a value
Obj	The root of the type hierarchy, can be any type
OO	Object oriented - a language which supports classes, instances of which combine data and program.
pod	Fantom code module. Pods are the unit of code compilation and dependency, and have version numbers as well as names.
signature	Type specification for function specifying parameter and return types.
slot	named field or method associated with a class and accessed via name from objects of the class using the . operator

static strongly typed	Types are specified in source code with variable and function declarations and checked by the compiler. All data has a type and all types must be correct.
string interpolation	Escaped sub-sequences within a string that are replaced by corresponding values

Chapter 1

Introduction

This tutorial should be read together with the excellent standard Fantom documentation. It is written for someone who has a fair knowledge of at least one C family language (e.g. Java or C++) and wants to read and write Fantom. It will focus on features of Fantom that are unusual and can be difficult for newcomers to understand. Mostly these features represent improvements of Fantom over many other languages, or at least design trade-offs that can be argued to be in a sweet spot. Fantom also has improvements that are obvious and intuitive while keeping most of the Java syntax and operators so that code looks familiar.

1.1 Getting Started

To get started:

1. Download and setup tools following the standard documentation get started guidance, or for a little more help: 2.1 and then 2.3
2. Read Section 1.3 that summarises the syntactic issues you need to know when writing code.
3. You should read the introduction to Chapter 3 right away in order to understand the highly unusual way in which static types are used in Fantom.

The other Chapters in this Tutorial are for when you want to understand aspects of the language semantics. It can be read consecutively or dipped into when you realise there is something you do not understand. When learning Fantom you will find the standard documentation, and particularly the documented methods of the standard library s, best for specific problems. This tutorial brings together material that is otherwise scattered throughout the standard documentation and in some cases nonexistent, and creates a language-aspect focussed overview.

1.2 Language Features

There follows a description of some of the notable characteristics of Fantom.

API A complete set of standard libraries rewritten from Java with the aim of providing cross-platform portability and a simple clean interface for the programmer. Its authors argue that the Fantom libraries, capturing functionality in a relatively small number of large but capable classes, make typical use cases less much burdensome and cleaner. This combines with the use of function parameters whenever this is appropriate to make a significant improvement over Java. Those who have only basic knowledge of Java libraries will find it easier to learn to use Fantom than the corresponding Java. Those very familiar with Java libraries will have some overhead learning new libraries, but will appreciate the improvements. Running on a Fantom can easily access Java native libraries as well, so the resources of the Java programming ecosystem remain available.

Types Both static and can be used in a pragmatic way. The idea is that the static type-checking will still catch nearly all type errors but things are done to improve readability over other language such as Java:

- Static types are deliberately kept simple. More complex type systems (Scala) can provide better static checking but at the cost of additional linguistic complexity. Fantom falls back gracefully to strong dynamic typing for anything too complex for the static type system.
- Wherever possible (local variables and closures) type inference is allowed and type-related cluttering code may be omitted. There is usually no need to give the types of local variables or closures because this is obvious from context: where not types can be written explicitly and will be checked.
- Constructs to reflect and do dynamic dispatch allow dynamic typing to be used where necessary. Dynamic checks will be added as needed to ensure safety.
- The static type system incorporates information about whether `null` is allowed in a type and tracks this. This reduces the typical Java or C++ null pointer errors that are so ubiquitous, and also makes finding them much simpler.
- Static typing incorporates implicit casting where this might be correct, thus eliminating the need for pervasive casting. Implicit casting puts some of the type checking back into the run-time, but since most type errors come from things that have no values in common (e.g. mixing `Float` and `Str`), it does not much change the ability of static typing to provide compile-time checking.

Closures Fantom regards functions as first class objects and the standard libraries have been written from ground up to use closures as parameters where this simplifies things. Other languages use closures (notably Ruby) but Fantom's closures are (sometimes) statically typed and both highly expressive, and supported by special syntax, a powerful tool.

It-blocks Idiomatic Fantom uses closures written in a compact syntax - it-blocks - that is intuitive: but only when you understand it! This is probably the single most difficult aspect of Fantom for newcomers. It is well worth the small effort to master this because code written using it-blocks is compact, clean, and expressive.

Actors Fantom has a model of concurrency based on message-passing using actors. An actor models possibly fine-grained concurrency with a message-processing function that is called repeatedly, once for each input message, and generates each time a reply message. Actors can have internal state, though this is not encouraged.

Mutability In order to support safe concurrent computation Fantom provides built-in linguistic support for whether or not expressions are mutable. An *immutable* expression can safely be passed between concurrent threads because order of execution cannot change its value. Immutable expressions include constants, as expected, but also dynamic structures such as Lists and Maps that have been *made immutable*. The compiler can track immutability within the static type system.

Syntax One of the most important determinants of usability in a language is syntax that makes programs compact and easy to read. The trade-off here is between simplicity and regularity (is it easy to learn) and expressiveness. Small changes in one part of the language can have far-reaching effects. For example, in Fantom, much of the syntactic niceness of the type system, allowing optional type inference, comes from the device of distinguishing type names with initial capital letters.

The Chapters 3-6 cover some of these areas highlighting things that are surprising or more complex. For more complete details refer to the standard documentation, which also has a good write-up of what is different and better about Fantom when compared with Java.

Fantom is positioned as a highly usable and above all productive language. Much as Python it's design aims to make the programmer more productive, with constructs to make typical code compact and highly readable. Python is irredeemably wedded to dynamic types and the lack of proper static typing, Fantom sees the advantages of mandatory static typing both in documenting programs and catching as many errors as possible errors at compile time.

Unlike many post-Java languages Fantom comes with its own complete set of standard libraries, rewritten for simplicity and much easier to use than the many-layered Java libraries. The language can run on a JVM and interface to Java where this is needed.

Although Fantom typically runs on a JVM it is highly portable and Fantom Pods can be run under JavaScript (for web pages) and .NET. That makes Fantom a good candidate for web development where server-side and client-side code can be written in the same language.

1.3 Syntactic Differences from Java

Much of Fantom syntax will be familiar to anyone who knows Java. The list below shows some of the most obvious syntactic differences:

- Names use *casing* to determine semantics. Types have initial upper-case. Everything else (methods, variables) must have initial lower-case. Camelcasing is conventionally used instead of underscores in names: `aLongVariableName`, `ALongTypeName`.
- `;` is optional at the end of lines.
- `()` is optional in method invocation, and where the last parameter of a method call is a closure this can be pulled out of the methods call brackets - which therefore often may be omitted - and written after the method name.
- Types are optional for local variables.
- Special syntax exists for Lists `[1,2,3]` and Maps `[1:1,2:4,3:9]`.
- `:=` must be used to declare and initialise a local variable.
- `null` is a value allowed only in nullable types (`List?`, `MyType?`) and not in types (`List`, `MyType`).
- is introduced by `$` with `{ }` braces optionally to allow more complex interpolated expressions `"The square of $n is ${n*n}."`. Triple-quoted strings are allowed and can include unescaped escape characters and newlines.
- s are not needed, and will be inserted implicitly with run-time type checks.
- User-defined are not allowed, `Obj` can be used instead with dynamic typing.
- Many shorthand operators (see 3.6) are available to make writing expressions with or dynamic types more convenient.
- In many contexts `{ }` introduces an closure with (optional) implicit single parameter `it`. See Chapter5.

Chapter 2

Fantom Development Tools

2.1 Prerequisites

Fantom is developed with a set of command line Fantom tools as described in the standard documentation. These require a JVM to run (Java 1.6 or 1.7 as of Fantom 1.0.68). The necessary Java can be downloaded as JRE 1.6 or 1.7 or JDK 6 or 7. Fantom will run with 32 or 64 bit code, and requires a 32 or 64 bit JVM. The links betwen the Fantom tools and the JVM they depend on is normally found automatically but can be configured via an environment variable. All the standard tools work equally well under Windows, linux and OS-X. From a command line `fan -version` provides information about the JVM and fantom compiler currently being used.

The Fantom tools are complete and except for unavoidable JVM dependence they are standalone. GUI code written in Fantom using the fwt Pod has one further dependence, a Java system-dependent swt jar file. This can be downloaded, for a given system, from the Sun.

To summarise, the standard tooling for Fantom requires:

- 32 or 64 bit Fantom standard compile and toolset (from www.fantom.org)
- Java (JDKor JRE) from Java standard downloads, 32 or 64 bit to match. Currently JRE 1.6 or 1.7 runs Fantom 1.0.68.
- For GUI code using fwt pod: Java swt jar file (system-dependent) for Fantom GUI code from Java standard downloads, 32 or 64 bit to match.

