Scala, un tutorial pratico

Parte 1.

Introduzione

Diamo il via ad una serie di tutorial sul linguaggio di programmazione Scala. Non che ce ne fosse bisogno ma ve lo beccate comunque. Diamo un taglio pratico alla faccenda perchè se stiamo qui a menarcela con le funzioni di ordine superiore, il type system e compagnia bella finiamo per stracciarci i marroni dopo due pagine.

Cos'è scala.

Scala è un linguaggio di programmazione e un'estensione per la piattaforma Java. I programmi scala sono a conti fatti programmi per la piattaforma Java che dipendono da un'estensione (scala-library.jar) da aggiungere alle librerie richieste dall'applicazione sviluppata. E' possibile usare le librerie della piattaforma Java in Scala e usare le librerie della piattaforma Scala in Java, anche se la seconda opzione deve fare i conti con un po' di arzigogoli di denominazione (ad esempio in Scala gli identificatori possono essere composti di caratteri non alfanumerici che sono convenzionalmente trasformati in identificatori Java con varie preposizioni che li fanno apparire strambi). E' comunque più facile che sia un programma Scala ad usare una libreria Java piuttosto che il contrario.

Non serve dire dove potete trovare un JDK, per scala andate sul sito http://www.scala-lang.org

Quanti agli IDE, ce ne sono tre, in forma di plug-in per NetBeans, Eclipse e IntelliJ IDEA. Dei tre il preferito da chi scrive è l'ultimo – è quello che a mio giudizio funziona meglio con Scala – il primo è apparentemente in stand-by, il secondo funzionicchia.

Due parole veloci veloci.

L'organizzazione dei sorgenti funziona grossomodo come in Java. Ci sono package, classi, interfacce che si chiamano trait e sono dei mix-in (cioè possono avere membri dichiarati e definiti e membri solo dichiarati), e object, che sono dei singleton.

```
package x.y.z

class UnaClasse {

}

package a

trait UnTrait {

}

package b

object UnSingleton {

}
```

Le classi possono avere un costrutore primario e un tot di costruttori alternativi. Il costruttore primario è dichiarato inserendo una lista di parametri nella dichiarazione della classe:

```
class UnaClasse(stringa: String, intero: Integer) {

}
```

Qui UnaClasse ha un costruttore primario che richiede una stringa e un intero. I costruttori alternativi sono obbligati a fornire dei valori per il costruttore primario:

```
class UnaClasse(stringa: String, intero: Integer) {

	def this() {//costruttore alternativo
		this(“pippo”, 10)
	}
}
```

In una dichiarazione di membro o variabile il tipo segue il nome:

```
//java
String valore = “hello”

//Scala
var valore: String = “hello”
```

E' possibile omettere il tipo poiché il compilatore inferisce generalmente lo inferisce (applicando il tipo più specifico):

```
var valore = “hello” //valore è uno String
var lista = List(“hello”, 10) //lista è una List[AnyRef]
```

Le variabili, sia membri che locali, sono dichiarate con la parola chiave var:

```
var variabile = “hello”
```

Le costanti, nel senso di riferimenti costanti – i final di Java, sono dichiarate con la parola chiave val:

```
val costante = “hello”
```

I metodi o funzioni membro sono dichiarati usando la parola chiave def:

```
class UnaClasse {
	def metodo(): Unit = {
		System.out.println(“hello world”)
	}
}
```

Il punto di entrata di un programma Scala è un qualsiasi metodo main di un object.

```
object Main {

	def main(args: Array[String]): Unit = {
		System.out.println(“hello world”)
	}
}
```

Unit corrisponde grossomodo a void – grossomodo perchè in Scala Unit è un tipo di dato vero e proprio. I blocchi Scala:

```
{ } //un blocco
```

sono espressioni che restituiscono un valore, precisamente il valore dell'ultima espressione contenuta nel blocco:

```
var x = {
	"hello world"
}
```

Qui x è una variabile di tipo String il cui valore è "hello world".

```
var y = {
	val x = new JFrame("test")
	x
}
```

Qui y è una variabile di tipo JFrame.

In scala le eccezioni sono tutte non controllate, anche quelle che in Java sono controllate.
Il modificatore di accesso predefinito è public: se non c'è niente è public. Altrimenti si possono usare private e protected. I modificatori possono poi essere annotati per generare incroci tipo private[nome package] e via dicendo.
I nomi dei metodi e i nomi dei campi condividono lo stesso spazio: se in una classe c'è un campo che si chiama pippo non può esserci un metodo che si chiama pippo e viceversa. Che a dirla così sembra una stupidata ma salta fuori che, per via di questa condivisione, accessori e mutatori (getter e setter) spariscono.

E potremmo andare avanti per dei giorni ma ci fermiamo qui.

Un programma.

Iniziamo creando un oggetto Scala che salva degli oggetti JavaBean in una cartella predefinita del sistema dell'utente.

```
object BeanStore {

}
```

Un object Scala è un singleton, non serve il new per istanziarlo, ne esiste uno per ogni classloader per ogni jvm. Nella maggior parte dei casi si può dire che ne esista uno e basta. BeanStore ha due metodi, uno che salva un JavaBean e uno che carica un JavaBean. Iniziamo con quello che salva. Serve un bean, ovviamente, e un id. L'id qualifica l'istanza salvata e permette di recuperarla.

```
package beanstore;

object BeanStore {

	def save(id: String, bean: AnyRef) {

	}
}
```

dire:

```
def save(id: String, bean: AnyRef) { … }
```

è un'abbreviazione per:

```
def save(id: String, bean: AnyRef): Unit = { … }
```

In pratica è un metodo void. Per salvare il bean ci serve una cartella in cui salvare un file xml. Usiamo la home dell'utente e ci ficchiamo dentro una cartella ad hoc:

```
package beanstore

import java.io.File

object BeanStore {

    var storeDirectory = {
        val dir = new File(System.getProperty("user.home"), "beanstore")
        if(!dir.exists) dir.mkdir
        dir
    }

    def save(id: String, bean: AnyRef) {

    }
}
```

Quanto vale storeDirectory? Vale quando il valore dell'ultima espressione nel blocco usato per la sua inizializzazione. L'ultima espressione è dir, ergo è un File. Si può scrivere per chiarezza:

```
var storeDirectory: File = { eccetera }
```

Ma essendo tre righe non è che ci si debbano strappare i capelli per arrivarci. Il membro storeDirectory è un var ed è pubblico che sarebbe il classico campo "oh san giuseppe aiutaci" di Java solo che in Scala, per via della condivisione dello spazio dei nomi tra i membri, è sempre possibile sostituire ad una campo un metodo e, tramite le funzioni, ad un metodo un campo. Ergo non gliene frega più niente a nessuno che storeDirectory sia un campo pubblico perchè posso sempre sostituirlo con un metodo senza che questo intacchi il codice utente.

