Luca Parimbelli
Matricola: 1053142
https://github.com/lucaparimbelli/libreriaDigitale-Scala

Documentazione progetto Scala
Libreria digitale

Introduzione
Il progetto si pone l'obiettivo di utilizzare i principali costrutti di Scala presentati durante il corso di "Informatica III – modulo programmazione" per implementare una libreria digitale. La libreria digitale consente di svolgere le principali attività tipiche di una libreria: visualizzazione degli articoli presenti nel sistema, ricerca di un articolo, inserimento e cancellazione di nuovi articoli, prestito e restituzione di articoli disponibili.

Struttura
Il progetto è diviso in 4 packages, ognuno dei quali contiene diverse classi che consentono di gestire aspetti differenti del programma. In particolare:

- *libreria*, contiene le classi che servono per modellizzare la struttura della libreria
- *manager*, contiene le classi che consentono di implementare la logica del programma
- *customException*, contiene eccezioni personalizzate utili per poter gestire situazioni specifiche
- *menuCLI*, contiene la classe che consente di gestire l'interfaccia grafica a linea di comando

Le classi contenute nei packages *libreria*, *manager* e *customException* vengono utilizzate dalle classi di *menuCLI* per realizzare l'interfaccia grafica a linea di comando del progetto (*Fig. 1*).

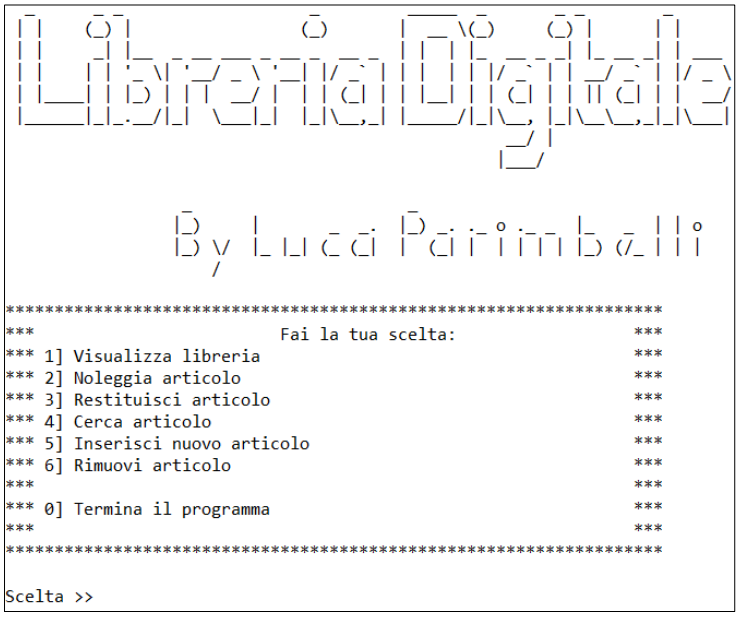

Fig. 1: interfaccia grafica CLI del progetto Scala

Classi e relazioni
Le classi e le loro relazioni all'interno di ciascun package sono descritte dai successivi *class diagrams*. Come si potrà osservare nei successivi diagrammi, il progetto realizzato in Scala è una versione ridotta del progetto realizzato in C++.

Il package *libreria* è descritto dal seguente class diagram (*Fig. 2*):

