



ÉCOLE POLYTECHNIQUE  
FÉDÉRALE DE LAUSANNE

LAMP  
LABORATORY FOR PROGRAMMING METHODS

SEMESTER PROJECT REPORT

---

# Reflecting Scala

---

*Author:*  
Yohann Coppel

*Supervisor:*  
Prof. Martin Odersky

*Assistant:*  
Gilles Dubochet

January 12, 2008

## Abstract

Object-oriented languages usually implement an API supporting meta-level operations such as reflection. However, reflection APIs generally do not follow the three design principle for reflection and meta-programming. These three principles specify that meta-level facilities must *a*) encapsulate their implementation (*encapsulation*); *b*) be separated from base-level functionality (*stratification*); and *c*) their ontology should correspond to the ontology of the language itself (*ontological correspondence*) .

The Scala programming language does not have any specific API supporting meta-programming. However, since it is compiled into Java bytecode, it is compatible with the complete Java library. Consequently, one can use the Java reflection API in order to access meta-level informations about a Scala program. Anyhow, this approach presents important limitations and raises some usability problems.

We introduce a new mirror based reflection API for the Scala programming language, following the three principles previously mentioned. This API also maintains an important separation between classes, seen as data structures, and types, more abstract elements.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Java and Scala metaprogramming . . . . .	3
1.2	Scala mirroring API overview . . . . .	4
<b>2</b>	<b>Pickler and UnPickler</b>	<b>5</b>
2.1	The Scala compiler's pickler . . . . .	5
2.2	Pickled informations . . . . .	6
2.3	Pickler format . . . . .	6
2.3.1	Symbols . . . . .	6
2.3.2	Types . . . . .	8
2.4	Pickled informations: problems, limitations and workarounds . . . . .	8
2.5	From pickled information to ontological correspondence . . . . .	10
<b>3</b>	<b>Reflecting Scala</b>	<b>12</b>
3.1	The Reflect object: main entry point . . . . .	12
3.2	Mirror . . . . .	13
3.3	Classes, objects and traits . . . . .	14
3.3.1	Trait mirror . . . . .	14
3.3.2	Instantiable class mirror . . . . .	14
3.3.3	Object mirror . . . . .	15
3.3.4	Java class mirror . . . . .	16
3.4	FieldMirror: Val and Var mirrors . . . . .	16
3.5	Dealing with types . . . . .	17
3.5.1	Type mirrors . . . . .	17
3.5.2	Alias mirrors . . . . .	18
3.6	Method mirrors . . . . .	18
3.6.1	Class method mirrors . . . . .	19
3.6.2	Instantiated method mirror . . . . .	19
3.6.3	Overriden method mirror . . . . .	20
3.7	Instance mirrors . . . . .	22
<b>4</b>	<b>Reflecting on Scala reflection</b>	<b>23</b>
	<b>Acknowledgement</b>	<b>25</b>

# 1 Introduction

## Prior definitions and notes

As in Scala the term “object” is reserved for data structures representing singletons, we will use exclusively “instance” to refer to a particular instance of a class. Assume that when you read “object” we are always referring to a Scala singleton.

Code listings are generally only segments, a “// ...” indicates that the listing is not complete.

## 1.1 Java and Scala metaprogramming

The concept of mirrors, introduced by Gilad Bracha and David Ungar in [BU04], in essence follows the three design principles for meta-programming facilities. We chose to implement our API based on mirror principles.

The Scala programming language [Oa04] is compiled to Java bytecode, and therefore can be executed by any Java virtual machine [LY99], as long as the Scala library is present. With this structure, we could use the Java reflection API in Scala, without any further modifications. However, this approach suffers from some limitations and problems:

- the Java reflection API does not follow the principle of stratification,
- the ontological correspondence can obviously not be respected when applied to a Scala program and its higher level data structures: during the compiling process, Scala data structures (such as objects, traits, case classes, vals. . .) are mapped to usual Java structures (classes, interfaces, variables. . .). This transformation often implies some renaming, and fresh names are generated in many situations, using reserved characters such as \$ or -,
- we are bound to the facilities offered by the Java [GJSB00] meta-programming system,
- we can’t take advantage of the facilities offered by the Scala language itself, like apply methods.

The situation can be even worse. The following example raises a `ClassNotFoundException`, as type aliases are converted at compile time, and the type `Scientist` doesn’t even exist for the JVM (as well as `Scala.Predef.String`, `scala.Predef.Int...` which is even more problematic):

```
class Person
type Scientist = Person
Class.forName("Scientist")
```

To address these problems, our API is built around the principles of mirrors [BU04]. The reflection API is completely contained into a separate subsystem (package `scala.reflect.mirrors`), and can be disabled to reduce the footprint<sup>1</sup>, or save deployment time of any non-reflective program. More benefits of mirroring APIs are discussed in [BU04].

---

<sup>1</sup>required memory

The mirror classes are outlined in Fig. 1, and will be discussed more in details in Sec. 3.

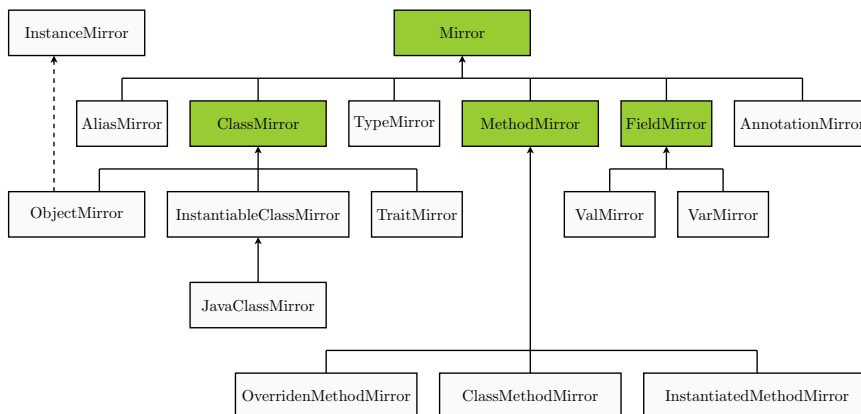


Figure 1: Mirrors API; green boxes are abstract, gray are instantiable.

## 1.2 Scala mirroring API overview

Before going into details, let's see what using our API looks like. It has been designed with flexibility and developer-friendliness in mind, so in some situations, there are several ways of obtaining the same results.