This Tutorial should be enough to get started but in case of problems refer to the Fantom standard documentation and if needed the Fantom discussion forum where thread searches will find additional information.

2.2 Standard Build System

Fantom comes as standard with a sophisticated standard build system, contained in Fantom module `build`, in which build instructions are written in Fantom itself. The build process makes executable pods from Fantom and Java native source files. Dependencies are explicitly stated. Figure 2.1 is a subclass of Fantom standard class *BuildPod* that will compile a set of files to make a pod (`.pod`) file that can be run by the Fantom launcher `fan`t. The build engine uses a number of classes managing the build process. Of these the most central is *BuildPod* that accepts targets - tasks to execute - for example *compile* or *clean*. Note that standard functionality like compile is built in as virtual methods of *BuildPod* that can be overridden. As shown in this example, subclasses of *BuildPod* can contain arbitrary Fantom code to customise the build process. Table 2.1 shows how the build module works. The lowest individual compilation unit is a Pod (a fantom module), which can be made from any number of source files, mixing Fantom, Java, and Native code, and which consists when compiled of a single `.pod` file. Dependencies between pods are handled via an explicit set of versioned *depends* clauses. *BuildPod* is the class which orchestrates compilation of a pod, checking dependencies.

A Fantom source file must explicitly reference any pods it uses with a `using` top-level clause as in Figure 2.1. These may be pods in the standard library, or other user-defined pods compiled separately.

Class	Target	Description
BuildPod	compile	Fundamental *compile* target, accepts lists of source files and dependencies, generates a Pod
	clean	Delete all intermediate files
	full	Run *clean*, *compile*, and then *test*
BuildScript	n/a	Runs a script containing instructions with BuildPod targets to run etc

Table 2.1: Build Class Functionality

A simple Fantom project will consist of a single pod that is compiled using a subclass of BuildPod as in Figure 2.1. The compile target will recompile the pod after first deleting the previously compiled pod file if this exists. This is so that F4 will not run an older version of a pod on new compilation failure.

This tutorial does not deal with running Fantom other than under a JVM, however the build system and standard tools allow this. Currently Fantom under JavaScript is very well supported, and actively used to write web applications, Fantom under .Net less well so. See the standard documentation for further information.

Need to describe how a typical development cycle with tests works.

```
using build
class Build : build::BuildPod
{
  new make()
  {
    podName = "projadmgui"
    summary = ""
    srcDirs = [`fan/`]
    depends = ["sys 1.0", "gfx 1.0+", "fwt 1.0+", "concurrent 1.0+",
               "fluxText 1.0+", "sql 1.0+"]
  }

  @Target{help = "Delete target and then recompile"}
  override Void compile()
  {
    outPodDir.plusName("${podName}.pod").toFile.delete
    super.compile
  }
}
```

Figure 2.1: Build for F4 IDE

2.3 F4 Eclipse IDE

F4 is an IDE for Fantom based on Eclipse written by xored (`www.xored.com`). It is the most capable of the IDEs currently existing for Fantom. The current distribution (1.0.1) needs to be updated with the latest Fantom interpreter after installation. The F4 distribution is a self-contained directory tree of files that need not be installed. You can thus have different Eclipse distributions for different languages coexisting with no interference. You may want to set up `.fan` extensions to open with your F4 distribution.

A set of fixes that works (at least on Windows, and I'd expect on other platforms) is:

1. Download latest Fantom distribution. Put Fantom directory somewhere, the default `C:/fantom` is fine on Windows.
2. Download the latest F4 directory tree release from github or xored. Unzip the code to some suitable directory.
3. Start F4.
4. Window→Preferences→Fantom/Interpreters. Press *search* to find your Fantom distribution. Tick this (instead of the in-built distribution)
5. Add override to method `compile` in the `build.fan` file for any created project as per Figure 2.1 (this is a workaround, to make build and run have expected semantics)

```
@Target{help = "Delete target and then recompile"}
  override Void compile()
  {
    outPodDir.plusName("${podName}.pod").toFile.delete
    super.compile
  }
```

This section is incomplete, possibly wrong, and needs updating. Need to describe how to use F4 with tests.

Chapter 3

Fantom Types

Fantom's mixture of static and dynamic typing is mostly intuitive. Typically the language is used as a statically typed language, with the burden of writing types reduced because normally local variables can infer their static type. The static type system is deliberately simple, with limited generics, to allow methods on s, s to have useful types, but without user-defined generic types.

The key matter to understand is how dynamic typing and static typing interact. In most static types languages the compiler guarantees that values at run-time *must lie within the static type*. Anything that could lie outside will be a compile-time error. Fantom changes the burden of proof. The compiler guarantees only that values *may lie within the static type*, and that if they do not the result will be caught by a run-time check. Static type errors come from assignments that could *never* work.

This is an interesting approach to the design trade-offs between static and dynamic types languages. It provides the code compactness and flexibility of a dynamic system while keeping much of the compile-time error-checking of a static type system. Where type errors are not discovered at compile-time the extra checks inserted whenever types are narrowed means that they will be caught at run-time near where they happen.

Another consequence of this strategy is that a relatively simple type system will serve, since it is always possible, with little cruft, to use a more general type when a precise static type does not fit. The programmer uses the static type system as an approximation to the real type, and the language makes it easy to approximate more where type system is too weak to capture the data's precise type.

In practice most type errors are caught statically in Fantom. Philosophically, if you view a type system as a set of contracts enforced at compile-time, Fantom is weaker than other static languages but stronger than dynamic languages. Paradoxically that can result in more accurate (and therefore more useful) type checking because for example nullability is built into the type system at no inconvenience to the programmer. Types are required for method s and fields so static types serve to document code: in the case of nullability the programmer must think whether a object is allowed to be null. Programmers can always choose to bypass static typing by using as the type everywhere, but this is not encouraged by the language, and typical code will have informative type annotations.

In summary:

- The language is as compact as a dynamic language because most type annotations and casts are unnecessary.
- Static checking will catch most errors at compile-time.
- The remaining type errors will be caught at run-time in the method in which they are made, because run-time checks will enforce typing at method boundaries.

3.1 Type Hierarchy and Fits

Fantom has as the basis for its type system a conventional object oriented type hierarchy with single inheritance in which every simply typed non-nullable expression is typed as a narrowing of *Obj*. Every type (except Obj) has a unique parent type, which it extends, for example by adding fields or methods. If type A can in all circumstances replace type B (equivalently, if it adds new contracts and satisfies all the old contracts of B type) we say that A fits B.

Given two types A & B we have either A fits B, B fits A, or neither type fits the other. In the latter case the two are incompatible. If A fits B and B fits A the two types are identical. If A is the parent of B then B fits A.

Fits as usual is a transitive relationship, in fact it is a partial order:

A fits B AND B fits $C \Rightarrow A$ fits C.

A fits B AND B fits $A \Rightarrow A = B$.

Value types (*Int*, *Float*, etc) are optimised by the compiler but otherwise part of the type hierarchy.

The simple tree-type hierarchy that Fantom shares with all OO languages is extended in two ways specific to Fantom.

3.1.1 Nullability

Any type A must be either nullable or non-nullable. Nullable types can have `null` as a value, non-nullable types cannot do so. The nullable equivalent of non-nullable A is written $A?$ values of type `A?` can include `null`.

Fits extends to nullability in the obvious way, non-nullable is a narrowing of nullable:

1. A fits $B \Rightarrow A$ *fits* $B?$
2. A fits $B \Rightarrow A?$ *fits* $B?$
3. $A?$ never fits B

The motivation for putting nullability into the static type system is that this provides some static protection against null pointer errors, which are otherwise, at run-time, unpleasant to debug. Note also that `null`, empty lists and maps, are distinct things. Note also that *Void* is a special type that has no return value (not even `null`). *Void?* does not exist.

Fantom will automatically insert run-time checks where a nullable type is used in a non-nullable context. Where a known `null` value is explicitly used is a non-nullable context there will be a static type error.

3.1.2 Mixins and Inheritance

Mixins are Fantom's equivalent of Java's interfaces - but they are more powerful. A is a variety of type which is not designed to be used stand alone. Instead a mixin packages a group of s together to be inherited into a class (or another mixin).