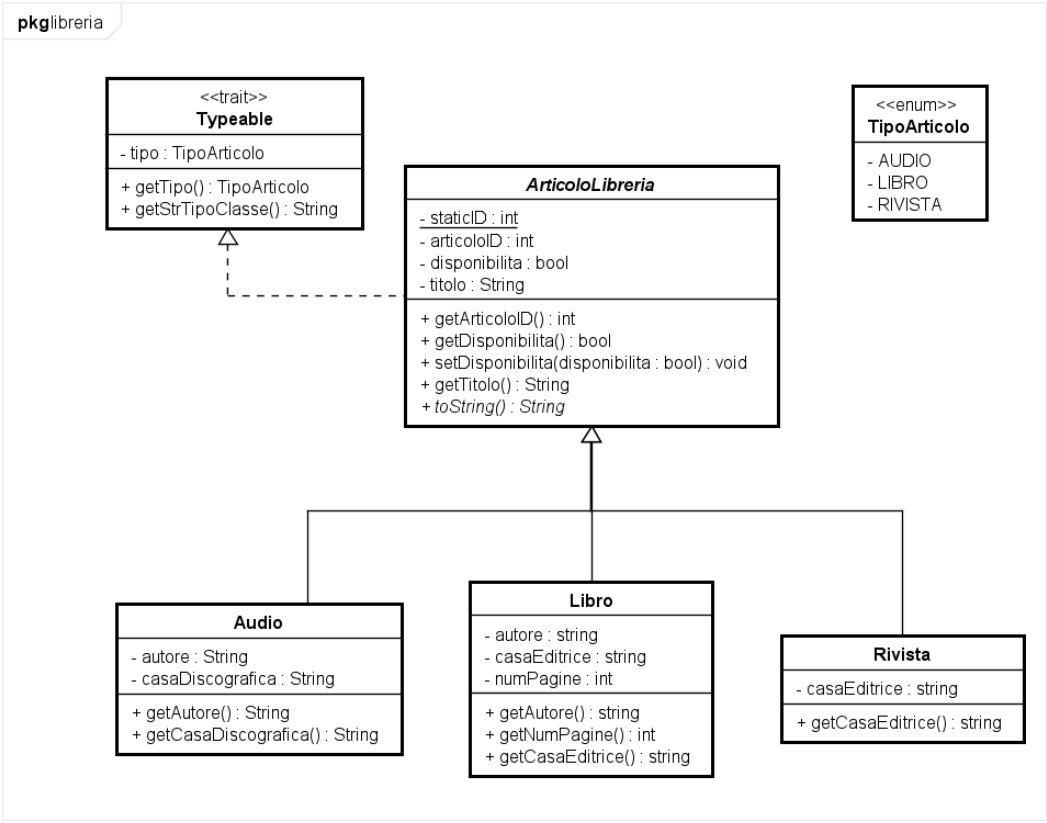

Fig. 2: libreria class diagram

Il package *manager* è descritto dal seguente class diagram (*Fig. 3*):

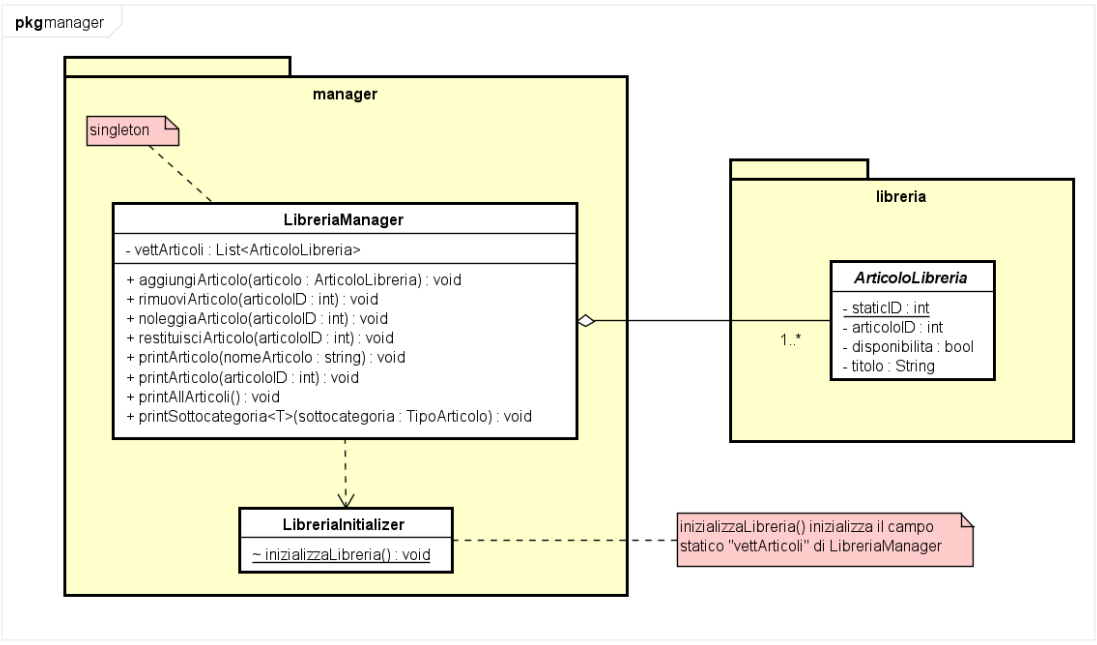

Fig. 3: manager class diagram

Il package *customException* è descritto dal seguente class diagram (*Fig. 4*):