But let's take a trivial example. Imagine we have this simple `Person` class:

```

class Person(val title: String, val name: String) {
  def this(title: String, firstName: String, lastName: String) =
    this(title, firstName + " " + lastName)
  override def toString: String = title + " " + name
  def greetings(greet: String) = println(greet + " " + this)
}
  
```

We want to instantiate an object `Person`, Sir Isaac Newton, store it in a variable `newton`, and execute the method named `greetings` with the argument "Hello" (and therefore print "Hello Sir Isaac Newton"). Using the standard Java reflection API in Scala, we would have to do something like:

```

val stringClass = Class.forName("java.lang.String")
val myClass = Class.forName("testpackage.Person")
val constructor = myClass.getConstructor(Array(stringClass, stringClass,
  stringClass))
val newton = constructor.newInstance(Array("Sir", "Isaac", "Newton"))
val method = myClass.getMethod("greetings", Array(stringClass))
method.invoke(newton, Array("Hello"))
  
```

With our new API, we could replace this code by:

```

val classMirror = Reflect.instantiableMirror("testpackage.Person")
val newton = classMirror("Sir", "Isaac", "Newton")
Reflect(newton)("greetings")("Hello")
  
```

From the user point of view, we can immediately notice some improvements: *a)* no need to define `Class` objects; *b)* implicit use of constructor and method instances; *c)* a more intuitive code thanks to *apply* methods; and *d)* a less verbose code.

## 2 Pickler and UnPickler

### 2.1 The Scala compiler's pickler

During the compilation process (represented on fig. 2), the Scala compiler generates two types of data. The first one is some classic Java bytecode, which can be read and executed by a standard Java virtual machine. The second one is what is called “Pickled data”, and represents the basic structure of the original source file.

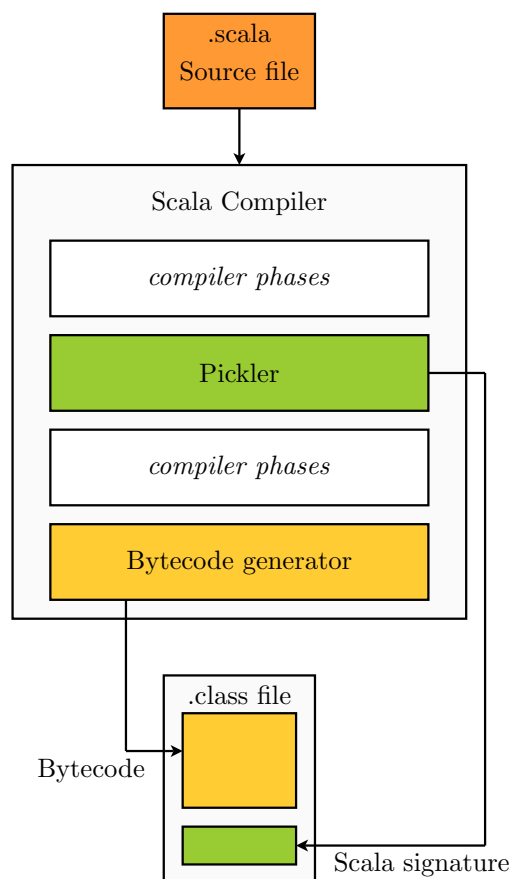


Figure 2: Pickler

This information is enclosed in a `.class` file. The Java bytecode specification [LY99] allows the compiler to “define and emit class files containing new attributes in the attributes tables of class file structures”. These attributes are silently ignored by JVMs if they do not recognize them.

The Scala compiler stores the pickled information in an attribute called `ScalaSig`. We will use this `ScalaSig` attribute to reconstruct the original Scala code structure.

## 2.2 Pickled informations

The Scala compiler generates pickled data for about any data structure in a Scala program, called symbols in the pickler context.

Symbols are stored linearly with the format shown on Fig. 3.

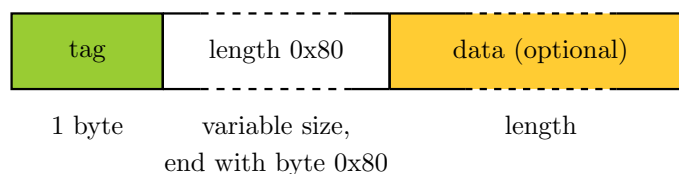


Figure 3: Pickle format

The tag represents the type of data stored, then the length gives the length of the following data block. The data block can contain multiple information, such as the name of a symbol.

A selected part of the `ScalaSig` attribute is defined on Fig. 4.

The first difficulty was to determine the meaning of every argument, and what kind of information was pickled with it.

The Symbol table defines a table, where each entry refers to one symbol. In a symbol description, we can not define a new entry, but we will refer to another entry defining this symbol instead. In other words, information is stored linearly, rather than hierarchically as suggested by a language like Scala.

## 2.3 Pickler format

We will describe here useful (and non-trivial) entries (in Fig. 4) used in this reflective API.

### 2.3.1 Symbols

**TYPEsym** (`len_Nat SymbolInfo`). A symbol for a type. The type is described in a `SymbolInfo`. Used for instance in method arguments.

**ALIASsym** (`len_Nat SymbolInfo`). An alias for a type. for instance

```
| type Scientist = Person
```

`SymbolInfo`, in `info_Ref` contains the reference to the entry for `Person` in the symbol table.

**CLASSsym** (`len_Nat SymbolInfo [thistype_Ref]`). A class element. This can represent in Scala either a class or a trait. The difference can be made by looking at the `flags_Nat` value in `SymbolInfo`.

```

ScalaSig = "ScalaSig" Version Symtab
Version  = Major_Nat Minor_Nat
Symtab   = numberOfEntries_Nat {Entry}
Entry    = TERMNAME len_Nat NameInfo
          | TYPENAME len_Nat NameInfo
          | NONEsym len_Nat
          | TYPEsym len_Nat SymbolInfo
          | ALIASsym len_Nat SymbolInfo
          | CLASSsym len_Nat SymbolInfo [thistype_Ref]
          | MODULEsym len_Nat SymbolInfo
          | VALsym len_Nat SymbolInfo [alias_Ref]
          | EXTref len_Nat name_Ref [owner_Ref]
          | EXTMODCLASSref len_Nat name_Ref [owner_Ref]
          | NOtpe len_Nat
          | NOPREFIXtpe len_Nat
          | THISTpe len_Nat sym_Ref
          | SINGLEtpe len_Nat type_Ref sym_Ref
          | CONSTANTtpe len_Nat type_Ref constant_Ref
          | TYPEREFtpe len_Nat type_Ref sym_Ref {targ_Ref}
          | TYPEBOUNDStpe len_Nat tpe_Ref tpe_Ref
          | REFINEDtpe len_Nat classsym_Ref {tpe_Ref}
          | CLASSINF0tpe len_Nat classsym_Ref {tpe_Ref}
          | METHODtpe len_Nat tpe_Ref {tpe_Ref}
          | POLYTtpe len_Nat tpe_Ref {sym_Ref}
          | IMPLICITMETHODtpe len_Nat tpe_Ref {tpe_Ref}
          | ...
SymbolInfo = name_Ref owner_Ref flags_Nat [privateWithin_Ref] info_Ref
NameInfo   = <character sequence of length len_Nat in Utf8 format>
NumInfo    = <len_Nat-byte signed number in big endian format>

```

Figure 4: The ScalaSig attribute definition. A more complete definition can be found in the `scala.tools.nsc.symtab.PickleFormat` source file. `_Ref` are references, stored as a `Nat`, to another table entry; `Nat` or `_Nat` are natural numbers, represented as a list of bytes ending with the byte `0x80`.