Mixins are similar to interfaces in Java or C#, but much more flexible. A Java or C# interface is purely a type definition of s, it can't actually include any behavior itself. Fantom mixins can declare concrete methods which provide a lot more power.

You can't create instances of a mixin - they are an abstract type designed to provide reuse when inherited into classes. Mixins also can't store state - although they can contain s to define a type contract which requires a field signature.

1. Mixins can themselves inherit zero or more mixins.
2. Mixin inheritance is transitive on both mixins and types.
3. If *A'* inherits set A_m of mixins, and *B'* inherits set B_m of mixins, then A' fits B if this is true ignoring mixins, and $A_m \supseteq B_m$

Mixin inheritance is restricted. From the standard documentation:

The inheritance rules listed above define which slots get inherited into a sub-type's slot name-space. Remember that a type's slots are keyed only by name, so under no circumstances can a type have two different slots with the same name. Because of this axiom, there are cases which prevent creating a subtype from conflicting super types:

- Two types with static methods of the same name can't be combined into a subtype
- Two types with s (either instance or static) of the same name can't be combined into a subtype

- Two types with instance slots of the same name and different signatures can't be combined into a subtype Two types with instance slots of the same name and same signature can be combined provided the following holds true:
  - One is concrete and the other is abstract
  - Both are virtual and the subtype overrides to provide unambiguous definition

Using the rules above, Fantom avoids the diamond inheritance problem. First mixins can't declare concrete fields, which mean they never store state. Second any ambiguity that arises from diamond inheritance or otherwise requires the subclass to explicitly disambiguate (or if the inherited slots are not virtual, then the subtype simply cannot be created).

3.2 Funcs and over or under binding of parameters

Functions in Java have zero or more parameters, each with a well-defined type. They have a return type that may be Void, indicating no return value.

Functions are typically written in methods, in which case the parameter and return types must be specified. Functions may also be written as closures, see Chapter 4, in which case the types of parameters and/or return value need not be specified if they can be inferred from context.

In a function call, over-binding of parameters is allowed, the extra parameters are ignored. Under-binding of parameters is allowed if all the parameters not bound have default values.

```
Int add(Int a, Int b)
{
   return a+b
}

Void main
{
  echo(add(1,2,5)) // this prints '3'
}
```

This is particularly useful in the API where for example the `Map.map` method:

```
Obj:Obj? map(|V,K->Obj?| c)
```

Takes a function `c` that is given two parameters, the mapped item's `value` and `key`. If the key is not required a single parameter function may be used for this that accepts only the value.

```
x := [1,2,3]
y := x.map( |Int x, Int y->Int| { return x*x})
z := x.map( |Int x ->Int| { return x*x})
w := x.map {it*it}
```

The expressions assigned to `x,y,z` are all equivalent.

3.2.1 Func.bind

Functions supplied with a smaller than required number of parameters can be repackaged as closures callable by supplying the extra parameters using `Func.bind` and giving the bound parameters as a list. Using `add` as above:

```
|Int->Int| plusone := add.bind([1])
```

3.2.2 Differences from Java in Class definitions

The main differences between Java and Fantom classes are listed here: this is not intended to be a complete language tutorial, for details please refer to the standard documentation.

- Static fields or methods must be const, and use both keywords
- methods and fields default to public scope
- Implicit virtual getters and setters are declared for every field and can be used by using or assigning to the field

3.3 Type Inference: Implicit and Explicit Type Casting

In Fantom static type checking must succeed or compilation will fail. Success means that every type in the program *could fit* the type required by its context. Note the difference from most other static type systems, which guarantee that every type *will fit* its context and therefore run-time type errors cannot happen.

Therefore in Fantom whenever a type is implicitly narrowed by the compiler a run-time check is inserted that will fail if the value is of the wrong type.

Because methods must be typed this localises things like null error problems to the method in which they occur. Think of the static types as being a shorthand for inserting dynamic type checks - *but only where this is needed because the inferred type could result in a bad value.*

This is a pragmatic approach that in practice catches many type errors at compile-time, and also provides documentation of programmer intent, without burdening the code with continual type-casts or the type system with very complex types.

Type widening or narrowing will be implemented implicitly when needed by the compiler. For example:

```
Obj test1( Int x)
{
   return x
}
```

In this example parameter `x` is type *Int* and is in a context given by the `test1` return Type of *Obj*. Therefore the value of `x` is widened to *Obj* implicitly and this code type checks, it can never fail at run-time.

```
Int test2( Obj x)
{
   return x
}
```

This code will also static type check, but the compiler will insert a run-time check to ensure that the parameter value is actually an *Int*.

Another example:

```
Int test( Int? x)
{
   return x
}
```

`Test3` will compile but return a run-time error if its parameter is `null` - which is allowed in the nullable type of `x` but not in the non-nullable return type.

3.4 Generic Typing for List, Map

Generic types, also known as polymorphic types, are useful to model operations that can be applied to different types and have type relationships between input and output types. For example a function that returns all the even numbered elements a list must return a list of the same type as its input. This cannot be captured by a fixed type system, since that would restrict the function.

Fantom allows generic types but only in system-defined methods on built-in types. For other cases dynamic typing must be used, with types approximated by *Obj*, as is always possible.

Figure3.1 shown the labels used for the constituent types of List and Map that are used in the type definitions of methods operating on these types.

3.5 Static vs Dynamic Type Checking for List, Map

```
Int [] Test( Int [] x)
{
   y = x.map(|Int n -> Int| {n*n} )
   y[0] = ""
   y[1] = 1
   return y
}
```

Type	Generic names of constituent types	Generic name for type
List	V[]	L=V[]
Map	[K:V]	M = [K:V]

Figure 3.1: Generics

This function type-checks because the result of the map operation is an *Obj[]* in Fantom. The map function is typed *Int-¿Int* but the context required by map is *Int-¿Obj* and so its static type is widened and the type inferred for `y` is therefore *Obj[]*. The resulting list will be implictly cast to *Int[].* as a return value.

```
Int [] Test( Int [] x)
{
   Int [] y = x.map(|Int n -> Int| {n*n} )
   y[0] = ""
   y[1] = 1
   return y
}
```

Here the programmer has (correctly) typed `y` as *Int[].* This function fails to static type-check because *Str* and *Int* are incompatible, so the assignment of an empty *Str* to `y[0]` fails static checking.

3.5.1 Holes in the static type system

List and Map types are not properly captured by the fits function. Specifically:

Int [] fits *Num []*

This is true for getters, because all elements read from an *Int[]* will be correct values of a *Num[].* It is untrue in general for setters. A float value may be written to a *Num[]* but this will be invalid written to an *Int[].*

The effect of this in Fantom is that in some circumstances dynamic type errors (caught by the JVM) are not seen by the compiler. This practically is acceptable because it does not happen often, and the errors are in any case caught. This is also in the spirit of implicit casting, pervasive in Fantom, which can also result in run-time errors.

3.6 Language Support for Dynamic Typing

Fantom has a selection of operators that make it more convenient to write expressions with dynamic types and `null` values. These are summarised in Figure 3.2.

Operator	Name	Example	Description
`->`	Dynamic Invoke	`obj->username`	Call a method or access a field available on the dynamic type of the instance `obj`. This is the dynamic equivalent of the "`.`" operator.
`?:`	Elvis	`x ?: y`	`if (x != null) return x else return y` The type of x must be nullable
`?.`	Safe Invoke	`userlist?.find("Bob")`	Returns the expression if the LHS is not null, otherwise returns null. Useful for chaining method calls
`?->`	Safe Dynamic Invoke	`userlist?->find("Bob")`	As above, but the method call is dynamic (can be used where the static type of userlist is *Obj*)
`as`	Safe Cast	`x as y`	Returns x cast to type `y` if this is possible, otherwise `null`. Note that the result type must therefore be nullable.
`( )`	Cast	`(Y) x`	Casts `x` to type `Y`. If this is not possible throw `TypeError` at run-time. Normally in Fantom casts are not needed since where necessary they are inserted implicitly by the compiler

Figure 3.2: Operators for dealing with types

Chapter 4

Functional Programming and Closures

Idiomatic Fantom is compact and readable partly because it can represent function calls with closure parameters compactly. This notation is easy to use and read when you understand it, but opaque when you don't. In this Chapter we will first learn about how functions and closures work in Fantom, and then look at the notation that makes it easy to write them.

There are two different use cases commonly found for this notation, used very heavily in good Fantom code.

- When constructing an object, particularly in GUI code where GUI windows are set up with complex nested widget constructor calls.
- When using a method whose last parameter is a function, such as `List.each`.