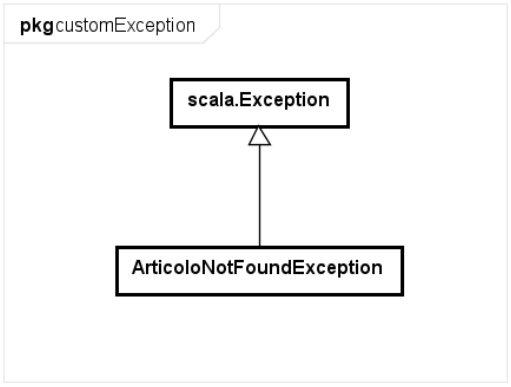

Fig. 4: customException class diagram

Il package *menuCLI* è descritto dal seguente class diagram (*Fig. 5*):

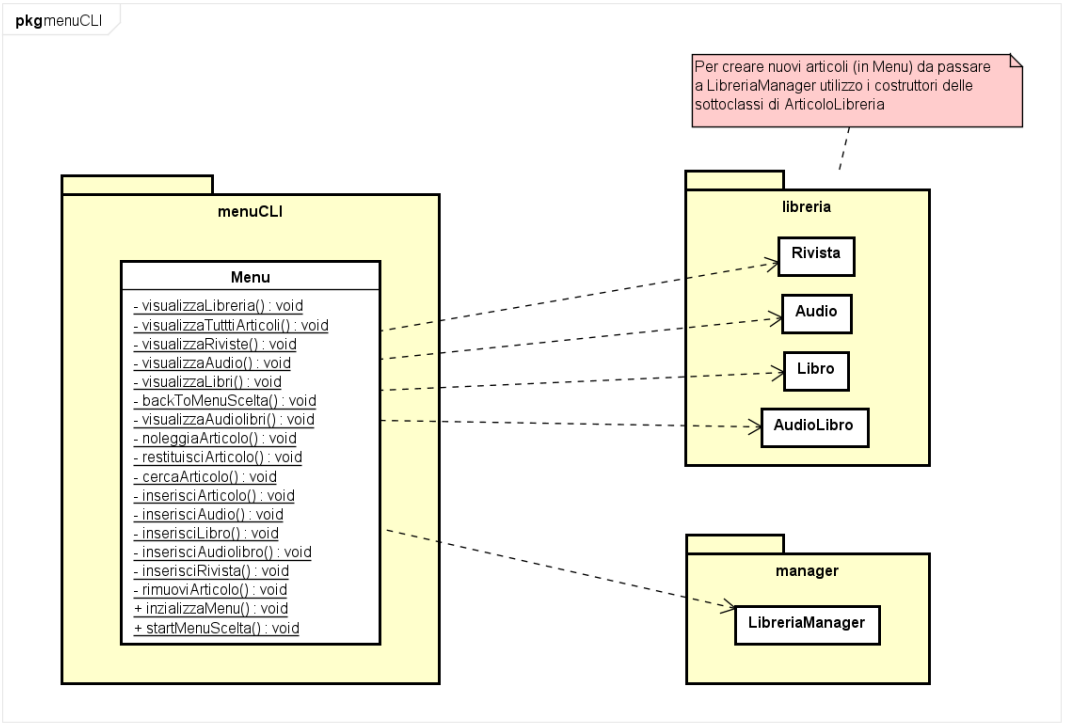

Fig. 5: menuCLI class diagram

Costrutti di Scala

Nei successivi sottoparagrafi vengono presentati i costrutti tipici di Scala che sono stati utilizzati per la realizzazione del progetto.

Classi

La struttura dei packages e le classi impiegate nel progetto sono illustrate in *Fig. 6*.