Il metodo save piglia l'id, lo trasforma nel nome di un file che sta nella cartella storeDirectory e lo usa per salvare "bean", che si suppone essere un JavaBean. Stabiliamo che tutti i file abbiano un'estensione convenzionale, dichiarata in un campo di BeanStore:

```
object BeanStore {

        var extension = ".bean.xml"

}
```

Poi creiamo un metodo che data una stringa di identificazione restituisce il file corrispondente:

```
        def idToFile(id: String) = new File(storeDirectory, id + extension)
```

Questa è una contrazione per:

```
        def idToFile(id: String): File = {
                new File(storeDirectory, id + extension)
        }
```

A dirla tutta non è una forma speciale, un'eccezione, ma è l'applicazione della regola per cui un def è un valore costante parametrico, cioè è un nome a cui si applicano dei parametri e che vale quanto una certa espressione eventualmente applicata a quei parametri. Usare il blocco dopo l'uguale significa semplicemente che il valore che si vuole assegnare al def richiede più linee di codice per essere espresso: se di linea ne basta una, se ne usa tranquillamente una senza blocco. E' invece un'eccezione la forma usata per il metodo save, quella senza tipo

restituito ed uguale:

```
def save(...) {

}
```

Questa sì è una contrazione che tiene conto del caso comune dei metodi che non restituiscono nulla. Chiusa parentesi, salviamo il bean. Trasformiamo l'id in un file:

```
def save(id: String, bean: AnyRef) {
    val file = idToFile(id)

}
```

Apriamo uno stream:

```
def save(id: String, bean: AnyRef) {
    val file = idToFile(id)
    val out = new FileOutputStream(file)

}
```

Salviamo il bean usando XMLEncoder:

```
def save(id: String, bean: AnyRef) {
    val file = idToFile(id)
    val out = new FileOutputStream(file)
    val encoder = new XMLEncoder(out)
    encoder.writeObject(bean)
    encoder.flush
    out.close
}
```

Che è quasi quello che si deve fare. Come in Java e per la verità in un qualsiasi linguaggio, le risorse potenzialmente limitate devono essere liberate dopo l'uso. In questo caso parliamo del flusso out. Anche in scala c'è il try-finally. Si può scrivere così:

```
def save(id: String, bean: AnyRef) {
    val file = idToFile(id)
    var out: FileOutputStream = null
    try {
        out = new FileOutputStream(file)
        val encoder = new XMLEncoder(out)
        encoder.writeObject(bean)
        encoder.flush
    } finally {
        if(out != null) out.close
    }
}
```

Scala ha il “null” per mantenere la compatibilità con Java ma non è idiomatico. Al suo posto si usa Option che denota un valore che può essere o non essere definito. In questo caso avremmo:

```
def save(id: String, bean: AnyRef) {
    val file = idToFile(id)
    var out: Option[FileOutputStream] = None
    try {
        out = Some(new FileOutputStream(file))
        val encoder = new XMLEncoder(out.get)
        encoder.writeObject(bean)
        encoder.flush
    } finally {
        out.foreach(x => x.close)
```

```
        }
    }
```

Detto questo, si può fare di più e di meglio o peggio, dipende dai punti di vista. Ad esempio, com'è in voga, si potrebbe pensage che nel linguaggio esista una parola chiave, chiamiamola “using” che data una risorsa esegua un certo blocco di codice e a prescindere da ciò che accade si assicuri che la risorsa sia liberata. Del genere:

```
def save(id: String, bean: AnyRef) = using(new FileOutputStream(idToString(id)) { out =>
        val encoder = new XMLEncoder(out)
        encoder.writeObject(bean)
        encoder.flush
}
```

Si può fare, poi uno deve giudicare quanto sia conveniente. Si possono cioè creare strutture il cui uso le faccia apparire come estensioni del linguaggio anche se altro non sono che comuni metodi o classi o oggetti eccetera. Il nostro using sarebbe qui un metodo che accetta per argomenti due funzioni, una che inizializza un parametro e l'altra che lo usa. Il corpo del metodo dichiarerebbe un valore opzionale, in un blocco try lo inializerebbe con il valore restituito dall'inizializzatore e lo passerebbe alla funzione utente. A prescindere dal risultato si occuperebbe poi di liberare ciò che è stato inizializzato:

```
package mylanguageextension


object using {

    def apply[T <: java.io.Closeable](c: => T)(body: T => Unit) = {
        var opc: Option[T] = None
        try {
            opc = Some(c)
            body(opc.get)
        } finally {
            opc.foreach(c => c.close)
        }
    }
}
```

Ci sono naturalmente una tonnellata di norme in gioco, peraltro le stesse che regolano quel che si può assegnare ad un def o che forma può avere una funzione, con l'unica differenza che qui bisogna esserne consapevoli mentre di norma si può far affidamente sul fatto che ciò che sembra è anche quello che capiterà. Chiusa la seconda parentesi, per BeanStore restiamo sul più rassicurante javese col try finally.

```
package beanstore

import java.io.{FileOutputStream, File}
import java.beans.XMLEncoder

/**
 * Salva e carica dei java bean in una cartella predefinita
 */
object BeanStore {

    /**
     * La cartella predefinita in cui sono salvati i bean
     */
    var storeDirectory = {
        val dir = new File(System.getProperty("user.home"), "beanstore")
        if(!dir.exists) dir.mkdir
        dir
    }

    /**
     * L'estensione dei file che contengono i bean salvati
     */
    var extension = ".bean.xml";
```

```
	/**
	 * Trasforma un id in un nome di file
	 */
	def idToFile(id: String) = new File(storeDirectory, id + extension)

	/**
	 * Salva un bean associandolo ad un id
	 */
	def save(id: String, bean: AnyRef) {
		val file = idToFile(id)
		var out: Option[FileOutputStream] = None
		try {
			out = Some(new FileOutputStream(file))
			val encoder = new XMLEncoder(out.get)
			encoder.writeObject(bean)
			encoder.flush
			encoder.close
		} finally {
			out.foreach(x => x.close)
		}
	}
}
```

Per salvare abbiamo salvato, mo' carichiamo. Possiamo dire:

```
	/**
	 * Carica il bean associato all'id in argomento
	 */
	def load(id: String): Option[AnyRef] = {
		var result: Option[AnyRef] = None
		val file = idToFile(id)
		if(file.exists) {
			var in: Option[FileInputStream] = None
			try {
				in = Some(new FileInputStream(file))
				val decoder = new XMLDecoder(in)
				result = Some(decoder.readObject)
				decoder.close
			} finally {
				in.foreach(x => x.close)
			}
		}
		result
	}
```

Qui load restituisce un Option, qualcosa che può non esserci. Può non esserci perchè lato BeanStore che ad id corriponda un JavaBean salvato è cosa ignota. Se il file c'è, si suppone che ci sia anche il bean e tentiamo il caricamento. A meno di eccezioni, il bean dovrebbe saltar fuori.