If you are familiar with Java, reading idiomatic Fantom is mostly a matter of understanding how these closures work, and what the compact syntax used for them means.

In this Chapter we will review what a function is in Fantom, and specifically how functions can be used as values. Then we will introduce the idea of a closure as a way to write an anonymous function. We will look at the (limited) support for functional programming in Fantom, as an example of the use of closures.

In Chapter 5 we will see how Fantom uses some clever syntax borrowed in part from Ruby to make use of closures in code very readable.

Those not used to functional programming will wonder why introduce this complexity of functions used as data? It turns out that there are many problems much more easily solved using this technique. In Fantom the standard library API makes heavy use of *function parameters*, for example as in `List.each` and these are normally provided by closures.

4.1 Functions as First-Class Objects

In Fantom functions are so-called `first class` objects, just like any other data type, and can be passed to other functions or stored in variables. A function is an object with a built-in `apply` method that can be used to call the function with zero or more parameters. Function objects called with too few parameters result in a run-time error. More parameters than needed is allowed, the extra parameters are ignored.

Think of a method as being a wrapper for a function object that binds it to a class allowing it to be accessed from the class or object via the method name. Function objects are normally used by calling the `apply` method:

```
sum.apply(2,3)
```

is normally written `sum(2,3)`. Here there are two possibilities for the name `sum`:

- sum is a variable set to the correct function object.
- sum is a method that wraps the correct function object.

If sum contains the standard binary add function object this will return the sum of `2` and `3`. In fact the syntax `2+3` is a shorthand for this apply method call on the `add` function.

One way to use a function object is to define a method with the required function and use the method name to access the function object. This requires some care, because in Fantom a method with zero parameters is automatically called by writing its name. To extract the function object we need to use the method name in a way that avoids this automatic function call:

```
class Funcs
{
  void main()
  {
        fvar := /*obtain sum function */
    tvar := /*obtain timeInUs function */
        echo("sum(2,3) is ${ fvar(2,3) }")
    echo("Time in us is ${tvar()}")
  }

  static Int sum(Int a, Int b) { return a+b }

  Int timeInUs() { return DateTime.nowTicks/1000 }
}
```

In class Funcs method Sum wraps the binary add function on integers. Let us see what expressions we might use to replace the comments in the code to make this work by setting fvar and tvar to the two function objects written in the `sum` and `timeInUs` methods.

1. `sumvar := sum` - the method name on its own will return the sum function object
2. `sumvar := Funcs.sum` - We could access this via the class name if we wanted because the method is static.
3. `sumvar := #sum.func` - this also works
4. `tvar := timeInUs` - this does not work because the function is automatically called
5. `tvar := #timeInUs.func` - this works
6. `tvar := Funcs#timeInUs.func`

4.2 Closures

A closure is a way to write an anonymous function as an expression.

Here is an example of a simple closure, representing an anonymous function with one `Int` parameter `n` that returns a value of type `Bool`, `true` if `n` is even. Although this code looks somewhat like a method definition, it is syntactically an expression.

```
|Int n->Bool|
{
  return n % 2 == 0
}
```

The first line `|Int n->Bool|` is the function signature that specifies parameter names and return type. Note the similarity with a method: the signature replaces the method header.

Closures are often written on a single line:

```
|Int n->Bool| { return n % 2 == 0 }
```

The `return` keyword can be omitted in a closure:

```
|Int n->Bool| { n % 2 == 0 }
```

A function from which the return type can be inferred can omit the `->ReturnType` in the signature. Note also that multiple statements inside the closure body can be written on a single line separated by `;`:

```
|Int n| { echo("WARNING"); echo("n=$n.toStr") }
```

The parameter types in the signature can be omitted when they can be inferred from the function body:

```
|n->Bool| { n % 2 == 0 }
```

As a special case a `Void` function with no parameters is written:

```
|->| { echo("Finshed") }
```

4.2.1 Unbound variables in closures

A Closure can be defined in a class or method and therefore may reference names not defined inside the closure. Such *unbound names* can be referenced by the closure if they are in scope where they the closure is defined. Inside a class fields or methods may be referenced. The object instance (of the containing class, not of the closure object) is referenced by `this` so:

```
this.methodname
```

A method local variable in scope is referenced by its name unqualified. Other unqualified names are implicitly taken as slots of `this`.

When unbound names are referenced in a closure the variable is accessed by reference, and may be read or written by the closure. Multiply calls of the closure will use the same storage for these unbound variables. Local variables and parameters will normally disappear when a method exits. If they are captured by a closure that remains alive then they will stay alive after the function call that created them terminates.

4.2.2 Mutability of closures

A closure is constant (immutable) if it cannot access any unbound mutable state (by capturing a non-constant field or variable).

4.3 Functional Programming

Fantom is an object oriented language and does not encourage functional programming. Mostly, in Fantom, functions are passed to the API rather than used to manipulate data in a functional way.

A small but useful subset of functional programming techniques is well supported by `map` and `reduce` methods on the List type, and `map` on the `Map` type.

4.3.1 Map and reduce

Map creates a new list from an old one by applying a given function to each input list element to generate the corresponding output list element. The input list is unchanged.

Fantom's static type system is not complex enough to infer the correct result type for the output list. The type will be available dynamically but to make static types checked the result type must be treated as an `Obj`.

Reduce works similarly. Again the result type is an `Obj`. The lack of precise type inference means that type casts or dynamic lookup must be used to access the output value.

4.4 Summary

Functions are first-class objects (data values) in Fantom. Methods wrap an underlying function. Closures are written as a signature in `| |` followed by a closure body in { }. There are no restrictions on what can be put inside a closure body: it has identical syntax to a method body.

The signature of a closure can as a shorthand use type inference to infer parameter and/or return types. These may then be omitted from the signature.

Closures can capture variables in scope when the closure is created. These can be local variables or object fields. These variables are captured by reference, and can be read and written from the closure. Local variables in a function remain alive even if the function terminates if they are captured by a closure that is alive.

Chapter 5

It-Blocks and Declarative Programming

Chapter 4 described how Fantom uses closures to represent functions that are used as data, for example as parameters to standard library functions.

Closures in Fantom can be written with a number of shortcuts, but normally consist of two parts: a signature in `| |` followed by a closure body in { }.

Here we will look an extension of this syntax in which the signature is omitted entirely, and some associated syntactic sugar for function calls, which together make idiomatic Fantom using closures look very clean. This is especially useful for declarative programming.

5.1 It-Blocks

Where there are no parameters, or one parameter with type inferred, the signature part of a closure is unnecessary. Fantom allows this to be omitted entirely *in contexts where a function value is expected.*

A closure without signature and without any `return` statement is called an *it-block* because the missing parameter can be used in the closure body as name *it*. Thus it-blocks are the ultimate compact way to write closures.

When a method's last parameter expects a closure it can be taken out of the method call brackets and written immediately after the method call. This syntax makes such constructions much more readable. The `map` method of a `List` expects a function so this compact notation represents a list of squares. All of these expressions mean the same thing in Fantom, and represent the list `[1,4,9]`.

```
[1,2,3].map() {it*it}      /*preferred idiomatic form */
[1,2,3].map( {it*it} )
[1,2,3].map |n| {n*n}
[1,2,3].map |Int n| {return n*n}
[1,2,3].map (|Int n->Int| {return n*n}) /*form with no shortcuts*/
```

5.1.1 Slot lookup and assignment in it-blocks

Inside an it-block closure the `it` parameter is used for default lookup of field or method names just like the object in a class:

```
[1,2,3].map { negate }  =>   [-1,-2,-3]
```

The method name `negate` is looked up as `it.negate` in this it-block so this negates each list value.

This slot lookup is particularly useful when an it-block is used *inside* a constructor. In Fantom this is common because many of the API constructors take a closure as optional last parameter of signature `|This|`. The parameter type-name `This` here is a special marker which represents the constructed object type and indicates that an it-block closure is expected. Note that in the constructor below `f` represents the it-block, and `this` binds to the constructed object. The function call `f(this)` thus applies the closure to the object.

```
// default constructor for fwt::Button accepts optional function f as parameter
// f is usually a closure written immediately after the constructor call
// as an it-block
```

```
new make(|This|? f:= null) { if (f != null) f(this) }

// constructor call with it-block initialisation
Button { size = "100x100" }
```

If an it-block closure is supplied to the constructor the closure is called, with its *it* parameter set to the constructed object, inside the object constructor.