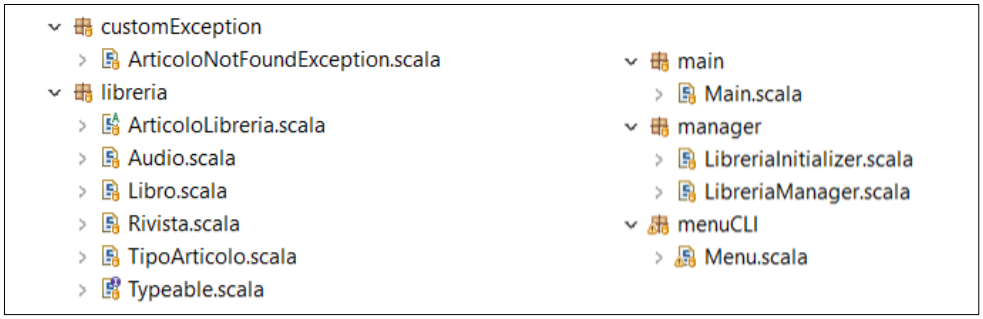

Fig. 6: classi e packages del progetto

Trait

Con l'obiettivo di esemplificare l'utilizzo dei traits di Scala, è stato introdotto il trait *Typeable*. I trait sono simili alle interfacce Java, ma a differenza loro possono avere una implementazione parziale dei metodi. Per questo motivo, il trait *Typeable* è stato progettato con tre membri:

- il campo *tipo*, che costringe le classi che implementano *Typeable* a definire un campo di tipo *TipoArticolo* che rappresenta la tipologia di articolo preso in considerazione (AUDIO, LIBRO, RIVISTA)
- il metodo *getTipo*, che è implementato direttamente nel trait e restituisce il valore del campo *tipo*
- il *metodo getStrTipoClasse*, che non è stato implementato nel trait e impone quindi alla classe che implementerà *Typeable* di definire un metodo con uguale segnatura e che restituisce una stringa contenente informazioni circa il tipo dell'oggetto che invoca il metodo.

Il metodo *getStrTipoClasse* non è stato volutamente implementato nel trait in quanto si è voluto sottolineare la già sopracitata differenza principale che sussiste tra interfacce Java e traits Scala: le prime non consentono di definire un'implementazione dei metodi nel loro corpo, i secondi invece sì.

```
trait Typeable {

  import TipoArticolo._

  val tipo : TipoArticolo

  //metodo già implementato
  def getTipo = tipo

  //metodo da implementare nelle classi che implementano il trait
  def getStrTipoClasse : String

}
```

Fig. 7: trait Typeable

Classe astratta

La classe *ArticoloLibreria*, come già presentato nel progetto realizzato in C++, è stata concepita come classe astratta. Come si può osservare in *Fig. 8*, la classe *ArticoloLibreria* viene resa astratta mediante l'utilizzo della parola chiave *abstract*. Questa classe implementa il trait *Typeable* e presenta nel suo corpo un metodo astratto *toString()*: questo significa che tutte le classi che erediteranno da *ArticoloLibreria* dovranno implementare i metodi *getStrTipoClasse()* e *toString()* e fornire un valore al campo *tipo* di *Typeable*.

```
abstract class ArticoloLibreria(val titolo: String) extends Typeable {

  //metodo astratto da implementare nelle sottoclassi
  def toString: String
```

Fig. 8: classe astratta ArticoloLibreria

Ereditarietà

Contrariamente a quanto avviene in C++, in Scala non c'è ereditarietà multipla, bensì solo ereditarietà singola. Nel progetto, le tre classi *Audio*, *Libro* e *Rivista* ereditano da *ArticoloLibreria* mediante l'utilizzo della parola chiave *extends*. L'ereditarietà da parte di queste tre classi e la chiamata al costruttore della superclasse *ArticoloLibreria* sono raffigurati in *Fig. 9*.

```
abstract class ArticoloLibreria(val titolo: String) extends Typeable {

----------

class Libro(titolo: String, val autore: String, val casaEditrice: String, val numPagine: Int)
extends ArticoloLibreria(titolo) {

----------

class Rivista(titolo: String, val casaEditrice: String) extends ArticoloLibreria(titolo) {

----------

class Audio(titolo: String, val autore: String, val casaDiscografica: String) extends
ArticoloLibreria(titolo) {
```