Ultima operazione del BeanStore è una bella scansione di tutto ciò che contiene lo store. Classica ricerca, per intenderci. Diciamo che la ricerca è una funzione applicata ad ogni elemento contenuto nel bean store. La funzione restituisce un boolean che determina se la ricerca deva proseguire o no.

```
	/**
	 * Applica un'operazione ad ogni elemento del bean store. L'applicazione
	 * termina quando f restituisce false o
	 * quando è stato esaminato l'ultimo elemento del bean store.
	 */
	def foreach(f: AnyRef => Boolean) {

	}
```

AnyRef => Boolean è il tipo della funzione: piglia un AnyRef (all'incirca java.lang.Object) e restituisce un Boolean.

L'esecuzione è (costosamente) banale. Data la directory in cui si trovano i bean, prendiamo tutti i file che terminano con l'estensione da noi prevista, li carichiamo come bean, li diamo in pasto alla funzione e se la funzione restituisce true, passiamo al successivo. Caricare un bean a partire da un file è un po' di quello che fa il metodo load quindi gli spezziamo le zampine e lo separiamo in due:

```
/**
 * Carica il bean associato all'id in argomento
 */
def load(id: String): Option[AnyRef] = load(idToFile(id))

/**
 * Carica il bean contenuto nel file in argomento
 */
protected def load(file: File): Option[AnyRef] = {
	var result: Option[AnyRef] = None
	if(file.exists) {
		var in: Option[FileInputStream] = None
		try {
			in = Some(new FileInputStream(file))
			val decoder = new XMLDecoder(in.get)
			result = Some(decoder.readObject)
			decoder.close
		} finally {
			in.foreach(x => x.close)
		}
	}
	result
}
```

Così possiamo usare il load nel metodo foreach. Protetto perchè serve a noi ma non all'utente. Una versione di foreach potrebbe essere questa:

```
/**
 * Applica un'operazione ad ogni elemento del bean store. L'applicazione termina
 * quando f restituisce false o
 * quando è stato esaminato l'ultimo elemento del bean store.
 */
def foreach(f: AnyRef => Boolean) {
	val filter = new FileFilter() {
		def accept(pathName: File) = pathName.getName.toLowerCase.endsWith(extension)
	}
	val beanFiles = storeDirectory.listFiles(filter)
	var fileIndex = 0
	var parseNext = true
	while(fileIndex < beanFiles.size && parseNext) {
		val beanFile = beanFiles(fileIndex)
		val opBean = load(beanFile)
		if(opBean.isDefined) {
			parseNext = f(opBean.get)
		}
		fileIndex += 1
	}
}
```

Il filtro è il normale FileFilter di java.io. In pratica pigliamo dalla directory quei file il cui nome finisce in ".bean.xml", per ciascuno di quei file tentiamo il caricamento e se funziona passiamo la palla alla funzione f. La funzione ci dice true se vuole andare avanti, false altrimenti. Si può fare di peggio? Eccomeno.

```
/**
 * Applica un'operazione ad ogni elemento del bean store. L'applicazione termina quando f restituisce false o
 * quando è stato esaminato l'ultimo elemento del bean store.
 */
def foreach(f: AnyRef => Boolean) {
	val filter = new FileFilter() {
```

```
            def accept(pathName: File) = pathName.getName.toLowerCase.endsWith(extension)
        }
        storeDirectory.listFiles(filter).flatMap(file => load(file)).exists(bean => !f(bean))
    }
```

Naturalmente un conto è poterlo fare un conto è volerlo fare. Che fa quella riga.

listFiles crea una lista di File, tutti quelli che finiscono per .bean.xml e che per noi contengono dei javabean.

Con flatMap transformiamo quella lista di file in una lista di bean, applicando ad ogni file il nostro metodo che carica i bean contenuti nei file. Quindi dopo il flatmap siamo passati da una lista di file ad una lista dei bean contenuti in quei file. Poi per ognuno di quei bean dobbiamo dire: ok, adesso applica f e se f restituisce false, fermati. Exists applica ad ogni elemento di una collezione una funzione finchè quella funzione non restituisce true. A noi serve il contrario, quindi !f(bean).

E' certamente più corto ma non andrei oltre.

Casi d'uso di BeanStore.

A 'sto punto abbiamo un BeanStore, che è uno strato di persistenza per istanze di classi JavaBean. Praticamente qualsiasi cosa. Come si dichiara un bean in Scala. C'è un'annotazione ad hoc, BeanProperty. Ad esempio:

```
class Persona {

        @BeanProperty var id = “”
        @BeanProperty var nome = “”
        @BeanProperty var cognome = “”
}
```

è un bean con tre proprietà di tipo string. Volendo salvarne uno con BeanStore diremmo:

```
val persona = new Persona
persona.id = “idpersona”
persona.nome = “pippo”
persona.cognome = “rossi”
BeanSore.save(persona.id, persona)
```

Per caricarlo, noto l'id, scriveremmo:

```
val op = BeanStore.load(“idpersona”)
if(op.isDefined) {
        val bean = op.get
        val persona = bean.asInstanceOf[Persona]
}
```

Quel “asInstanceOf” è la conversione esplicita tra tipi che in Scala è un metodo. Un tristo esempio completo è:

```
package beanstore

import reflect.BeanProperty

object Test {

    def main(args: Array[String]) {
        store()
        load()
    }

    def load() {
```

```
        val op = BeanStore.load("idpersona")
        if(op.isDefined) {
            val bean = op.get
            val persona = bean.asInstanceOf[Persona]
            System.out.println(persona.id)
            System.out.println(persona.nome)
            System.out.println(persona.cognome)
        }
    }

    def store() {
        val persona = new Persona
        persona.id = "idpersona"
        persona.nome = "pippo"
        persona.cognome = "rossi"
        BeanStore.save(persona.id, persona)
    }
}

class Persona {

    @BeanProperty var id = ""
    @BeanProperty var nome = ""
    @BeanProperty var cognome = ""
}
```

Un generatore di identificatori univoci.