To see why this is useful remember that field and method names inside an it-block are looked up against *it*. In this case *it* is the new Button object and so this is a convenient way to initialise fields of a new class instance.

In this example `Button.size` is set to `100x100`.

Inside the constructor the closure is given static permission to set (initialise) `const` fields on the object. So this is a way to parametrise newly created constant objects. Note that the same operation would not work with the closure called outside the constructor on the object instance, because at that time `const` fields cannot be changed.

5.1.2 It-Add Statements (AKA comma operator)

One final tweak of the it-block syntax is especially useful when it-blocks are used to initalise construction of objects. If an expression inside the it-block is terminated with a `,` the expression is used as a parameter in an `it.add` method call:

```
x,   => it.add(x)
```

Idiomatically this allows it-blocks used to initialise constructors that have a number of field assignments, followed by a number of calls to the `add` method. This is useful in GUI programming as the following examples show.

5.2 With-blocks

With-blocks superficially look and behave like it-blocks, but they are semantically very different.

An it-block is a closure appended to a method expecting a closure, possibly to a *constructor* (typically shortened to a class name). Inside the closure the comma operator appended to an expression translates to an `add` method call on the constructed object because of the magic of a special constructor that *expects a closure* as parameter and applies the closure to `this` *inside* the constructor. other statements can be used conveniently to set object fields because a name that does not resolve locally will be matched against the closure `it` parameter which is, because of the form of the constructor, the same as the constructor's `this`.

A *with-block* is an extension of this paradigm to a more general case, where an it-block closure `clos` is appended to any expression `exp` *not* expecting a closure. When this happens the expression's `with` method `exp.with` is implicitly used. The effect of this is to apply the expression to the closure, returning the original expression:

```
clos(exp); exp
```

That is equivalent to the `with` method (on `Obj`, inherited by every class) being defined:

```
virtual This with(|This| f)
{
  f.call(this)
  return this
}
```

Parenthetically it is worth noting that because `with` can be overridden with-block semantics is customisable on a per-class basis.

For a simple example, incorporating the `,` add operator:

```
[1,2] {3,4,} => [1,2].with { it.add(3); it.add(4) } => [1,2].add(3).add(4)
```

Here {`3,4,`} is the with-block. Compare this with:

```
List {1,2,3,4,} =>  /* error */
```

This does not compile because, unusually, the List class does not have any constructor written specially to accept a closure. Also the List class has special syntax for its constructor and no *make* method, so List on its own is not a valid expression.

The semantic distinction between it-blocks and with-blocks is that an it-block applied to a (closure accepting) constructor will operate on the constructed object *inside* the constructor and therefore may initialise `const` fields and objects.

A with-block operates on an already constructed object and therefore cannot change `const` fields.

Syntactically the two forms are identical.

```
ClassName  <it-block contents>
```

The `ClassName` written on its own will, if possible, result in an implicit call to its constructor. The semantics then depends on whether this accepts a closure parameter, or whether it does not.

5.3 Declarative Programming Examples

Here is how to define a top-level Window widget, containing three Button widgets.

```
class Main
{
  Void main()
  {
    Window
    {
      it.title = "two Buttons"
      it.size = Size(100,200)
      Button { text = "A"},
      Button { text = "B"},
      Button { text = "C"}
    }.open
  }
}
```

The `Window` widget is initialised with an it-block that also uses it-add to add three `Button` widgets to the window. The `add` method is invoked by appending the `,` operator to the relevant expression (the Buttons). Note that as a convenience the *last* expression in the block is assumed it-add and does not require a `,`.

Each Button widget is itself initialised on construction with a nested it-block that sets the `text` field of the widget.

The `open` method of the top-level window starts the GUI event loop.

User classes can be used in the same fashion as this example from the standard Fantom desktop fwt example shows:

```
class DesktopDemo : Canvas
{
  Void main()
  {
    Window
    {
      it.title = "Desktop Demo"
      it.size = Size(600,400)
      DesktopDemo {},
    }.open
  }

  /* custom onPaint method for DesktopDemo not shown */
}
```

This function constructs a `Window` GUI widget and then calls its `open` method which has the effect of starting the GUI event loop.

The it-block that constructs the `Window` initialises fields `title` and `size` on the `Window` object and then calls the `add` method of `Window` to add the constructed object of type `DesktopDemo` to the newly created window. The `DesktopDemo` object is sub-classed from `Canvas` so this will add a `Canvas` widget to the window. Note that the it-block that initialises `DesktopDemo` (whose constructor is inherited from `Canvas`) in this case contains no further initialisation. This style can easily be used to add nested sub-widgets as is needed, either from the standard API or user-defined.

5.4 Summary

Closures can be written compactly in Fantom as *it-blocks*. This notation combines with syntactic sugar that allows a last function parameter to be pulled out of a function call's brackets and written without brackets after the function. It allows clean code when used with many API functions that accept functions as parameters.

Widgets, and many other Fantom API classes, have constructors that accept an optional last parameter that is a closure called from inside the constructor. That allows the closure to initialise object fields, even `const` fields.

Inside an it-block an expression terminated by `,` is translated into a call of the `add method` on the `it` parameter.

GUI code typically consists of nested `make` and `add` calls with initialisation and benefits from this notation. Closures are written using field assignment to initialise each widget, and `,` after sub-widgets constructors - themselves using it-blocks - to add sub-widgets.

Any expression not expecting a closure, with it-block appended, forms a *with-block*. The with-block closure is called with the expression as parameter. This is a post-construction equivalent of an it-block appended to constructor.

Chapter 6

Concurrency

Modern CPUs have 4, 8, or more CPU threads and to use them efficiently requires a program that is *concurrent*. Concurrency is also useful in GUIs to provide real-time response while simultaneously executing background code.

Fantom uses an Actor framework (as used in Erlang) to control concurrency. Actors run on different concurrently executing threads. Communication between actors (and therefore possibly between concurrent threads) is via messages. Fantom forces messages to be , something that greatly reduces the likelihood of obscure errors due to shared mutable state between concurrent threads.

Actors are defined by extending the *Actor* class with a concrete method.

```
Obj receive(Obj msg)
```

This method contains the code that defines what an actor does when it receives message `msg`. The return value is the *reply* message that can optionally be read by the sender of the original message.

This paradigm allows both synchronous (where the sender waits for the reply) and asynchronous (where the reply is ignored) concurrency.

Given an actor you can send messages to it, and wait optionally for reply messages coming back from it, as in Example 1 below. Actors are viewed by Fantom as immutable, and can be passed by reference to other actors in messages, as in Example 2. So this framework allows arbitrarily complex networks of concurrent processes that pass messages.

Actors are controlled by an object of class *ActorPool* that automatically schedules actors onto concurrent threads as required by the current message load.

6.1 Example 1 - Concurrent Computation with Results Collected

Figure 6.1 illustrates the communication with a number of actors from the code in Listing 6.1. The `main` thread creates a large number of actors, each of which computes concurrently the primality of the integer it is sent, the Boolean results are passed back to `main` and collated. The arrows in the Figure represent messages. In Fantom code messages are processed by two methods from module *concurrent*:

- `fut := a.send(msg)` //sends message mess to Actor `a`, returning Future `fut`
- `fut.get` // waits for the reply to message `msg`, which by definition is equal to `a.receive(msg)`

Each Actor receives a single message (the *Int* for it to check) and returns a Boolean value indicating primality using method `isPrime`, which is set to be the code executed by the actor on receiving a message.