**MODULEsym** (`len_Nat SymbolInfo`). A module, or object. A module is the equivalent as a Java package, but can contain methods. Objects in Scala are actually modules.

**VALsym** (`len_Nat SymbolInfo [alias_Ref]`). A method symbol.

It is called `VALsym` since `val` and `var` declarations (inside a class) are mapped to methods:

```

| val finalVariable : String = "Hello"
| var anotherVar : Int = 9

```

creates the following methods:

```

| def finalVariable() : String
| def anotherVar() : Int
| def anotherVar_Seq(Int) : Unit

```

We will discuss more about the way our API interprets this later (Sec. 3.4).

**EXTref** (`len_Nat name_Ref [owner_Ref]`) (as well as **EXTMODCLASSref**) refers to a class or module defined in another `.class` file. Here we only give its name, and its owner.

### 2.3.2 Types

**SINGLEtpe** (`len_Nat type_Ref sym_Ref`). A type, or part of a type.

- `type_Ref` refers to the parent type (or prefix),
- `sym_Ref` is the type symbol.

For instance, given the definition `SINGLEtpe(prefix, symbol), java.lang.String`, can be seen as:  
`SINGLEtpe(SINGLEtpe(SINGLEtpe(java), lang), String)`.

**TYPEREftpe** (`len_Nat type_Ref sym_Ref {targ_Ref}`). Refers to a type similar to **SINGLEtpe**, except that it accepts types parameters.

**POLYtpe** (`len_Nat tpe_Ref {sym_Ref}`). A type accepting type parameters.

- `tpe_Ref` is a result type,
- `{sym_Ref}` is an optional list of type parameters.

if `sym_Ref` is present, then `tpe_Ref` must be either a **METHODtpe** or a **CLASSINF0tpe**. if not, then this **POLYtpe** is an eval-by-name type.

**METHODtpe** (`len_Nat tpe_Ref {tpe_Ref}`). A type for a method.

- `tpe_Ref` is the result type,
- `{tpe_Ref}` is an optional list of parameters for each of the method's arguments.

## 2.4 Pickled informations: problems, limitations and work-arounds

**Infinite loops** For some data, following the basic description leads to infinite loops. For instance, the pickled information for this kind of code:

```
| def printSomething[A](something: A) = println(something)
```

The pickled information for the method `printSomething` contains the list of arguments. This list of arguments includes a reference to the argument `something` of type `A`. However, the **TYPEsym** describing the `something`'s type, makes reference, in its **SymbolInfo** to the **VALsym** of `printSomething`. This special case must be handled properly.

Here is a simplified example of the symbol table for this case:

	index			
definition	$x$	METHODsym	tpe_Ref	{tpe_Ref}
content		<i>printSomething</i>	ref to Unit	$y$
definition	$y$	POLYtpe	tpe_Ref	{Sym_Ref}
content		$A$	ref to Unit	ref to $x.A$

We see that the symbol at table index  $x$  contains a reference to the one at index  $y$ , and vice versa.

To break this loop, the `Pickler` uses objects called `DeferredSymbol` and `DeferredType`. They both holds only a reference to the position of the symbol they represent in the symbol table: we will note `DeferredSymbol(p)` for the symbol stored at position  $p$  in the symbol table. When we construct some symbols or types, known to produce some of these loops, we return instead an instance of `DeferredSymbol` to replace a `Symbol`, or `DeferredType` to replace a `Type`. Once we have built all the other symbols and types, we go through this list again, and replace these deferred instances by the actual symbol or type that we can instantiate now.

For our previous example, this means that when we first read in the symbol table the method `printSomething`, we create a `MethodSymbol` with a `DeferredType(p)` as type. Later, when going through the list of symbols, we find that the `MethodSymbol` for `printSomething` has a `DeferredType` in it's field `type`. At this time, the object `MethodSymbol` is already built and can be used in a `Type`: we can read again the definition in the symbol table at the position  $p$ . We just eliminated the loop.

To simplify, after the first pass, our symbol list looks like this:

```
| MethodSymbol("printSomething", SingleType("Unit"), DeferedType(y) :: Nil)
```

And we have stored that together with the index  $x$ . During the second pass, we find a `DeferredType`, and replace this object by a new one, with something like:

```
val x = MethodSymbol("printSomething", SingleType("Unit"),
  DeferedType(y) :: Nil)
val y = fixDefferedT(y) // retrieve the real type at position y.
  // basically: PolyType("A", SingleType("Unit"), x :: Nil)
x.parameters = y :: Nil
```

Now, both objects  $x$  and  $y$  are referring to each other. We have reconstructed the loop.

**Supertypes missing** When classes, objects or traits are pickled, there seems to be no reference to their supertype.

One workaround for this would be to use the Java method `getSuperclass` (in `java.lang.Class`), to get a `Class` instance, then use our API to reflect the given class.

Another one would be to modify the compiler's pickler, to include that information, which seems to be the best approach since we could for instance extend types only existing in the Scala program, that are converted by the bytecode generator (for instance a type aliases, or more simply `AnyRef` - converted to `Object` in bytecode).



class. If it does contains such an attribute, we are really reading a Scala class. Both situations are handled in a different manner starting from now.

**Java class case** If we are accessing a Java class, we still want to be able to use our reflection system on it. But there is no need of the class file for that: we wrap the informations we can get using the standard Java reflection library (such as a list of methods, or enclosed classes), inside a `JavaClassMirror`, which extends a `ClassMirror` trait (see Fig. 1). Therefore we can use reflection over a Java object as if it was a Scala one, using the Scala mirroring API. See Sec. 3.3.4 for more details on `JavaClassMirror`.

**Scala class case** More interesting things happens in case of a Scala class.

We have seen in Sec. 2.2 how `ScalaSig` attributes are read and written: pickled Informations are stored linearly, and we must reconstruct the symbols' hierarchy. The only information we have is that every symbol present in the symbol table, and that contains a `SymbolInfo` (plus few others) have a parent information.