Fig. 9: ereditarietà singola da ArticoloLibreria

Enum

Come già discusso nella documentazione del progetto C++, il formato specifico ideato per la stringa che contiene le informazioni circa un articolo e che viene restituita dal metodo *toString()* prevede di specificare la tipologia dell'articolo nella prima riga della stringa. Per questo motivo è stato progettato un enum *TipoArticolo* che può assumere solo 3 valori: AUDIO, LIBRO e RIVISTA. Il campo *tipo* di *ArticoloLibreria* è un campo che deve essere obbligatoriamente implementato a causa del fatto che la classe *ArticoloLibreria* implementa *Typeable*. L'enum *TipoArticolo* e l'implementazione del campo *tipo* da parte delle sottoclassi *Audio*, *Libro* e *Rivista* sono illustrati in F*ig. 10*.

```
object TipoArticolo extends Enumeration {
  type TipoArticolo = Value
  val AUDIO, LIBRO, RIVISTA = Value
}

----------

class Libro(titolo: String, val autore: String, val casaEditrice: String, val numPagine: Int)
extends ArticoloLibreria(titolo) {
  //implementazione campo e metodo trait
  import TipoArticolo._
  val tipo: TipoArticolo = TipoArticolo.LIBRO

----------

class Rivista(titolo: String, val casaEditrice: String) extends ArticoloLibreria(titolo) {
  //implementazione campo e metodo trait
  import TipoArticolo._
  val tipo: TipoArticolo = TipoArticolo.RIVISTA

----------

class Audio(titolo: String, val autore: String, val casaDiscografica: String) extends
ArticoloLibreria(titolo) {
  //implementazione campo e metodo trait
  import TipoArticolo._
  val tipo: TipoArticolo = TipoArticolo.AUDIO
```

Fig. 10: enum TipoArticolo e inizializzazione campo tipo

Companion object e membri statici

A differenza di Java, in Scala non esiste la parola chiave *static*. Per poter definire l'equivalente dei membri statici di Java in Scala è necessario utilizzare il cosiddetto companion object. Un companion object è un oggetto (*object*):

1. con lo stesso nome della classe
2. che si trova nello stesso file in cui è dichiarata la classe

Il companion object e la classe possono accedere ognuno a tutti membri dell'altro (di ogni tipo di visibilità). Tutto ciò che è dichiarato all'interno del companion object (visibilità permettendo) può essere richiamato

all'occorrenza utilizzando direttamente il nome della classe, senza necessità di creare un'istanza di quest'ultima.
Nel progetto sono stati definiti diversi campi e metodi statici, in particolare:

- il campo statico *staticID* è definito nel companion object di *ArticoloLibreria* e viene utilizzato per incrementare automaticamente il valore *articoloID* degli articoli che vengono inseriti nella libreria digitale
- il metodo statico *inizializzaLibreria* è dichiarato nell'object *LibreriaInitializer* ed è utilizzato per inizializzare la libreria digitale con degli articoli di default nella fase di avvio del programma
- le classi *LibreriaManager* e *Menu* hanno tutti i loro membri definiti all'interno dell'object (sono singleton) e possiamo quindi considerare tutti i loro membri come se fossero statici

I membri dell'elenco soprastante sono presentati più nel dettaglio in *Fig. 11*.

```
abstract class ArticoloLibreria(val titolo: String) extends Typeable {
   ...
}

//companion object per membro statico
object ArticoloLibreria {
  var staticID: Int = 0
}
```

```
object LibreriaInitializer {
  private[manager] def inizializzaLibreria() {
        ...
    }
  }
```

```
object LibreriaManager {
      ...
  }
```

```
object Menu {
      ...
  }
```