The `main` thread operates as follows:

1. Create an *ActorPool* object to control the actors:

   ```
   apool := ActorPool {maxThreads = numOfThreads}
   ```

2. Create all actors, store them in Map `actors` keyed by the integer they will check for primality.

```
using concurrent

class PrimeNumbers
{
  Int startNumber := 100000000000
  Int numberToTest := 1000
  Str checkInterval := "0.3sec"

  Void main()
  {
    p := PrimeNumbers()
    p.test(10)
  }

  static Bool isPrime( Int primeCandidate)
  {
    for (i := 2; i < (primeCandidate.toFloat).sqrt.ceil.toInt+1; i++)
    {
      if (primeCandidate % i == 0) return false
    }
    return true
  }

  Void test(Int numOfThreads)
  {
    apool := ActorPool {maxThreads = numOfThreads}
    [Int:Actor] actors := [:]
    [Int:Future] futures := [:]

    echo("creating Actors...")
    for (pc := startNumber; pc < startNumber+numberToTest; pc++)
    {
      actors[pc] = Actor(apool, |Int i->Bool|{isPrime(i)} )
    }

    t1 := Duration.nowTicks
    actors.each |a, pc| {futures[pc] = a.send(pc)}

    num := 0
    apool.stop //no new messages allowed to actors, allows pool to be done
    while (! apool.isDone)
    {
      num = 0
      futures.each { if (it.isDone) num++ }
      echo("$num actors finished")
      try
        apool.join(Duration.fromStr(checkInterval))
      catch(TimeoutErr e) {}
    }

    elapsedMs := (Duration.nowTicks - t1)/1000000
    num = 0
    futures.each { if (it.get) num++ }

    echo("Finished in ${elapsedMs}ms using $numOfThreads threads.")
    echo("$num primes found in $numberToTest numbers tested")
  }
}
```

Listing 6.1: Example 1

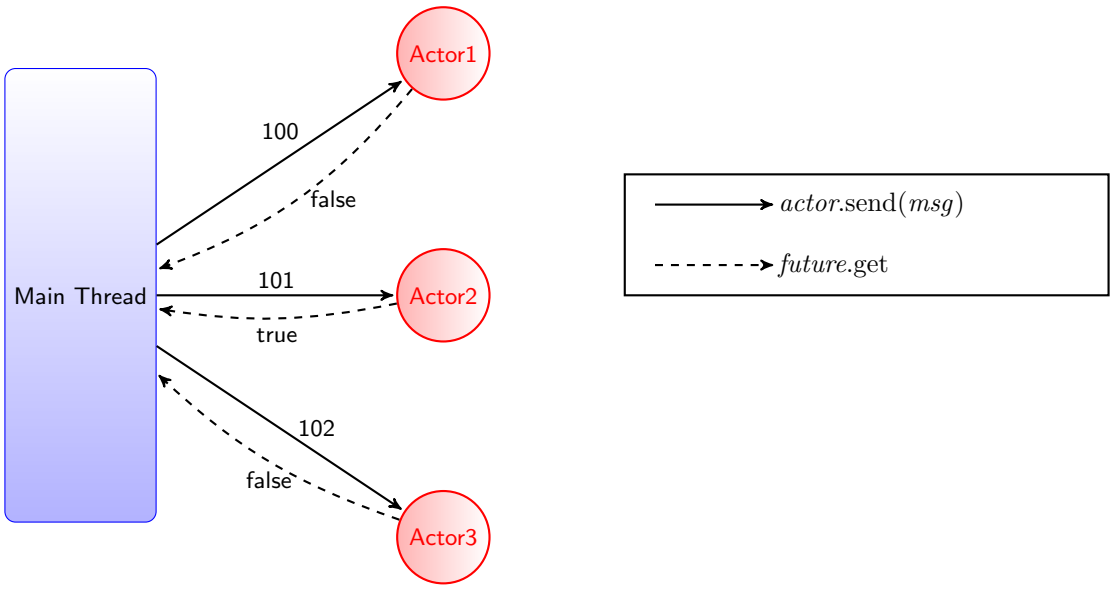

Figure 6.1: Actors to calculate primes concurrently

```
actors[pc] = Actor(apool, |Int i->Bool|{isPrime(i)} )
```

3. Send each actor its integer `pc` (a Fantom *Int*) to check, storing the corresponding *Future* object (from which the reply message can be read) in Map `futures`.

```
futures[pc] = a.send(pc)
```

4. Mark the *ActorPool* (`apool`) as stopped. This signals that no more messages can be sent to the Actors, and allows the `apool` to reply with `isDone = true` when all actors have finished processing their messages.

```
apool.stop
```

5. Loop waiting for actors to finish. Each loop first counts the number of finished actors and prints this:

```
num = 0
futures.each { if (it.isDone) num++ }
echo("$num actors finished")
```

then waits for either all actors to finish or a given time (`checkInterval`) to elapse. The `ActorPool.join` method has a timeout that throws *TimeoutError* if the actors are still not finished at the end of the given interval and implements this:

```
try
  apool.join(Duration.fromStr(checkInterval))
catch(TimeoutErr e) {}
```

6. When all actors have finished `get` obtains their results to display the total number of primes found:

```
futures.each { if (it.get) num++ }
```

7. This code is elaborate to illustrate different ways of interacting with an *ActorPool*. If the aim were just to obtain all the results, successive calls to `get` on each of the `Future` objects would return the correct values, waiting as necessary for each actor to finish:

```
futures.each { if (it.get) num++ }
```

8. The maximum number of threads used by the `ActorPool` is given as the parameter `numOfThreads` of `test` in `main`. As this number is changed from 1 upwards it results in faster execution with more concurrency. For example, on an i7 CPU with 8 threads the execution time will continue to decrease until `numOfThreads` is 8 after which increases make no difference, as the extra actors are scheduled sequentially on the available threads.

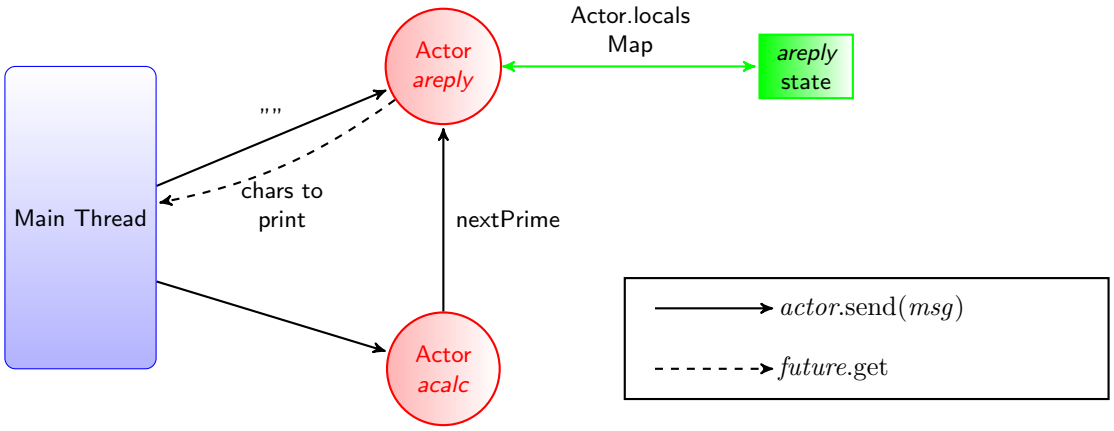

Figure 6.2: Actors to return real-time output from asynchronous long-running process

6.2 Example 2 - Output from a long-running task

Figure 6.2 illustrates the messages sent between actors when the `main()` thread creates a long-running process defined by a Fantom function running on an Actor, as in Listing 6.2. Standard output from the program is returned to the `main()` thread and displayed there. In Fantom actors all share the same standard output as the main thread, so that a similar effect could be obtained more simply by using `echo` in the actor. The structure presented here is useful because the output from the long-running process can be manipulated programmatically by the main thread.

1. The `main` thread creates two actors in order to run an asynchronous program and print the output in real time.

```
areply := Actor(apool, |Obj o->Obj?|{printFunc(o)})
acalc := Actor(apool, |Int pc->Void| {findPrimes(pc, numberToTest, areply)} )
```

2. Actor `acalc` runs continuously and outputs prime numbers from time to time. It sends these back to `main` via the second actor `areply`. Actor reply accepts messages from both `acalc` and `main`. Integers sent from `acalc` are interpreted as numbers to print, and strings from `main` are interpreted as requests to output all queued characters. The characters remaining to print are queued in internal `areply` state `Actor.locals` and returned to `main` as a reply to an empty *Str* message sent from `main`.

3. This communication structure allows data to be sent back from an actor to the main program but does not synchronise. The main program must repeatedly poll `areply` by sending it a message and waiting for the reply. The code here inserts a 0.1 second delay between each successive poll so that the main thread does not hog CPU. This is good practice, see point 7 below.

```
while (true)
{
  s = areply.sendLater(100ms,"").get
  Env.cur.out.writeChars("$s")
}
```

4. The `acalc` actor is given a `receive` function, which defines its operation:

```
acalc := Actor(apool, |Int pc->Void| {findPrimes(pc, numberToTest, areply)} )
```

5. The closure used here to define the `acalc` actor must be immutable. This is so because `findPrimes` is static, and its parameters: `pc` is passed to the closure, `numberToTest` is static, `areply` is a (constant) actor.