The object `UnPickler`, and its inner class `UnPickle`, handle the symbol's decoding of the `ScalaSig` attribute. They return a flat list of symbols.

Then, the class `Reflector` recreates the hierarchy from the symbol list, and generate the final mirrors seen on Fig. 1.

This job is not that trivial, since information arrives sequentially, and potentially in any order (the parent may arrive later than the child in the symbol list).

Another problem is that in the API, a parent may consist of a list of children (e.g. a `ClassMirror` containing a list of `MethodMirror`), but every child must also contain a reference to its parent. And because we want to prevent further modification on our instances from the user, we must avoid the use of mutable variables, and therefore we can not create final mirrors directly. The `Reflector` uses temporary mirrors to build the hierarchy. Once this is done, we convert all temporary mirrors to the mirrors the user will actually be using, and with Scala mixins, we can update parents and children afterwards. Here is an example for a `TraitMirror` :

```
trait ClassMirror extends Mirror {
  def fields : List[FieldMirror]
  def methods : List[MethodMirror]
  // ...
}

trait EditableClassMirror extends ClassMirror {
  var _parent : Mirror = NoMirror
  var _fields : List[FieldMirror] = Nil
  var _methods : List[MethodMirror] = Nil
  def parent = _parent
  def fields = _fields
  def methods = _methods
  // ...
}

trait TraitMirror extends ClassMirror
```

```

object TraitMirror {
  def apply(_name : String) = new TraitMirror with EditableClassMirror {
    def name = _name
  }
  // ...
}

```

Then, we can construct a `TraitMirror` using the companion object `TraitMirror`, and consequently having a reference to it to later construct its methods and fields.

The user doesn't see that its trait extends `EditableClassMirror`, and doesn't have access to its underlying constructs.

### 3 Reflecting Scala

In this part we will describe the API itself and the role of each data structure.

#### 3.1 The Reflect object: main entry point

One idea of mirrors, is that meta-level facilities are not accessed directly from an object or a class. Instead meta programming facilities are exposed by mirrors. To obtain a `Mirror`, one need a global factory, that can be accessed from anywhere. This allows different implementations of the reflecting API, sharing the same interface. This is the first statement, *encapsulation*, saying that meta level facilities should encapsulate their implementation.

In our library, the `Reflect` object is such a factory. It is the entry point for all kind of reflection. It's goal is to create the mirrors for different types of data.

In our implementation, it is also in charge of controlling the work flow described on Fig. 5: retrieving the actual `.class` file, using a `ClassReader` and `UnPickler` to read and reconstruct the class structure, and finally producing the corresponding `ClassMirror` or `InstanceMirror` depending on what type of information was given to him.

We will not list every detail of implementation here, but rather explain the interface.

The first method is an *apply* method:

```

1 object Reflect {
2   def apply(a: AnyRef) : InstanceMirror = {
3     val clazz = a.getClass
4     val mirror = classMirror(clazz)
5
6     val r = if (mirror.isInstanceOf[ObjectMirror]) {
7       mirror.asInstanceOf[ObjectMirror]
8     } else if (mirror.isInstanceOf[ClassMirror]) {
9       InstanceMirror(a, mirror.asInstanceOf[ClassMirror])
10    } else {
11      null // should never happend.
12    }
13    r
14  }

```

It is used to reflect an instance, as in `Reflect(new Person("Sir", "Isaac Newton"))`. We get in return an `InstanceMirror`. The code retrieves the Java `Class` object (line 3), and gets a `ClassMirror` out of it (line 4). Then, an `InstanceMirror` is created out of this `ClassMirror` and the initial instance we are mirroring (note that `ObjectMirror` extends `InstanceMirror`).

The user also has the option of asking for a mirror from a `Class` instance. This can be useful if he is using some Java libraries, using `Class` objects.

```

def classMirror(clazz : Class) : ClassMirror = {
  // build the right ClassMirror from the .class file
}

def classMirror(s: String) : ClassMirror = {
  val clazz = retrieveClass(s)
  classMirror(clazz)
}

```

Implementation details are not really interesting here. These two methods are equivalent, one using a string to get a `Class` object, the other one taking directly a `Class` object.

The problem using this method, is that we retrieve a `ClassMirror`, and that a `ClassMirror` is not the best object to work with for some operations like calling methods (listing methods is supported though), because different implementations of `ClassMirror` return different types of `MethodMirrors`. Therefore, there exists four other methods, basically doing a cast operation for us, that we can use if we are sure of the kind of object we try to reflect:

```

def objectMirror(a: AnyRef) : ObjectMirror = {
  apply(a).asInstanceOf[ObjectMirror]
}

def objectMirror(s: String) : ObjectMirror = {
  classMirror(s).asInstanceOf[ObjectMirror]
}

def instantiableMirror(s: String) : InstantiableClassMirror = {
  classMirror(s).asInstanceOf[InstantiableClassMirror]
}

def instantiableMirror(c: Class) : InstantiableClassMirror = {
  classMirror(c).asInstanceOf[InstantiableClassMirror]
}
// ...
}

```

## 3.2 Mirror

`Mirror` is the base class for every mirroring class. It's a really simple trait:

```

trait Mirror {
  def name : String
  def parent : Mirror
  // ...
}

```

Scala, inspired by Beta [MMP89], allows everything to be nested, therefore every `Mirror` has a parent (since any kind of structures can be inside another one). The parent is the mirror containing this mirror. For instance, a `MethodMirror` has as parent the `ClassMirror` defining it.

For a top level element, the parent is set to a `NoMirror` object, defined as:

```
| object NoMirror extends Mirror {
|   def name = "<NoMirror>"
|   def parent = NoMirror
| }
```

### 3.3 Classes, objects and traits

`ClassMirror` together with `MethodMirror`, is the most important `Mirror` implementation, since the `Reflect` object always return a `ClassMirror`, either directly, either through an `InstanceMirror`.

The `ClassMirror` is defined as follow:

```
| trait ClassMirror extends Mirror {
|   def fields : List[FieldMirror]
|   def methods : List[MethodMirror]
|   def aliases : List[AliasMirror]
|   def nestedClasses : List[ClassMirror]
|
|   lazy val clazz = Reflect.retrieveClass(javaString)
|
|   def friendlyString: String = List(this.toString + " {",
|     " " + (fields ::: methods ::: aliases ::: nestedClasses).mkString("\n "), "}")
|     .mkString("\n")
| }
```

The important point is the `clazz` field: the method `Reflect.retrieveClass` return a (Java) `Class` instance. This instance is used later to call a method on a particular instance, using the Java reflection.