Fig. 11: companion objects e membri statici

Design pattern: singleton

La classe *LibreriaManager* è la classe che gestisce la logica del programma: inizializza la libreria digitale all'avvio del programma inserendo diversi articoli in un campo denominato *vettArticoli* e fornisce metodi utili per la manipolazione di quest'ultimi. Per come è strutturata *LibreriaManager*, vi è la necessità di avere un un'unica istanza di questa classe in modo da garantire che le altre classi che operano sulla libreria digitale operino sempre sulla stessa istanza di *vettArticoli*. Per questo motivo, la classe LibreriaManager è stata progettata e implementata seguendo i principi espressi dal design pattern singleton: esiste solo una istanza di *LibreriaManager* e dunque una sola istanza del campo *vettArticoli*, sulla quale operano tutte le classi che utilizzano i metodi offerti da *LibreriaManager*. In Scala, per concretizzare il design pattern singleton, è sufficiente definire un object al posto della classe. Per questo motivo, per *LibreriaManager* esiste solo l'object e tutti i metodi e i campi sono definiti all'interno del corpo dell'object. Da notare inoltre che, dato che in Scala tutti i membri statici sono definiti all'interno dell'object, per come è strutturata l'architettura del programma, anche *Menu* si ritrova ad essere un singleton. Le classi *singleton* sono rappresentate in *Fig. 12*.

```
object LibreriaManager {
   ... campi e metodi ...
}
```

```
object Menu {
   ... campi e metodi ...
}
```

Fig. 12: LibreriaManager e Menu come singleton

Gestione delle eccezioni ed eccezioni personalizzate

La classe *LibreriaManager* presenta un campo *vettArticoli* di tipo *ListBuffer[ArticoloLibreria]* che contiene tutti gli articoli di tipo *Audio*, *Rivista* e *Libro* della libreria digitale. Per poter gestire tali articoli, sono stati implementati diversi metodi in *LibreriaManager* che operano su *vettArticoli*. Poiché spesso, per poter implementare le funzionalità richieste dai metodi di *LibreriaManager*, è necessario ricercare preliminarmente uno specifico articolo (per ID o nome) in *vettArticoli*, è stata definita una eccezione specifica per indicare la situazione nella quale un articolo cercato non è presente in *vettArticoli*. L'eccezione personalizzata che è stata definita è denominata *ArticoloNotFoundException* e fornisce il messaggio "Articolo non trovato". Questa eccezione viene lanciata (*throw*) dai metodi di *LibreriaManager* e gestita (*try – catch*) dai metodi di *Menu*. La sua definizione e un esempio del suo utilizzo sono mostrati in *Fig. 13*.

Classe ArticoloNotFoundException

```
class ArticoloNotFoundException extends Exception("Articolo non trovato"){

}
```

Classe LibreriaManager

```
def rimuoviArticolo(articoloID: Int) = {
  var trovato = false
  for (temp <- vettArticoli) {
    if (temp.getArticoloID == articoloID) {
      println("Articolo \"" + temp.getTitolo + "\" rimosso correttamente! ")
      vettArticoli -= temp
      trovato = true
    }
  }
  if (!trovato) {
    throw new ArticoloNotFoundException
  }
}
```

Classe Menu

```
private def rimuoviArticolo = {
  print("Inserisci ID dell'articolo che si vuole rimuovere >> ")
  val articoloID = scala.io.StdIn.readInt()

  println()

  try {
    LibreriaManager.rimuoviArticolo(articoloID);
  } catch {
    case e: ArticoloNotFoundException =>
      println("Errore. Articolo con ID_" + articoloID + " non trovato.")
  }

  println("\n")

  backToMenuScelta
}
```

Fig. 13: definizione ArticoloNotFoundException, throw e try - catch

Overload

L'overload di un metodo si ha quando il metodo preso in considerazione ha lo stesso nome di altri metodi, ma presenta una diversa definizione dei parametri (tipo e/o numerosità). Il compilatore sceglie, in fase di compilazione, quale metodo dovrà essere invocato nella successiva fase di esecuzione sulla base del tipo degli argomenti che vengono passati come parametri al metodo richiamato. Nel progetto, l'overload viene principalmente sfruttato per poter definire molteplici costruttori nelle diverse classi, ognuno dei quali prende più o meno parametri. In questo modo è possibile definire dei costruttori che consentono di istanziare oggetti che non hanno tutti i campi inizializzati ad uno specifico valore definito dall'utente. Esempi di overload dei costruttori sono riportati in *Fig. 14*.

```
class Audio(titolo: String, val autore: String, val casaDiscografica: String) extends
ArticoloLibreria(titolo) {

  //overload del costruttore
  def this(titolo:String, autore:String) = this(titolo, autore, null)

-----------------------------------------------------------------------------------------

class Libro(titolo: String, val autore: String, val casaEditrice: String, val numPagine: Int)
extends ArticoloLibreria(titolo) {

   //overload del costruttore
  def this(titolo:String, autore:String) = this(titolo, autore, null,0)

-----------------------------------------------------------------------------------------

class Rivista(titolo: String, val casaEditrice: String) extends ArticoloLibreria(titolo) {

   //overload del costruttore
  def this(titolo:String) = this(titolo, null)
```