6. The `areply` actor must store the characters sent to it from `acalc` so that they can be output in reply to a message from `main`. This is done using `Actor.locals` a special Map that holds internal state to the actor:

```
h :=  Actor.locals.get("printFuncState","") as Str
if (s is Int)
{
```

```
using concurrent

class PrintPrimesAsync
{
  Int startNumber := 10000000000000
  static const Int numberToTest := 10

  Void main()
  {
    apool := ActorPool {maxThreads = 2}
    areply := Actor(apool, |Obj o->Obj?|{printFunc(o)})
    acalc := Actor(apool, |Int pc->Void| {findPrimes(pc, numberToTest, areply)} )

    acalc.send(startNumber) // start asynch program

    while (true)
    {
      s = areply.sendLater(Duration.fromStr("0.1sec"),"").get
      Env.cur.out.writeChars("$s")
    }
  }

  static Str? printFunc(Obj s)
  {
    h :=  Actor.locals.get("printFuncState","") as Str
    if (s is Int)
    {
      Actor.locals["printFuncState"] = "$h$s.toStr\n"
      return(null)
    }
    else
    {
      Actor.locals["printFuncState"] = ""
      return h
    }
  }

  static Void findPrimes( Int start, Int numToPrint, Actor printFuncActor)
  {
    pr := start
    i := 0
    while (i < numToPrint)
    {
      pr++
      while (!isPrime(pr)) pr++
      printFuncActor.send(pr)
      i++
    }
  }

  static Bool isPrime( Int pc)
  {
    for (i := 2; i < (pc.toFloat).sqrt.ceil.toInt+1; i++)
    {
      if (pc % i == 0) return false
    }
    return true
  }
}
```

Listing 6.2: Example 2

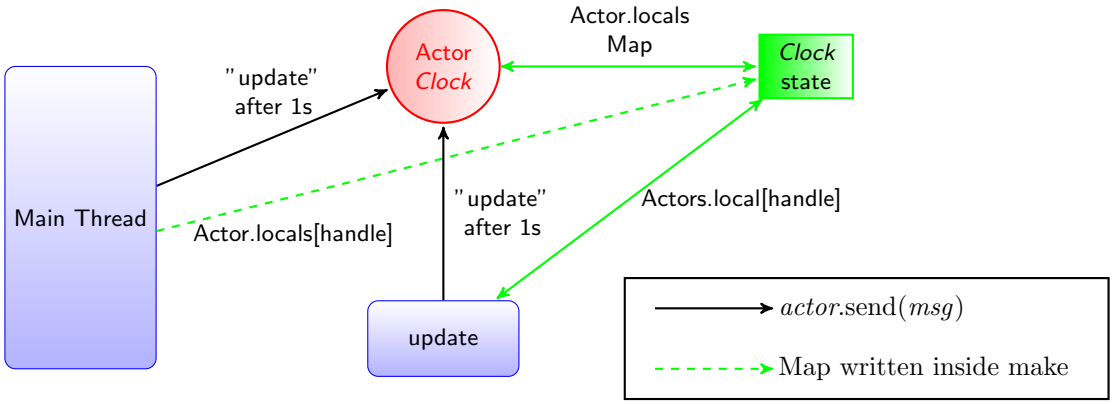

Figure 6.3: Actor with Desktop.callAsync

```
    Actor.locals["printFuncState"] = "$h$s.toStr\n"
    return(null)
}
```

7. This code requires two hardware threads to run successfully (for example a dual CPU). Actor `acalc` will run continuously on whichever thread it is scheduled. Actor `areply` will receive queued messages but not process them unless it can run on another hardware thread. It requires very little time to process each message so will not run continuously. The main thread sleeps in between each time it polls `areply`, so can be successfully scheduled on the same thread as `areply`. Actors have no notion of time-slice multitasking on granularity smaller than processing a message. Normally actors will process incoming messages quickly so that scheduling on a per message basis is fine. Note that in this example `acalc` runs forever having been sent just one message and so is atypical.

8. This example illustrates how the *Future* returned from a message send command need not be used if the communication is unidirectional and so a reply is not needed.

6.3 Example 3 - Interaction with a GUI

The previous example shows how to capture output from a long-running task. Typically this would be done inside a GUI, with the GUI event loop replacing the `main` thread. One way to implement this would to replace the `main` thread `areply` poll in 6.2 by a polling function running in the GUI event loop at fixed intervals. A much easier solution is available using the Fantom *Desktop* class from module *Fwt*. A special function `Desktop.callAsync` is designed to be static and can be called by any actor. This allows any actor directly to interact with the GUI by posting an immutable callback which is processed by the event queue in the Desktop UI thread - typically this is the `main` thread and not an actor.

The trick in this example is how to obtain the necessary communication between the callback and the UI. The callback cannot directly reference mutable UI state - it must be immutable. But it *can* reference its own actor's `Actor.locals` internal state, and this is mutable, although Fantom treats this special actor instance field as *immutable*.

Using something wrapped by `Actor.locals` answers half the problem: how the callback can operate on mutable state. But how can private actor local state contain a UI widget? After all, we cannot send a mutable reference to an actor, nor can we create a mutable reference in any actor field.

The answer is that in the actor's constructor, before the actor itself starts execution (on some foreign thread) anything can be written into the `Actor.locals` Map from the main thread under some known key, and then accessed by the actor, and also by the callback.

the clock example for *fwt* under the standard documentation is shown in Listing 6.3 and illustrates this example. It works as follows:

- Construct an actor `act` in the UI thread

```
clock := Clock(display)
```

- Inside the actor constructor store the UI object with state that needs to be accessed (typically a widget `widget`) in `Actor.locals` under a unique known constant (Str) handle `handle`, pass this handle to the actor, or else as here give it to the actor at compile-time as a `const` field.

```
new make(Label label) : super(ActorPool())
  {
    // Clock instance must be created by the main UI thread,
    // which is where we need to cache the label
    Actor.locals[handle] = label

    // send the first clock-tick
    sendLater(1sec, updateMsg)
  }
```

- Inside the receive function of actor `act` call `Desktop.callAsync` with a immutable closure `update`.

```
if (msg == updateMsg)
    {
      Desktop.callAsync |->| { update }

      // send the next clock-tick
      sendLater(1sec, updateMsg)
    }
```

- Closure `update` will be queued on the UI event queue and execute in the UI thread.
- Inside closure `update` the reference to the actor's `Actor.locals` Map can be used. Read `Actor.locals[handle]` to retrieve `widget`, then operate on `widget` to read or mutate UI state. Note that `update` is written in this example, for convenience, as a method of class `Clock` which extends `Actor`.

```
Void update()
  {
    label := Actor.locals[handle] as Label
    if (label != null)
    {
      time := Time.now.toLocale("k:mm:ss a")
      label.text = "It is now $time"
    }
  }
```

- Data from `act` can be passed to the UI via bound parameters in `update` (good practice) or additional objects shared between the actor and `update` and stored in `Actor.locals` (bad practice).

Listing 6.3: Clock GUI communication

```
using concurrent
using fwt
using gfx

**
** Display a clock label and update from a background task
** using an actor.
**
const class Clock : Actor
{
  **
  ** the only message we understand...
  **
  static const Str updateMsg := "update"

  **
  ** generate a unique handle for this actor
  **
  const Str handle := Uuid().toStr

  **
```

```
  ** create a Clock that updates a UI Label
  **
  new make(Label label) : super(ActorPool())
  {
    // Clock instance must be created by the main UI thread,
    // which is where we need to cache the label
    Actor.locals[handle] = label

    // send the first clock-tick
    sendLater(1sec, updateMsg)
  }

  **
  ** receive a message in actor's own pool thread
  **
  override Obj? receive(Obj? msg)
  {
    // assuming we recognize the message, have the
    // UI thread execute an update (with this actor's
    // update method, below)
    if (msg == updateMsg)
    {
      Desktop.callAsync |->| { update }

      // send the next clock-tick
      sendLater(1sec, updateMsg)
    }
    return null
  }

  **
  ** Must be called from the UI thread; will look up
  ** our label using the const handle, and set its
  ** text with the current time.
  **
  Void update()
  {
    label := Actor.locals[handle] as Label
    if (label != null)
    {
      time := Time.now.toLocale("k:mm:ss a")
      label.text = "It is now $time"
    }
  }


  static Void main()
  {
    // label to update
    display := Label
    {
      text   = "Does anybody know what time it is?"
      halign = Halign.center
    }

    // clock actor
    clock := Clock(display)

    Window
    {
      size = Size(200,100)
      title = "Clock Actor"
      display,
    }.open
  }
}
```

6.4 Actors In Detail

Actors are useful because they hide details of concurrency synchronisation and allow you to write your code at an abstract level. They just work, and tend to suffer less from race and lock conditions than more general concurrent code.