This abstract class is derived into 3 particular implementations: `TraitMirror`, `ObjectMirror`, and `InstantiableClassMirror`.

#### 3.3.1 Trait mirror

`TraitMirror` is only a basic `ClassMirror`, and is mirroring a `trait` structure.

```
| trait TraitMirror extends ClassMirror
```

It is instantiable through the companion object `TraitMirror`.

#### 3.3.2 Instantiable class mirror

An `InstantiableClassMirror` mirrors any class that can be instantiated. Hence, it has a list of constructors, retrieved from the `Class` instance.

```

trait InstantiableClassMirror extends ClassMirror {
  lazy val constructors = clazz.getConstructors()

  def apply(args: Any*) : AnyRef = // ...
}

```

The `apply` method choose the right constructor to call, by using the method `isInstantiableFrom`, which gives some liberty to the user, and returns a new object instantiated with the given list of arguments.

### 3.3.3 Object mirror

`ObjectMirror` represents a Scala object, i.e. a singleton class. It is defined as follows:

```

trait ObjectMirror extends InstanceMirror with ClassMirror {
  override def method(name: String) : OverridenMethodMirror =
    OverridenMethodMirror(methods.filter(_.name.equals(name)))
  // ...
}

```

Here, the overridden method `method` comes from the trait `InstanceMirror` described in Sec. 3.7.

It's interesting to study the companion object for `ObjectMirror`:

```

1 object ObjectMirror {
2   def apply(_name: String) = new ObjectMirror with EditableClassMirror {
3     def name = _name
4     def ref = this
5     def classMirror = this
6
7     override def methods : List[InstantiatedMethodMirror] = object_methods
8
9     lazy val object_methods = _methods.map(m => InstantiatedMethodMirror
10      (
11        m.name, m.parent, this, m.returnType, m.parametersType, m.
12          typeParameters))
13   }
14   def unapply(o: ObjectMirror) = Some((o.name, o.parent, o.fields, o.methods,
15     o.aliases, o.nestedClasses))
16 }

```

The `methods` method on line 7, inherited from `ClassMirror` is overridden to return a list of `InstantiatedMethodMirror` instead of generic `MethodMirrors`.

The returned list is defined by the lazy value `object_methods` on line 9 and 10, which constructs, for each method, an `InstantiatedMethodMirror` with as instance `this` (all the other parameters are the same).

Here is a simple example:

```

object MainApp extends Application {
  def printAPerson(person : Person) = println("Hi " + person.name)
  def square(value : Int) = value * value
  // ...
}

```

that can be used as follows:

```
val reflect = Reflect.objectMirror("testpackage.MainApp")
reflect("printAPerson")(new Person("Mr", "Haskell Brooks Curry"))
println("5^2 = " + reflect("square")(5))
```

this will print:

```
Hi Haskell Brooks Curry
5^2 = 25
```

### 3.3.4 Java class mirror

A `JavaClassMirror` is a kind of wrapper to play with Java reflection through our mirroring API. It's a subclass of `InstantiableClassMirror`:

```
trait JavaClassMirror extends InstantiableClassMirror {
  def nestedClasses = (for (val v <- clazz.getDeclaredClasses) yield
    JavaClassMirror(v)).toList
  def methods : List[ClassMethodMirror] =
    (for (val m <- clazz.getDeclaredMethods) yield {
      ClassMethodMirror(m.getName, this, JavaClassMirror.buildTypeRef(m.
        getReturnType.getName),
        m.getParameterTypes.map(c => JavaClassMirror.buildTypeRef(c.
          getName)).toList)
    }).toList
  // ...
}
```

The most important points are the methods `nestedClasses` and `methods`, inherited from `InstantiableClassMirror`. The first one gives a list of nested classes, all being obviously other `JavaClassMirrors` (it is not really possible to use Scala classes from Java); the second one gives the list of methods: for each method, a `ClassMethodMirror` is created using type parameters, and the return type of the underlying Java method.

Here is an example of use:

```
val f = new File("build.xml")
val jc = Reflect(f) // this is a JavaClassMirror
val exists = jc("exists")()
```

## 3.4 FieldMirror: Val and Var mirrors

*Note: this section 3.4 is not currently implemented, but we describe our plan.*

Values and variables are specially handled in Scala. The code:

```
val finalVariable : String = "Hello"
var anotherVar : Int = 9
```

is desugared to

```

| def finalVariable() : String
| def anotherVar() : Int
| def anotherVar_$eq(Int) : Unit

```

i.e. one accessor in case of a `val`, and two (a “getter” and a “setter”) in case of a `var`.

So, using normal reflection, we would find the methods `finalVariable()`, `anotherVar()` and `anotherVar_$eq(Int)`. However, we still want the user to see them as `val` and `var` in order to preserve ontological correspondence. That’s why in 1 we observe an abstract class `FieldMirror`, and two implementations `VarMirror` and `ValMirror`.

When reflecting a class, the reflection API detects those cases, and instead of creating the methods as `MethodMirror`, they are mapped to `FieldMirrors` and included into the `field` list of their class.

### 3.5 Dealing with types

While we are mirroring Scala data structures, we often need to represent types. Types are used for example in methods, for arguments. We do not use the term of “mirror” when we are referring to these types, given that in that context, they are an abstract concept, and are very different from the notion of class, object or trait. This distinction does not exist in Java: the class `Class` is used for both purposes. This also gives us more flexibility, as we can define a type representing a class with a given set of methods, without actually having to construct a “virtual” `ClassMirror`, like for structural types.

Types are defined by a very simple trait:

```

| trait Type {
|   def javaString : String
| }

```

which is extended for different types, basically following each different pickled types (described in Sec. 2.3.2). Ideally, these types should follow the type hierarchy defined by the Scala specification. However, we would need more time to fully understand and generate the mapping from pickled data to a more appropriate Scala types hierarchy.

We make use of the method `javaString` when we need to retrieve a Java `Class` object, with the intention of using Java reflection to call a method for instance. This method returns a `String` that can be recognized by the Java library and find the appropriate `.class` file.

#### 3.5.1 Type mirrors

Type mirrors are used in type parameters for methods<sup>2</sup>. They only include a string, since a type parameter is only a name:

```

| def m[A](a : A) = println(a)

```

---

<sup>2</sup>We could not obtain type parameters for classes

The method *m* has a type parameter *A*. It's represented by a `TypeMirror("A", owner)`.

We must understand the difference between a `TypeMirror` and a `Type`. `Types` are used where we refer to “real” types, they do not extend `Mirror`. A `Mirror` have a parent, or owner, like this `TypeMirror`, where a `Type` is only referring to an abstract concept, which can not be mirrored.