BeanStore salva usando una stringa come identificatore univoco. Creiamo un generatore di id univoci che usi BeanStore per persistere. Una cosa del genere:

```
UIDGenerator
        def generateId(): String
```

Che gestisca dietro le quinte la persistenza del valore usato per il numero seguente. Il classico menelavolemani. Prima ci serve il bean che tenga traccia dell'ultimo valore generato:

```
class UIDBean {

    @BeanProperty var id: Long = 0xFFFFFFFFFFFFFFFFL
}
```

Il generatore carica il bean, se esiste, o ne istanzia uno nuovo. Prende l'id, lo aumenta di uno, lo converte in una stringa, diciamo un esadecimale di lunghezza costante, salva il bean e restituisce la stringa. Supponendo di usare come id del bean UIDBean la stringa convenzionale UIDBean, sappiamo che possiamo caricarlo dicendo:

```
val bean = BeanStore.load(“uidbean”)
```

Qui bean è un Option[AnyRef] e avrà un valore (isDefined vale true) se il bean è già stato salvato, altrimenti sarà indefinito. Un Option ha un metodo, getOrElse, che, applicato, restituisce o il valore dell'option stesso, se è definito, o un'alternativa, se l'option non è definito (cioè punta a null). Quindi se dico:

```
val bean = BeanStore.load(“uidbean”).orElse(new UIDBean)
```

bean vale l'istanza di UIDBean salvata dal bean store o un nuovo UIDBean se è la prima volta che lo creiamo. Il tipo restituito è AnyRef, noi sappiamo che un UIDBean (be', lo speriamo almeno) quindi eseguiamo, con un certa sicumera, una conversione esplicita:

```
val bean = BeanStore.oad(“uidbean”).orElse(new UIDBean).asInstanceOf[UIDBean]
```

Caricato il bean incrementiamo il valore del campo id di 1

```
bean.id += 1
```

Preleviamo il valore incrementato

```
val id = bean.id
```

Salviamo il bean

```
BeanStore.save(“uidbean”, bean)
```

e restituiamo l'id come una stringa esadecimale di 32 caratteri:

```
String.format(“%032X”, id.asInstanceOf[AnyRef])
```

La conversione di id è necessaria perchè un numerico Scala non è un AnyRef (la versione scala di java.lang.Object) ma un Any e il metodo format di java.lang.String richiede una lista di Object-AnyRef. Fine del generatore di identificatori univoci.

```
package beanstore

import reflect.BeanProperty

object UIDGenerator {

    def generateId(): String = {
        val bean = BeanStore.load("uidbean").getOrElse(new UIDBean).asInstanceOf[UIDBean]
        bean.id += 1
        val id = bean.id
        BeanStore.save("uidbean", bean)
        String.format("%032X", id.asInstanceOf[AnyRef])
    }
}

class UIDBean {

    @BeanProperty var id: Long = 0xFFFFFFFFFFFFFFFFL
}
```

Ebbene sì, vi tocca.

E' chiaro che vi becchiate l'agenda ma, diciamocelo, già con la Persona avreste dovuto sentirvela soffiare sul collo. Non occorre molta fantasia per immaginare il programma. C'è un tipo di dato che rappresenta un contatto che ha un tot di campi ed è un bean. Esiste un'interfaccia che per ogni campo del bean mostra un analogo campo modificabile dall'utente. L'interfaccia ha tre pulsanti, nuovo, salva e trova. Il contatto potrebbe essere questo:

```
package agenda

import reflect.BeanProperty

class Account {

    @BeanProperty var id = ""
    @BeanProperty var name = ""
    @BeanProperty var email = ""
    @BeanProperty var phone = ""
    @BeanProperty var notes = ""
}
```

Dei cinque campi il primo, id, è nascosto all'interfaccia utente, gli altri sono associati ad altrettanti campi di testo. La base del form potrebbe essere questa:

```
package agenda

import java.awt.BorderLayout
import javax.swing.JPanel

class AccountEditor extends JPanel(new BorderLayout) {

}
```

Da notare che l'invocazione di super-costruttore alternativo in Scala si fonde nell'intestazione della classe. La piattaforma Scala ha una sua libreria per le interfacce grafiche utente nel pacchetto scala-swing.jar, è un wrapper per le api Swing, se riuscite a trovare una ragione per usarlo sappiate che è lì.
L'editor ha un account in corso di modifica, inizialmente vuoto, tutti i campi necessari al suo riempimento e tre pulsanti, nuovo, salva e cerca.

```
package agenda

import java.awt.BorderLayout
import javax.swing.{JTextArea, JTextField, JButton, JPanel}

/**
 * Editor di un account
 */
class AccountEditor extends JPanel(new BorderLayout()) {

	//Contatto attualmente gestito dall'editor
	private var editedAccount = new Account()

	//Campi di controllo dei valori del contatto
	private val name = new JTextField(20)
	private val email = new JTextField(20)
	private val phone = new JTextField(20)
	private val notes = new JTextArea(10, 10)

	//Pulsanti delle operazioni sul contatto corrente
	private val saveAccount = new JButton("Salva")
	private val newAccount = new JButton("Nuovo")
	private val findAccount = new JButton("Trova")

	//Barra dei pulsanti
	private val toolbar = new JToolBar()

}
```

Dove si mette il codice che in java andrebbe nel costruttore? Nel corpo della classe. Il corpo di una classe (o di un object) è anche lo spazio in cui risiedono le istruzioni di inizializzazione delle istanze della classe o dell'object di turno.
Dopo le dichiarazioni passiamo alla composizione dell'interfaccia. Da notare che poiché tutte le dichiarazioni contenute nel corpo della classe sono membri è possibile, nel corpo della classe o object, accedere ad un nome di membro (un val, un var, un def o un tipo annidato e altro ancora) in una riga che precede la dichiarazione di quel membro. Nel caso di campi, il valore a cui si accede sarà quello ante-inizializzazione. Il che potrebbe non essere molto utile.

```
package agenda

import javax.swing._
import java.awt.{GridLayout, FlowLayout, Component, BorderLayout}

/**
 * Editor di un account
```

```
 */
class AccountEditor extends JPanel(new BorderLayout()) {

	//Contatto attualmente gestito dall'editor
	private var editedAccount = new Account()

	//Campi di controllo dei valori del contatto
	private val name = new JTextField(20)
	private val email = new JTextField(20)
	private val phone = new JTextField(20)
	private val notes = new JTextArea(10, 10)

	//Pulsanti delle operazioni sul contatto corrente
	private val saveAccount = new JButton("Salva")
	private val newAccount = new JButton("Nuovo")
	private val findAccount = new JButton("Trova")

	//Barra dei pulsanti
	private val toolbar = new JToolBar()

	//Contenitore dei campi di controllo
	private val form: JComponent = {
		//crea un wrapper con bordo titolato
		def titled(title: String, component: Component) = {
			val panel = new JPanel(new GridLayout(1, 1))
			panel.setBorder(BorderFactory.createTitledBorder(title))
			panel.add(component)
			panel
		}
		val panel = Box.createVerticalBox
		panel.add(titled("Nome", name))
		panel.add(titled("E-Mail", email))
		panel.add(titled("Telefono", phone))
		panel.add(titled("Note", new JScrollPane(notes)))
		val fixer = new JPanel(new FlowLayout(FlowLayout.LEADING, 0, 0))
		fixer.add(panel)
		fixer   //ultima espressione nel blocco, determina il valore del blocco stesso
	}

	//Composizione dell'interfaccia utente
	toolbar.setFloatable(false)
	toolbar.add(newAccount)
	toolbar.add(saveAccount)
	toolbar.add(findAccount)

	add(toolbar, BorderLayout.NORTH)
	add(form, BorderLayout.CENTER)
}
```