Fig. 14: overload costruttori di Audio, Libro e Rivista

Override

Per avere override in Scala è necessario definire un metodo in una sottoclasse con stessa segnatura e valore ritornato (o un sottotipo del valore ritornato) di un metodo della superclasse e far precedere tale metodo dalla parola chiave *override*. Una volta definito l'override di un metodo, in fase di esecuzione il metodo da invocare verrà stabilito sulla base del tipo dell'istanza che effettivamente invocherà il metodo. Nel progetto, il metodo di cui si fa l'override è il metodo *toString()* di *ArticoloLibreria*. A runtime, questo metodo viene invocato quasi sempre attraverso variabili di tipo *ArticoloLibreria* (contenute nel *ListBuffer[ArticoloLibreria] vettArticoli*) e quindi, sulla base di quale sarà effettivamente l'istanza che invocherà il metodo, verrà richiamato il metodo *toString()* di *Audio, Libro* oppure *Rivista*. In *Fig. 15* viene riportato l'override del metodo *toString()*.

Classe Audio

```
override def toString = {
  val sTipo: String = getTipo + " (ID_" + getArticoloID + ") :";
  val sTitolo: String = "Titolo = " + getTitolo;
  val sAutore: String = "Autore = " + getAutore;
  val sCasaDiscografica: String = "Casa discografica = " + getCasaDiscografica;
  val sDisponibilita: String = "Disponibile = " + getDisponibilita;
  val sRes: String = sTipo + "\n" + sTitolo + "\n" + sAutore + "\n" + sCasaDiscografica+"\n" +
sDisponibilita + "\n"

  sRes
}
```

Classe Libro

```
override def toString = {
  val sTipo: String = getTipo + " (ID_" + getArticoloID + ") :"
  val sTitolo: String = "Titolo = " + getTitolo
  val sAutore: String = "Autore = " + getAutore
  val sCasaEditrice: String = "Casa editrice = " + getCasaEditrice
  val sNumPagine : String = "Pagine = " + getNumPagine
  val sDisponibilita: String = "Disponibile = " + getDisponibilita
  val sRes: String = sTipo + "\n" + sTitolo + "\n" + sAutore + "\n" + sCasaEditrice + "\n" +
sNumPagine + "\n" + sDisponibilita + "\n"

  sRes
}
```

Classe Rivista

```
override def toString = {
  val sTipo: String = getTipo + " (ID_" + getArticoloID + ") :";
  val sTitolo: String = "Titolo = " + getTitolo;
  val sCasaEditrice: String = "Casa editrice = " + getCasaEditrice
  val sDisponibilita: String = "Disponibile = " + getDisponibilita;
  val sRes: String = sTipo + "\n" + sTitolo + "\n" + sCasaEditrice + "\n" + sDisponibilita +
"\n"

  sRes
}
```

Fig. 15: override toString() di Audio, Libro e Rivista

Funzioni senza parametri

In Scala vige il *principio di accesso uniforme*: l'accesso a variabili e funzioni senza parametri deve avvenire utilizzando la stessa sintassi. Per questo motivo, quando definiamo o richiamiamo una funzione senza parametri, Scala non richiedere di specificare le parentesi tonde. Un esempio nel quale viene sfruttato il *principio di accesso uniforme* nel progetto è mostrato in *Fig. 16.*

```
  def getAutore: String = autore
  def getCasaEditrice: String = casaEditrice
  def getNumPagine: Int = numPagine

  override def toString = {
    val sTipo: String = getTipo + " (ID_" + getArticoloID + ") :"
    val sTitolo: String = "Titolo = " + getTitolo
    val sAutore: String = "Autore = " + getAutore
    val sCasaEditrice: String = "Casa editrice = " + getCasaEditrice
    val sNumPagine : String = "Pagine = " + getNumPagine
    val sDisponibilita: String = "Disponibile = " + getDisponibilita
    val sRes: String = sTipo + "\n" + sTitolo + "\n" + sAutore + "\n" + sCasaEditrice + "\n" +
sNumPagine + "\n" + sDisponibilita + "\n"

    sRes
  }
```

Fig. 16: principio di accesso uniforme (evidenziato)

Funzioni annidate

In Scala è possibile definire delle funzioni annidate. Una funzione B definita all'interno di un'altra funzione A può essere richiamata solo nel corpo di A. Nel progetto sono state spesso utilizzate, per questioni di praticità, delle funzioni annidate. Un esempio è riportato in *Fig. 17*.