Sometimes it is helpful to understand more about the underlying implementation. In Fantom each Actor object (referred to throughout this section as an actor) is associated with an ActorPool object which controls scheduling and is given by the programmer a fixed number of threads as a resource that can, but need not, be used. These threads are in fact Java Threads in the implementation, and represent lightweight tasks that can be scheduled onto available physical CPU threads (typically between 2 and 16) in the hardware. The scheduling of Java threads onto physical threads is handled by Java, and not described here.

The Fantom programmer can decide whether to use one ActorPool for all actors, sharing threads, or whether to put different actors in different ActorPools each with a separate thread allocation.

The Fantom run-time then schedules actors onto available ActorPool threads. This scheduling is done at a message granularity. When an actor has messages to process, and hence computational work to do, it is scheduled onto the next available thread, or immediately onto a new thread if the ActorPool thread limit has not been reached. It will be kept scheduled on the same thread until either it has no more messages to process, or it has processed 100 messages. Then it may be removed from the thread and resources given to another actor instance.

`Actor.locals` is actor local state that persists from one call of the receive function to the next and that travels with the actor whenever it is scheduled as part of the actor's context. `Actor.locals` is a single reference to a map that is immutable - although the Map itself is mutable, and which is copied to a new thread when the actor is scheduled. The context also contains locale information which is read/write and copied to and from a thread when an actor is scheduled or descheduled.

6.4.1 Exceptions in Actor Code

Exceptions that happen in an Actor `receive` function are not necess arily caught. If the function results are read using the the future, the get method that does this will be given the exception, which will be obvious if not handled. However if an actor executes asynchronously any exceptions are not automatically caught. Therefore good practice using actors is:

Decide where any exception will be handled: If synchronous usually this will be in the future, if asynchronous in the actor receive function. Do one of the following:

- Wrap the receive function in a `try - catch` block to check for exceptions, unexpected exceptions could be logged to stdout.
- Wrap the `future.get` call in a `try - catch` block.

Chapter 7

Conclusions

Fantom sits with a number of other post-Java languages in a space made available by good compiler technology and virtual machines, with run-time systems that can easily support different high level languages. It is characterised by pragmatism, preferring what works to what is a fun feature. It shares a concern with language utility with Python, but adds to that a much more conventional view that static typing brings benefits in documentation and easy correction of programmatic errors. It is about as fast as Java and therefore does not have the performance overhead of dynamic languages. One caveat is that Fantom is determinedly future-proof, and all numbers are therefore 64 bit. This introduces a speed penalty in 32 bit numeric code for the benefit of greater simplicity and scalability.

Programmers no longer have to use languages that are plain bad. As understanding has developed so what works best has changed and any new language will tend to be better than languages that have evolved from a starting point many years ago.

For new languages to be used good tools and a stable mature compiler are essential. Fantom has both, nicely documented on the Fantom web site. The language is actively supported by a number of developers who use it commercially in major projects.

One important and actively supported use of the language is for web programming where it can be compiled to JavaScript and run on the client side of applications. The ability to use the same language for client and server side code simplifies web development.

Personally I enjoy Fantom because it has the great productivity and compactness of a scripting language such as Python while providing the contractual security of a statically typed language. I enjoy writing in Python but find the lack of static typing a continual irritant and a real headache when maintaining code. Why should programmers not get proper help from compilers when this can be provided easily and at no cost?

One way to summarise a language is to look at where it sits in the spectrum of design tradeoffs on various axes.

Functional programming

Fantom provides exceptional support for functions as first class objects, with closures that can capture local variable references, syntax that is superior to any other language I know, and one of the reasons for liking the language. However it is not particularly designed for functional programming. Support for functional operations on lists, maps is good with convenient syntax and standard `map`, `reduce` but not perfect. The static type system does not currently (1.0.68) track types properly across `List.map` and this is annoying, because results default to `Obj? []` and so static type checking is unnecessarily lost. other language constructs that support convenient manipulation of functional data structures, such as anonymous types and pattern matching, do not currently exist. In Fantom functions tend to be used to manipulate object-oriented data structures and the trade-offs in adding the missing functional features are unclear. The language therefore sits on the object oriented side of the OO-functional axis.

Static versus Dynamic Typing

Fantom lets the programmer choose within a framework that encourages the use of static typing. For many applications Fantom's combines the benefits of both styles. For example, having nullability made explicit in the type system means that programmers must think more carefully about whether objects can be null and provides

static checking for a large and frustrating set of typical java run-time errors. On the other hand Fantom's static type system is in other ways not as expressive as Java with only limited support for generics. This is of course not required because wherever static types don't fit dynamic typing can be used, but some will mind the loss of static expressiveness.

Type inference, on the other hand, is very welcome. Those who have used strongly typed functional languages will take type inference for granted: its use eliminates a lot of unnecessary programming cruft and makes programs more compact and programming more productive.

The key design decision that places Fantom in between static and dynamic languages is the use of implicit casting whenever the cast value *could be correct*. This works surprisingly well: nearly all type errors are still caught by the compiler, and those which are not so caught will be localised to the method in which they originate. The implicit casts together with type inference almost eliminate type-related cruft in code, while static types in method and field signatures are still required.

Overall Fantom sits towards the static typed end but unlike for example Scala does not provide very complex and competent static typing, preferring to replace that complexity where needed by dynamic typing.

Built-in high level data types

Fantom as all modern languages scores well with built-in convenient support for maps (hashes) and lists, including specialised literal syntax. Sets currently are missing as a built-in construct, as are user-defined generics that would allow user-defined functions with polymorphic type signatures.

Pattern Matching and Anonymous Tuples

Fantom does not have either feature and this is a loss which however is less important than you might think because the language encourages heavy use of closures to operate on objects rather than functional programming.

Domain Specific Languages

The modern (and much more powerful and well motivated) equivalent of C's preprocessor is the *domain-specific language* or DSL. A DSL is way to modify language syntax locally with custom features for a particular application. Fantom has pluses and minuses when it comes to extending language syntax. Its built-in DSL mechanism provides a seamless and very powerful extension to language syntax with convenient usage. Some built-in DSLs deal with, for example, regular expressions. However that power is provided by exposing the compiler internals to the programmer and although the compiler is pretty stable it is not guaranteed to remain so, therefore complex use of DSLs, even though it is relatively easy to implement, is potentially less maintainable than it would be in a language with a defined DSL semantics that is guaranteed not to change.

Fantom thus takes DSLs seriously, but has an implementation which is both powerful and questionable.

Reflection and Serialisation

Fantom has excellent support for reflection, similar to that which you would expect in a dynamic language, and built-in serialisation syntax that is easy to use, looks like Fantom code, and is one of the merits of the language.

Completeness

Fantom replaces the standard libraries of its main target, the JVM. That makes a higher bar for takeup for experienced programmers who have used Java - who wants to learn a new set of library functions? The benefits are:

- Better standard libraries for most common tasks (the standard libraries are fairly complete but there will always be things they do not do). This is specifically motivated by the wish to use larger classes and provide function parameters where that is natural. It is not surprising that thoughtful rewrite of the Java libraries, which have evolved over many years, results in a cleaner interface.
- Better portability, specifically between JVM and JavaScript although .NET is also a target.

So Fantom is on the all-in-one side of this trade-off. The Fantom tool set is very good and mature as far as deployment compilation and debugging go. There are multiple available competent IDEs but this could be improved. The only IDE with complete in-editor syntax error detection works fine as of now but is not actively supported.

Concurrency

Fantom takes concurrency very seriously by building mutability into its type system. The Fantom implementation of actors is fit for purpose, and can be used. It provides higher level support than typical programming with threads. However it is not a core language feature and has peculiarities. Concurrency is perhaps an area of Fantom where new constructs would be welcomed.

Simplicity versus Complexity

Fantom is a simple language and easy to learn, especially for those already familiar with Java. It has evolved weighing up the trade-offs of adding features against their cost. That means fewer features, and a more mundane language, but arguably a better one. The main barriers to understanding Fantom are surprising, but natural when understood, it-block syntax and a high-level actor-based mechanism used where concurrency is needed.