Qui direi che l'unica cosa interessante (a metà) è il valore di “form”.

```
private val form = { … }
```

Significa che il valore di form è il valore di un blocco che è come dire: per computare il valore iniziale di form occorre far due conti preliminari, intrallazzare un po' e alla fine restituire il valore effettivo. Il blocco che determina il valore di form fa uso di una funzione, chiamiamola d'appoggio, title, che, dato un testo e un componente, restituisce un contenitore col testo come bordo titolato. Poi segue la dichiarazione di un contenitore in cui si infilano un po' di questi pannelli intermedi che circondano i campi veri e propri. Infine si impacchetta il tutto in un contenitore che piazza il contenuto in alto a sinistra. Quel contenitore, fixer, è anche il valore di form. In questo caso, per farla breve, l'inizializzazione di un valore con un blocco altro non è che un modo per dire “qui ci sono un tot di operazioni di scarsissimo interesse il cui risultato finale è il valore che assegno a qualcuno”. I blocchi o, meglio, la notazione {}, ha anche scopi più proficui, in particolare nell'ambito della definizione di funzioni anonime multi linea.

```
val incrementa = x: Int => x  + 1
```

è una funzione così come lo è:

```
val incrementa = { x: int => x + 1 }
```

La differenza è che la seconda notazione permette di usare più linee per definire cosa faccia la funzione:

```
val incrementa = { x: Int =>
	val y = x + 1
	val z = y + 2
	z + 3//il risultato di questa espressione è anche il risultato della funzione incrementa
}
```

Chiusa l'ennesima parentesi. Data la relativa semplicità del programma, non creiamo una classe ad hoc per la finestra che conterrà l'editor dei contatti ma dotiamo AccountEditor di un metodo ad hoc per generare e aprire sullo schermo la sua finestrella, un po' come fosse una finestra di dialogo. Aggiungiamo quindi, nel corpo della classe AccountEditor, questo “def”:

```
	//apre una finestra che contiene questo account editor
	def popupWindow() {
		val frame = new JFrame("Agenda")
		frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE)
		frame.add(this)
		frame.setSize(640, 480)
		frame.setVisible(true)
	}
```

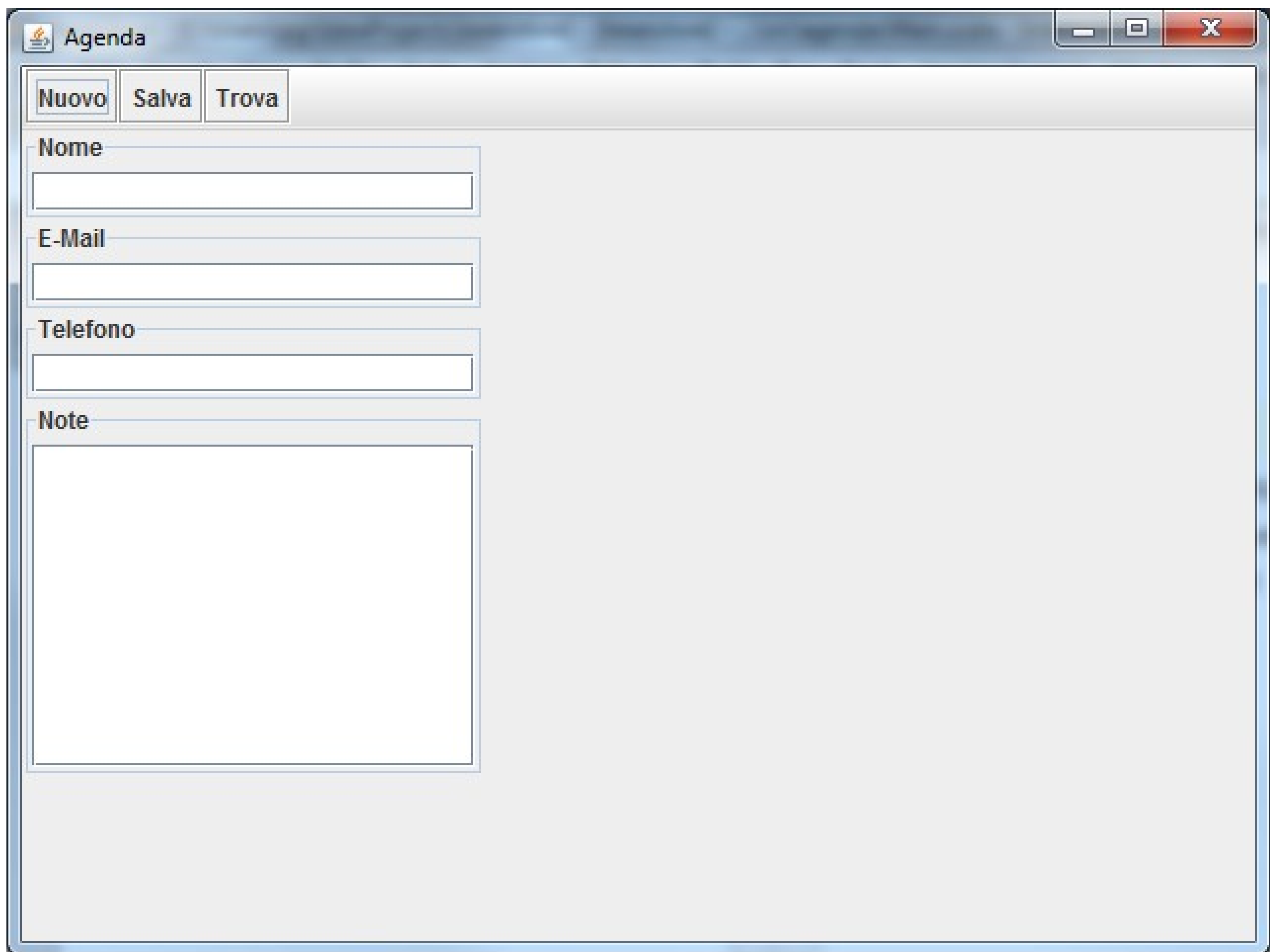
Prima di collegare i pulsanti a delle azioni, vediamo un po' cosa salta fuori. Creiamo una “classe Main”, salvo il fatto che in scala non è una classe ma un object:

```
package agenda

object Main {

	def main(args: Array[String]) {
		java.awt.EventQueue.invokeLater(new Runnable {
			def run = new AccountEditor().popupWindow
		})
	}
}
```

Lanciandola (da IntelliJ click col destro su Main nella scheda Project e premere run Main.main), compare la nostra delizia:



Non resta che definire la parte dinamica del programma. Il pulsante newAccount elimina tutti i dati contenuti nei campi di controllo. Per mantere in sincronia l'account in corso di modifica con il valore dei campi di controllo, creiamo un metodo setEditedAccount che assegni ai campi i valori del bean e definiamo un metodo clear come l'impostazione di un account vuoto. Sempre nella classe AccountEditor, scriviamo:

```
//Imposta l'account gestito dall'editor
def setEditedAccount(a: Account) {
	editedAccount = a
	name.setText(a.name)
	email.setText(a.email)
	phone.setText(a.phone)
	notes.setText(a.notes)
}

//elimina i dati contenuti nei campi di controllo
def clear() {
	setEditedAccount(new Account)
}
```

Ora colleghiamo la pressione del pulsante newAccount all'invocazione del metodo clear. Possiamo scrivere, nel corpo della classe AccountEditor:

```
//azioni
newAccount.addActionListener(new ActionListener {
	def actionPerformed(e: ActionEvent) = clear()
})
```

Consegue alla graziosità della sintassi di Scala che questa non sia l'unica opzione e certo non la più elegante. Tra le possibilità vi è quella di far sì che il nome di un metodo diventi un alias per un ActionListener. Si tratta di

decidere quanto sia conveniente farlo, sia dal punto di vista della mitologica leggibilità (che non può che dipendere da chi legge), sia dal punto di vista della sintesi (che è una riduzione quantitativa a parità di significato e non la prima stupidaggine più corta che ci venga in mente).
Posso dire:

```
newAccount.addActionListener(clear)
```

Se istruisco il compilatore a riconoscere che il nome del membro clear, un def, è equivalente ad un ActionListener pur non essendo il tipo del primo discendente del secondo. L'istruzione si realizza con l'introduzione di una conversione implicita. Quel che voglio è che un metodo senza argomenti il cui tipo restituito è irrilevante sia accettato come un ActionListener. La dichiarazione di convertibilità, che introduco sempre nel corpo della classe AccountEditor, ha la seguente forma:

```
private implicit def conversione(block: => Unit): ActionListener = new ActionListener {
    def actionPerformed(e: ActionEvent) = block
}
```

Uso il nome “conversione” per esser chiaro ma un nome più significativo – ad esempio “blockToActionListener”. Taccio sul resto (più che altro per evitare di dire stupidaggini) ma non è l'unica forma in cui che questa conversione “da nome di metodo a qualcos'altro” può essere data.
Insomma, premendo newAccount capita “clear”.
Per il salvataggio dei dati, creiamo prima un metodo che trasforma il contenuto dei campi in un contatto.

```
/**
 * Restituisce il contatto gestito dall'editor. Se richiesto genera
 * un id dove manchi.
 */
def getEditedAccount(generateMissingId: Boolean) = {
    if(generateMissingId && editedAccount.id.isEmpty) {
        editedAccount.id = UIDGenerator.generateId
    }
    editedAccount.name = name.getText
    editedAccount.email = email.getText
    editedAccount.phone = phone.getText
    editedAccount.notes = notes.getText
    editedAccount
}
```

Essendo opportuno salvare un contatto solo quando ci siano dei dati nel form, creiamo anche un metodo che stabilisca se l'editor abbia o no dei dati:

```
//True se i campi di controllo non contengono alcun dato
def isEmpty() = {
    name.getText.isEmpty &&
    email.getText.isEmpty &&
    phone.getText.isEmpty &&
    notes.getText.isEmpty
}
```

Il salvataggio consiste allora nel verificare se ci siano dei dati da salvare e, in caso affermativo, nella scrittura degli stessi attraverso BeanStore.

```
//Salva il contatto gestito dall'editor
def saveEditedAccount() {
    if(isEmpty) {
        JOptionPane.showMessageDialog(this, "Nessun dato da salvare.")
    } else {
        val account = getEditedAccount(true)
        BeanStore.save(account.id, account)
        JOptionPane.showMessageDialog(this, "Contatto salvato.")
    }
```

```
}
```

Quanto al collegamento tra il pulsante saveAccount e il metodo saveEditedAccount, in forza della conversione implicita, basta aggiungere:

```
saveAccount.addActionListener(saveEditedAccount)
```

Rimane la ricerca di un contatto. L'utente immette parte dei dati nel form. Per ogni account nel bean store, verifichiamo se i suoi dati contengano in tutto o in parte le stringhe immesse nel form. Aggiungiamo gli account che passano il controllo ad una lista e presentiamo questa lista all'utente. Iniziamo col dire "quando una stringa di ricerca è più o meno uguale ad un'altra":

```
def stringMatches(c: String, v: String) = {
	c.isEmpty || v.toLowerCase.contains(c.toLowerCase)
}
```

Proseguiamo col dire quanto un account di ricerca è più o meno uguale ad un altro:

```
def matches(c: Account, a: Account) = {
	stringMatches(c.name, a.name) &&
	stringMatches(c.email, a.name) &&
	stringMatches(c.phone, a.phone) &&
	stringMatches(c.notes, a.notes)
}
```

E chiudiamo applicando questa similitudine tra un account di ricerca ed uno concreto.

```
//cerca un account usando i valori nel form come criteri
def searchAccount() {
	val searchCriteria = getEditedAccount(false)
	val result = ListBuffer[Account]()
	BeanStore.foreach{ bean =>
		if(bean.isInstanceOf[Account]) {
			val account = bean.asInstanceOf[Account]
			if(matches(searchCriteria, account)) result.append(account)
		}
		true
	}
}
```

Rendendo la cosa più esplicità, là c'è scritto:

```
//cerca un account usando i valori nel form come criteri
def searchAccount() {
	val searchCriteria = getEditedAccount(false)
	val result = ListBuffer[Account]()
	val funzione: Function1[AnyRef, Boolean] = (bean: AnyRef) => {
		if(bean.isInstanceOf[Account]) {
			val account = bean.asInstanceOf[Account]
			if(matches(searchCriteria, account)) result.append(account)
		}
		true
	}
	BeanStore.foreach(funzione)
}
```

Il motore inferenziale del compilatore scala permette poi di eliminare un tot di specificazioni di tipo e si arriva alla prima formulazione.
Dopo il foreach abbiamo tre casi: la lista è vuota, la lista contiene un solo elemento, la lista contiene più elementi.
Se la lista è vuota, mostriamo un messaggio di commiato.

```
//cerca un account usando i valori nel form come criteri
def searchAccount() {
	val searchCriteria = getEditedAccount(false)
	val result = ListBuffer[Account]()
	BeanStore.foreach { bean =>
		if(bean.isInstanceOf[Account]) {
			val account = bean.asInstanceOf[Account]
			if(matches(searchCriteria, account)) result.append(account)
		}
		true
	}
	if(result.isEmpty) {
		JOptionPane.showMessageDialog(this, "Nessun risultato trovato.")
	}
}
```