### 3.5.2 Alias mirrors

Scala gives the user the ability to define types aliases, for instance:

```
| class Person  
| type Scientist = Person
```

In this case, a `Scientist` is exactly the same as a `Person`. A `Person` can be assigned to a `Scientist`, as well as the other way around, and we can obviously use this type alias in the same way we use a classical type:

```
| val s : Scientist = new Person  
| val p : Person = new Scientist  
| def doScientificStuff(s : Scientist) : Long = // do some crazy calculations, and  
|   return a big number.
```

By looking at the bytecode only (as with the Java reflection), the method `doScientificStuff` takes as argument a `Person`, and, according to the Java reflection, the signature for `doScientificStuff` is `(Person) => Long`. This behavior doesn't follow the concept of “ontological correspondence”.

Fortunately, in the pickled informations, the reference to `Scientist` remains, and the signature is `(Scientist) => scala.Predef.Long`.

An `AliasMirror` is only defined by a name, and a `type`: the name, in our example is the String `"Scientist"`, and the type refers to the type `Person`. `AliasMirrors` can be found into `ClassMirrors`.

```
| trait AliasMirror extends Mirror {  
|   def tpe : Type  
| }
```

A companion object is also defined.

The difference between the Scala pickled informations and the Java bytecode requires some attention when actually calling a method through Java reflection. Each time we need a class, we try to load the Java class using the name given to us. If this fails, we check if this type is actually an `AliasMirror`. In this case, we load and reconstruct the class containing this `AliasMirror`, and use its `tpe` attribute as a new class to try to load. We have to do that recursively, since one `AliasMirror` can be an alias for another `AliasMirror`.

## 3.6 Method mirrors

`MethodMirrors` are the type of mirror representing a method. Typical information contained in a `MethodMirror` is: return type, parameters' type, and type parameters.

```

trait MethodMirror extends Mirror {
  def returnType : Type
  def parametersType: List[Type]
  def typeParameters: List[TypeMirror]

  lazy val method =
    if (parent.isInstanceOf[ClassMirror]) {
      parent.asInstanceOf[ClassMirror].clazz.getDeclaredMethod(name,
        Reflect.retrieveClass(parametersType, typeParameters).toArray)
    } else {
      null
    }

  // ...
}

```

During the compilation process, type parameters, as well as Java generics, are converted to `java.lang.Object`:

```

def printSomething[A](smtg : A) = println("Hello object " + smtg.toString)

```

is found in the Java bytecode as a method (`Object`)  $\Rightarrow$  `void`. But once again, original type parameters can be retrieved thanks to the pickled information.

Note that the `MethodMirror` does not have any `apply` method. Neither does it have a companion object, and consequently cannot be instantiated. Instances are delegated to more specific objects, with different `apply` methods.

### 3.6.1 Class method mirrors

A `ClassMethodMirror` is in fact a function, accepting as input any instance, and returning an `InstantiatedMethodMirror` (see Sec. 3.6.2). It is simply defined as follows:

```

trait ClassMethodMirror extends MethodMirror with (AnyRef =>
  InstantiatedMethodMirror) {
  def apply(ref: AnyRef) = InstantiatedMethodMirror(name, parent, ref,
    returnType, parametersType, typeParameters)
}

```

A `ClassMethodMirror` is supposed to be used when we are working on a `ClassMirror` only, and not on an `InstanceMirror` or `ObjectMirror`, since their methods always return a list of `InstantiatedMethodMirrors`.

### 3.6.2 Instantiated method mirror

An instantiated method is a method which already knows the instance on which it will be invoked. For instance, a method contained in an object is always invoked from its unique owner (which is a singleton). Therefore, we want to skip the step of defining this instance each time we call the method. An `InstantiatedMethodMirror` stores the instance on which it will be applied when the method `method` is called.

```

trait InstantiatedMethodMirror extends MethodMirror {
  def instance : AnyRef
  def apply(args: Any*) : Any = applySeq(args.map(x => x))
  def apply() : Any = method.invoke(instance, null)
  def applySeq(args: Seq[Any]) : Any = method.invoke(instance, Reflect.
    convertArgs(args).toArray)
}

```

There exists a companion object for `InstantiatedMethodMirror`, thus, it can be instantiated.

### 3.6.3 Overriden method mirror

An `OverridenMethodMirror` is a utility class. It is used to encapsulate different methods with the same name, but with different arguments list.

The code for this trait is a little bit more tricky.

```

1 trait OverridenMethodMirror {
2   def methods : List[InstantiatedMethodMirror]
3
4   def apply(args: Any*) : Any = {
5     val argsClasses = Reflect.retrieveClass(args).toList
6     val methodsOk = methods.filter(m =>
7       Reflect.retrieveClass(m.parametersType, m.typeParameters).length ==
8         args.length &&
9       {
10        val zipped = Reflect.retrieveClass(m.parametersType, m.
11          typeParameters).toList.zip(argsClasses)
12        zipped.forall(pair => pair._1.isAssignableFrom(pair._2))
13      })
14     if (methodsOk.length != 1) {
15       throw new NoSuchElementException("'" + methodsOk.length + "
16         methods found.")
17     }
18     methodsOk(0).applySeq(args.map(x => x))
19   }
20
21   def apply() : Any = // similar to the other apply method, but without
22     parameters.
23
24   //...
25 }

```

The `apply` method takes a list of any elements, and retrieves one `Java Class` instance for each of them. Then, on line 6, we filter the list of methods to keep only those who have the same number of arguments, and with argument types compatibles with the ones passed to the `apply` method.

On line 12, if we end up with only one method, everything went fine, and we can invoke this method (note that we need to unbox each argument first) (line 15).

If we have no methods left, this means that the user made a mistake, and that it is not possible to call any method of this name with the given set of arguments.

However, if we end up with more than one method, we cannot decide which one to call. This will happen in this kind of situation:

```
| def m(s: String, a: AnyRef) = // ...  
| def m(a: AnyRef, s: String) = // ...
```

Which one should be called if the user passes two `Strings`? We can't decide. In Java, this case can't happen, since the Java reflection will not find any method taking two `Strings` as arguments. So the user has to choose which one he actually wants by requesting the *exact* same type (`Class` objects must be *equal*).

Basically, the Java reflection will throw a `NoSuchMethodException` in this case:

```
| public class C {  
|     public static void m(Object o) { /* ... */ }  
| }
```

then:

```
| Class str = Class.forName("java.lang.String");  
| Class c = Class.forName("C");  
| Method m = c.getMethod("m", str); // this throws a NoSuchMethodException !
```

There is *no* method named `m`, taking a `String` as argument; only one taking an `Object`.