Se la lista contiene un solo elemento, lo passiamo all'editor:

```
//cerca un account usando i valori nel form come criteri
def searchAccount() {
	val searchCriteria = getEditedAccount(false)
	val result = ListBuffer[Account]()
	BeanStore.foreach { bean =>
		if(bean.isInstanceOf[Account]) {
			val account = bean.asInstanceOf[Account]
			if(matches(searchCriteria, account)) result.append(account)
		}
		true
	}
	if(result.isEmpty) {
		JOptionPane.showMessageDialog(this, "Nessun risultato trovato.")
	} else if(result.size == 1) {
		setEditedAccount(result(0))
	}
}
```

Se la lista contiene più risultati la trama si complica e passiamo la palla ad un metodo ad hoc:

```
//cerca un account usando i valori nel form come criteri
def searchAccount() {
	val searchCriteria = getEditedAccount(false)
	val result = ListBuffer[Account]()
	BeanStore.foreach { bean =>
		if(bean.isInstanceOf[Account]) {
			val account = bean.asInstanceOf[Account]
			if(matches(searchCriteria, account)) result.append(account)
		}
		true
	}
	if(result.isEmpty) {
		JOptionPane.showMessageDialog(this, "Nessun risultato trovato.")
	} else if(result.size == 1) {
		setEditedAccount(result(0))
	} else {
		chooseSearchResult(result)
	}
}

//Permette all'utente di scegliere un account tra i risultati di una ricerca
def chooseSearchResult(result: ListBuffer[Account]) {

}
```

Per la scelta tra più risultati usiamo una finestrella con una lista interattiva. Quando l'utente scegli un elemento con un click del mouse, chiudiamo la finestra di dialogo e impostiamo il contatto nell'editor.

```
	//Permette all'utente di scegliere un account tra i risultati di una ricerca
	def chooseSearchResult(result: ListBuffer[Account]) {
		val model = new DefaultListModel
		val list = new JList(model)
		val scroller = new JScrollPane(list)
		val frame = JOptionPane.getFrameForComponent(this)
		val dialog = new JDialog(frame, true)
		dialog.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE)
		list.addMouseListener(new MouseAdapter {
			override def mouseClicked(e: MouseEvent) = if(SwingUtilities.isLeftMouseButton(e)) {
				val index = list.locationToIndex(e.getPoint)
				if(index >= 0) {
					setEditedAccount(result(index))
					dialog.dispose()
				}
			}
		})
		result.foreach(account => model.addElement(account))
		dialog.setTitle("Risultato ricerca")
		dialog.add(scroller)
		dialog.setSize(new Dimension(300, 200))
		val x = frame.getLocation.x + (frame.getWidth - dialog.getWidth) / 2
		val y = frame.getLocation.y + (frame.getHeight - dialog.getHeight) / 2
		dialog.setLocation(x, y)
		dialog.setVisible(true)
	}
```

La lista usa (anche) il metodo toString di Account per mostrare gli account all'utente. Il metodo toString predefinito è poco significativo, lo ridefiniamo:

```
package agenda

import reflect.BeanProperty

class Account {

	@BeanProperty var id = ""
	@BeanProperty var name = ""
	@BeanProperty var email = ""
	@BeanProperty var phone = ""
	@BeanProperty var notes = ""

	override def toString() = List(name, email, phone, notes).find(e => !e.isEmpty).getOrElse("Senza Nome")
}
```

Vale a dire il primo non vuoto tra nome, email, telefono e note oppure un rassegnato “Senza Nome”.

Colleghiamo il pulsante findAccount al metodo searchAccount:

```
	findAccount.addActionListener(searchAccount)
```

E l'agenda è bell'e che finita.

Il codice del programma.

Tutto quello che serve è già stato scritto, ripeto il codice per comodità dei masochisti.

BeanStore.scala

```
package beanstore

import java.beans.{XMLDecoder, XMLEncoder}
import java.io.{FileFilter, FileInputStream, FileOutputStream, File}

/**
 * Salva e carica dei java bean in una cartella predefinita
 */
object BeanStore {

	/**
	 * La cartella predefinita in cui sono salvati i bean
	 */
	var storeDirectory = {
		val dir = new File(System.getProperty("user.home"), "beanstore")
		if(!dir.exists) dir.mkdir
		dir
	}

	/**
	 * L'estensione dei file che contengono i bean salvati
	 */
	var extension = ".bean.xml";

	/**
	 * Trasforma un id in un nome di file
	 */
	def idToFile(id: String) = new File(storeDirectory, id + extension)

	/**
	 * Salva un bean associandolo ad un id
	 */
	def save(id: String, bean: AnyRef) {
		val file = idToFile(id)
		var out: Option[FileOutputStream] = None
		try {
			out = Some(new FileOutputStream(file))
			val encoder = new XMLEncoder(out.get)
			encoder.writeObject(bean)
			encoder.flush
			encoder.close
		} finally {
			out.foreach(x => x.close)
		}
	}

	/**
	 * Carica il bean associato all'id in argomento
	 */
	def load(id: String): Option[AnyRef] = load(idToFile(id))

	/**
	 * Carica il bean contenuto nel file in argomento
	 */
	protected def load(file: File): Option[AnyRef] = {
		var result: Option[AnyRef] = None
		if(file.exists) {
			var in: Option[FileInputStream] = None
			try {
				in = Some(new FileInputStream(file))
				val decoder = new XMLDecoder(in.get)
				result = Some(decoder.readObject)
				decoder.close
			} finally {
				in.foreach(x => x.close)
```

```
            }
        }
        result
    }

    /**
     * Applica un'operazione ad ogni elemento del bean store. L'applicazione termina quando f restituisce false o
     * quando è stato esaminato l'ultimo elemento del bean store.
     */
    def foreach(f: AnyRef => Boolean) {
        val filter = new FileFilter() {
            def accept(pathName: File) = pathName.getName.toLowerCase.endsWith(extension)
        }
        val beanFiles = storeDirectory.listFiles(filter)
        var fileIndex = 0
        var parseNext = true
        while(fileIndex < beanFiles.size && parseNext) {
            val beanFile = beanFiles(fileIndex)
            val opBean = load(beanFile)
            if(opBean.isDefined) {
                parseNext = f(opBean.get)
            }
            fileIndex += 1
        }
    }
}
```

UIDGenerator.scala

```
package beanstore

import reflect.BeanProperty

object UIDGenerator {

	def generateId(): String = {
		val bean = BeanStore.load("uidbean").getOrElse(new UIDBean).asInstanceOf[UIDBean]
		bean.id += 1
		val id = bean.id
		BeanStore.save("uidbean", bean)
		String.format("%032X", id.asInstanceOf[AnyRef])
	}
}

class UIDBean {

	@BeanProperty var id: Long = 0xFFFFFFFFFFFFFFFFL
}
```