Our reflection API uses instead the method `isAssignableFrom`, which returns `true` if the argument can be assigned to the `Class` it is called on. This is more permissive than Java. The drawback is that we can have more than one method corresponding to a given list of `Classes`.

However, this choice has been made because this class, `OverridenMethodMirror`, is intended to be used as a simplification for selecting the right method to call. In ambiguous cases, the user still has the possibility to choose the correct method to call, by using a filter on the method list. In this case, the complexity is the same as Java reflection.

Here is an example of the use of `OverridenMethodMirror`:

```
| object MainApp extends Application {  
|     def printAPerson(person : Person) = println("Hello " + person.name)  
|     def printAPerson(person : Person, greet : String) = println(greet + person.  
|         name)  
|     // ...  
| }  
  
| val reflect = Reflect.objectMirror("testpackage.MainApp")  
| val m = reflect("printAPerson")  
| m(new Person("Sir", "Isaac Newton"))  
| m(new Person("Mr", "Alan Turing"), "Good morning")
```

this will output:

```
Hello Sir Isaac Newton  
Good morning Mr Alan Turing
```

As a summary of how the user can actually use this hierarchy of `ClassMirrors` and `MethodMirrors`, the Fig. 6 shows a diagram of each method mirror, and class mirror, and what every method returns. This can help you to find your way from a `ClassMirror` all the way to the result of a method invocation.

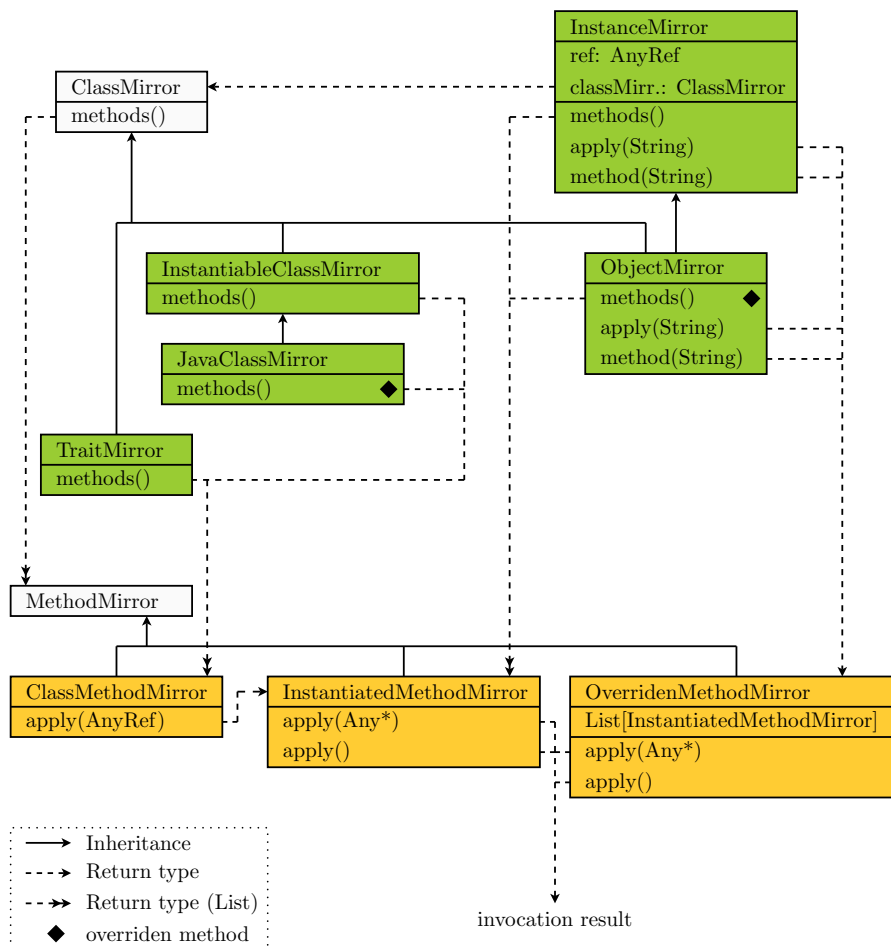


Figure 6: Interaction between `ClassMirrors` and `MethodMirrors`

### 3.7 Instance mirrors

Instance mirrors are used to reflect an instance, and not the underlying data structure. It is the mirror for a particular instance of a given class. There is no equivalent construction in the current Java metaprogramming API.

Let's see how it is implemented:

```

trait InstanceMirror {
  def classMirror : ClassMirror
  def ref : AnyRef
  def methods : List[InstantiatedMethodMirror] = cached_methods
}
  
```

```

def method(name: String) : OverridenMethodMirror =
  OverridenMethodMirror(methods.filter(_.name.equals(name)))

def apply(name: String) : OverridenMethodMirror = method(name)

lazy val cached_methods = classMirror.methods.map(m =>
  InstantiatedMethodMirror(
    m.name, m.parent, ref, m.returnType, m.parametersType, m.
      typeParameters))

  // ...
}

```

As you can see, an `InstanceMirror` has a reference to a `ClassMirror`, as well as a reference to a particular instance: the mirrored instance.

Once we get an `InstanceMirror`, we can freely access the underlying structure through the `ClassMirror`, and get a list of methods, fields, aliases, or inner classes. This is a classical way of doing it: get an `InstanceMirror`, and access that information through its class mirror.

However, if we want to call a method, this way is not really convenient. Remember, a method in a `ClassMirror` does not know on which instance it must be called, you must specify it before. The `InstanceMirror` trait gives a way to avoid this. Since the instance is already known, it has a few helpful methods:

- one method `methods`, returning a list of `InstantiatedMethodMirror`, that you can invoke only by specifying the arguments;
- one method `method(String)`, which returns an `OverridenMethodMirror`;
- and one `apply` method, which is an alias for `method`

This class gives us the ability to do something like that:

```

val newton = new Person("Sir", "Isaac", "Newton")
Reflect(newton)("greetings")("Hello")

```

which prints:

```

| Hello Sir Isaac Newton

```

`Reflect(newton)` returns an `InstanceMirror`, on which we call the method `apply`, returning an `OverridenMethodMirror`, on which again, we call the method `apply` with the argument `"Hello"`. The `greetings` method in `Person`, prints the argument, followed by the title and the person's name.

## 4 Reflecting on Scala reflection

During this project, we faced several difficulties. The first one was to rewrite a new `UnPickler` independent from the Scala compiler: the later actually includes one `UnPickler`, but it is closely linked to the compiler structure, and makes use

of a lot of facilities offered by the compiler itself. Using this `UnPickler` would mean loading the entire compiler when reflection is used, which is a pretty heavyweight program. This option being not entirely satisfactory, we chose to rewrite an independent `UnPickler`.

Mapping pickled informations to actual data structures and types was the second challenge. The first thing we had to do for this was to reverse-engineer the exact meaning of each tag of the symbol grammar (see Fig. 3). This step is not as straightforward as we could think: a method for instance can be represented in several ways in the pickled data. For this, we write a lot of examples, we compile them, look at the resulting `ScalaSig` signature, make some modifications, and start again to observe the differences. Using this technique gave us enough informations on the meaning of some tags useful to us, but it didn't give us some answers: some points are still obscure.

Then we had to rewrite a hierarchy for types and symbols, close enough to the pickled information, but having already all the information we need for later use. It would serve as an intermediate hierarchy between the pickled information, and the mirrors. This has been done through the `Symbols`. The same thing was necessary for types. Then, given that pickled information, and symbols, were still presented in a "flat" way (basically, we retrieve from the `UnPickler` a flat list of `Symbols`), we needed to reconstruct the hierarchical structure of a class. This is done with the `Reflector` object. Again, deferred initialization and mixins had to be used to construct the final `ClassMirrors` that could be used without risk by the user.

The last point, was to construct a usable and developer friendly, yet structured and powerful API, following the principles of mirrors. This API is entirely contained in its own implementation, and corresponds to the Scala language ontology.

The API as it is at the end of this project is usable, and almost ready to be shipped. But before that, here is a list of known bugs or missing features that we would like to implement:

**Type parameters for classes** were not found in the pickled informations. This is mandatory for some classes.

**Case classes** are compiled in a special way, and pickled data turns out to be special too. There seems to be a method with the same name as the class in the module (package) containing the case class. This is like a top level method. We need more exploration for this case, and even if few code has to be written to support this, it has not been done yet.

**Java interfaces** In the current implementation, `JavaClassMirror` is used to reflect both Java classes and interfaces, this doesn't cause any problem, it works for interfaces too: no distinction exists in `java.lang.reflect` API between classes and interfaces anyway, instead a method `isInterface` is present in the `Class` class. However, it would be good to use a `JavaInterfaceMirror`.

**Tags** indicating for example whether a method is private, public or protected, are read, but not included in the final mirror. This should be easy to implement. We would also need some helper methods to access this information.

**Annotations** are parsed by the `UnPickler`. However, they are not mirrored yet.

**Partial functions** We could use partial functions at some places, like in the case of a `ClassMethodMirror`: it has a method *apply*, taking `AnyRef`, and returning an `InstantiatedMethodMirror`. We could make this `ClassMethodMirror` a partial function, defined only for instances extending the correct class or trait.

**Caching** When we reflect a class, we always call the `ClassLoader`, which returns the same `.class`, but we unpickle every single time all the information. We could implement a caching mechanism.

## Acknowledgement

I would like to thank Martin Odersky for suggesting me this really interesting project and Gilles Dubochet whose help, suggestions and excellent Scala knowledge helped me.

Iulian Dragos and Stéphane Micheloud also helped me in different parts of my work.

Philippe for a final last minute review of this document.

## References

- [BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, New York, NY, USA, 2004. ACM.
- [GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.
- [LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [MMP89] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM.
- [Oa04] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

## List of Figures

1	Mirrors API . . . . .	4
2	Pickler . . . . .	5
3	Pickle format . . . . .	6
4	ScalaSig attribute definition . . . . .	7
5	Mirrors API . . . . .	10
6	Class and methods mirrors . . . . .	22

## Index

- .class, 10
- accessor, 17
- aliases, 23
- AliasMirror, 18
- ALIASsym, 6
- Annotations, 25
- API, 11, 12
- apply method, 3
- attribute, 6, 10
- attributes tables, 5
  
- Beta, 14
- bytecode, 1, 3, 5
  
- Class, 5, 17
- ClassLoader, 10
- ClassMethodMirror, 19
- ClassMirror, 11, 14
- ClassNotFoundException, 3
- ClassReader, 10
- CLASSsym, 6
- compiler, 5
- constructors, 14
  
- data structures, 24
- design principle, 1
  
- EditableClassMirror, 12
- encapsulation, 1, 12
- EXTref, 8
  
- factory, 12
- FieldMirror, 16
- fields, 23
- file, 10
- footprint, 3
- function, 19
  
- generics, 19
  
- hierarchy, 11
  
- infinite loops, 8
- inner classes, 23
- InstanceMirror, 22
- InstantiableClassMirror, 14
- instantiated method, 19
  
- InstantiatedMethodMirror, 19
- invoke, 20
  
- Java, 3
- Java class, 10
- Java interfaces, 24
- Java reflection, 16
- Java virtual machine, 5
- JavaClassMirror, 11, 16
- JavaInterfaceMirror, 24
- javaString, 17
  
- Mapping, 24
- meta-programming, 1
- meta-programming facilities, 3
- method, 18, 20
- method invocation, 22
- MethodMirror, 11, 18
- methods, 23
- METHODtpe, 8
- Mirror, 12–14
- mirror based, 1
- mirrors, 3, 12
- mixins, 11
- MODULEsym, 7
  
- nested, 14
- non-reflective program, 3
  
- ObjectMirror, 14, 15
- ontological correspondence, 1, 3, 17, 18
- ontology, 1
- OverridenMethodMirror, 20
  
- parameters, 18
- parent, 14
- Person, 4
- Pickle format, 6
- Pickled data, 5
- pickled data, 6
- pickled information, 6, 18
- pickled Informations, 11
- POLYtpe, 8
- private, 24
- protected, 24
- public, 24
  
- Reflect, 12, 14

- reflection process, 10
- Reflector, 11, 24
- retrieveClass, 14
- return type, 18
  
- Scala, 1, 3
- Scala class, 11
- Scala compiler, 23
- Scala data structures, 17
- ScalaSig, 6, 10, 11
- singleton, 15
- SINGLEtpe, 8
- stratification, 1
- structural types, 17
- supertype, 9
- symbol, 6
- symbol grammar, 24
- Symbol table, 6
- symbol table, 8, 11
- SymbolInfo, 11
  
- tag, 24
- Tags, 24
- temporary mirrors, 11
- TraitMirror, 14
- Type, 18
- type, 18
- type parameters, 17, 18
- TypeMirror, 17, 18
- TYPEREFTpe, 8
- types, 17
- types alias, 18
- TYPEsym, 6
  
- UnPickle, 11
- UnPickler, 11, 23
  
- val, 16
- ValMirror, 17
- VALsym, 7
- var, 16
- VarMirror, 17