Account.scala

```
package agenda

import reflect.BeanProperty

class Account {

	@BeanProperty var id = ""
	@BeanProperty var name = ""
	@BeanProperty var email = ""
	@BeanProperty var phone = ""
	@BeanProperty var notes = ""

	override def toString() = List(name, email, phone, notes).find(e => !e.isEmpty).getOrElse("Senza Nome")
}
```

AccountEditor.scala

```
package agenda

import javax.swing._
import beanstore.{BeanStore, UIDGenerator}
import collection.mutable.ListBuffer
import java.awt.event.{MouseAdapter, ActionListener, ActionEvent, MouseEvent}
import java.awt._

/**
 * Editor di un account
 */
class AccountEditor extends JPanel(new BorderLayout()) {

	//Contatto attualmente gestito dall'editor
	private var editedAccount = new Account()

	//Campi di controllo dei valori del contatto
	private val name = new JTextField(20)
	private val email = new JTextField(20)
	private val phone = new JTextField(20)
	private val notes = new JTextArea(10, 10)

	//Pulsanti delle operazioni sul contatto corrente
	private val saveAccount = new JButton("Salva")
	private val newAccount = new JButton("Nuovo")
	private val findAccount = new JButton("Trova")

	//Barra dei pulsanti
	private val toolbar = new JToolBar()

	//Contenitore dei campi di controllo
	private val form: JComponent = {
		//crea un wrapper con bordo titolato
		def titled(title: String, component: Component) = {
			val panel = new JPanel(new GridLayout(1, 1))
			panel.setBorder(BorderFactory.createTitledBorder(title))
			panel.add(component)
			panel
		}
		val panel = Box.createVerticalBox
		panel.add(titled("Nome", name))
		panel.add(titled("E-Mail", email))
		panel.add(titled("Telefono", phone))
		panel.add(titled("Note", new JScrollPane(notes)))
		val fixer = new JPanel(new FlowLayout(FlowLayout.LEADING, 0, 0))
		fixer.add(panel)
		fixer   //ultima espressione nel blocco, determina il valore del blocco stesso
	}

	//Composizione dell'interfaccia utente
	toolbar.setFloatable(false)
	toolbar.add(newAccount)
	toolbar.add(saveAccount)
	toolbar.add(findAccount)

	add(toolbar, BorderLayout.NORTH)
	add(form, BorderLayout.CENTER)

	//azioni
	newAccount.addActionListener(clear)
	saveAccount.addActionListener(saveEditedAccount)
	findAccount.addActionListener(searchAccount)

	//converte un blocco in un action listener
	private implicit def conversione(block: => Unit): ActionListener = new ActionListener {
```

```
        def actionPerformed(e: ActionEvent) = block
}

//apre una finestra che contiene questo account editor
def popupWindow() {
        val frame = new JFrame("Agenda")
        frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE)
        frame.add(this)
        frame.setSize(640, 480)
        frame.setVisible(true)
}

//Imposta l'account gestito dall'editor
def setEditedAccount(a: Account) {
        editedAccount = a
        name.setText(a.name)
        email.setText(a.email)
        phone.setText(a.phone)
        notes.setText(a.notes)
}

//elimina i dati contenuti nei campi di controllo
def clear() {
        setEditedAccount(new Account)
}

/**
 * Restituisce il contatto gestito dall'editor. Se richiesto genera
 * un id dove manchi.
 */
def getEditedAccount(generateMissingId: Boolean) = {
        if(generateMissingId && editedAccount.id.isEmpty) {
                editedAccount.id = UIDGenerator.generateId
        }
        editedAccount.name = name.getText
        editedAccount.email = email.getText
        editedAccount.phone = phone.getText
        editedAccount.notes = notes.getText
        editedAccount
}

//True se i campi di controllo non contengono alcun dato
def isEmpty() = {
        name.getText.isEmpty &&
        email.getText.isEmpty &&
        phone.getText.isEmpty &&
        notes.getText.isEmpty
}

//Salva il contatto gestito dall'editor
def saveEditedAccount() {
        if(isEmpty) {
                JOptionPane.showMessageDialog(this, "Nessun dato da salvare.")
        } else {
                val account = getEditedAccount(true)
                BeanStore.save(account.id, account)
                JOptionPane.showMessageDialog(this, "Contatto salvato.")
        }
}

def stringMatches(c: String, v: String) = {
        c.isEmpty || v.toLowerCase.contains(c.toLowerCase)
}

def matches(c: Account, a: Account) = {
        stringMatches(c.name, a.name) &&
        stringMatches(c.email, a.name) &&
        stringMatches(c.phone, a.phone) &&
        stringMatches(c.notes, a.notes)
}
```

```
    //cerca un account usando i valori nel form come criteri
    def searchAccount() {
        val searchCriteria = getEditedAccount(false)
        val result = ListBuffer[Account]()
        BeanStore.foreach { bean =>
            if(bean.isInstanceOf[Account]) {
                val account = bean.asInstanceOf[Account]
                if(matches(searchCriteria, account)) result.append(account)
            }
            true
        }
        if(result.isEmpty) {
            JOptionPane.showMessageDialog(this, "Nessun risultato trovato.")
        } else if(result.size == 1) {
            setEditedAccount(result(0))
        } else {
            chooseSearchResult(result)
        }
    }

    //Permette all'utente di scegliere un account tra i risultati di una ricerca
    def chooseSearchResult(result: ListBuffer[Account]) {
        val model = new DefaultListModel
        val list = new JList(model)
        val scroller = new JScrollPane(list)
        val frame = JOptionPane.getFrameForComponent(this)
        val dialog = new JDialog(frame, true)
        dialog.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE)
        list.addMouseListener(new MouseAdapter {
            override def mouseClicked(e: MouseEvent) = if(SwingUtilities.isLeftMouseButton(e)) {
                val index = list.locationToIndex(e.getPoint)
                if(index >= 0) {
                    setEditedAccount(result(index))
                    dialog.dispose()
                }
            }
        })
        result.foreach(account => model.addElement(account))
        dialog.setTitle("Risultato ricerca")
        dialog.add(scroller)
        dialog.setSize(new Dimension(300, 200))
        val x = frame.getLocation.x + (frame.getWidth - dialog.getWidth) / 2
        val y = frame.getLocation.y + (frame.getHeight - dialog.getHeight) / 2
        dialog.setLocation(x, y)
        dialog.setVisible(true)
    }
}
```

Main.scala

```
package agenda

object Main {

	def main(args: Array[String]) {
		java.awt.EventQueue.invokeLater(new Runnable {
			def run = new AccountEditor().popupWindow
		})
	}
}
```