Classe LibreriaManager

```
def printArticolo(articoloID: Int) = {

  def findArticolo(articoloID: Int) = {
    val tempList = vettArticoli.filter(x => x.getArticoloID == articoloID)
    if (tempList.size == 0)
      throw new ArticoloNotFoundException
    else
      println(tempList(0))
  }

  try {
    findArticolo(articoloID)
  } catch {
    case e: ArticoloNotFoundException =>
      println("Errore. Articolo con ID_" + articoloID + " non trovato.\n")
  }

}
```

Fig. 17: funzione findArticolo interna (giallo) alla funzione printArticolo (arancione)

Metodi generici

Nella classe *LibreriaManager* è stato ideato e implementato un metodo generico che consente di estrarre e stampare dal *ListBuffer[ArticoloLibreria] vettArticoli* dei "sotto-vettori" contenenti solo le istanze di uno specifico tipo tra *Audio*, *Libro* e *Rivista*. L'utilizzo del metodo generico rende il programma molto più flessibile in quanto con solo 1 metodo generico si consegue lo stesso risultato che si sarebbe potuto ottenere con la scrittura di 3 metodi non generici. Il metodo generico utilizzato per la generazione e la stampa dei "sotto-vettori" è mostrato in *Fig. 18*.

Classe LibreriaManager

```
//T sottotipo di ArticoloLibreria
def printSottocategoria[T <: ArticoloLibreria] ( sottoCategoria : TipoArticolo)  = {

  def getVettSottocategoria[T <: ArticoloLibreria] ( sottoCategoria : TipoArticolo) :
ListBuffer[T] = {
    var vettSottoCat: ListBuffer[T] = new ListBuffer()
    vettArticoli.foreach(x => if(x.getTipo==sottoCategoria) vettSottoCat += x.asInstanceOf[T])
    vettSottoCat
  }

  getVettSottocategoria[T](sottoCategoria).foreach(println)

}
```

Fig. 18: metodo generico printSottocategoria

Cast esplicito

In Scala, per poter realizzare un cast esplicito, è necessario utilizzare la clausola *asInstanceOf[_tipo_]*. Un esempio di utilizzo del cast esplicito è riportato nella figura soprastante (*Fig. 18*) dove viene effettuato il cast di un elemento di *vettArticoli* dal tipo *ArticoloLibreria* al tipo generico *T* che è una sottoclasse di *ArticoloLibreria*, ossia *Libro*, *Audio* oppure *Rivista*.

Visibilità in Scala

I membri di una classe in Scala sono pubblici per default e possiamo controllare la loro visibilità mediante l'utilizzo di apposite parole chiave, in modo simile a come si fa in Java. Nel progetto, per mostrare come si può modificare la visibilità in Scala, il metodo *inizializzaLibreria()* della classe *LibreriaInitializer* è stato definito con visibilità package. In questo modo riusciamo a far sì che il metodo che inizializza la libreria digitale all'avvio del programma possa essere invocato solo dalle altre classi contenute nel package *manager*, in particolare *LibreriaManager*. La parola chiave utilizzata per definire una visibilità di tipo package è illustrata in *Fig. 19*.

Package: manager
Classe: LibreriaInitializer

```
//private[manager] = visibilità package -> solo LibreriaManagar potrà richiamare questo metodo
private[manager] def inizializzaLibreria() {
```

Package: managaer
Classe: LibreriaManager

```
object LibreriaManager {

  //vettArticolo inizializzato in fase di load di LibreriaManager
  LibreriaInitializer.inizializzaLibreria();
```

Fig. 19: visibilità package per il metodo inizializzaLibreria()

Scala funzionale

Nel progetto vengono spesso utilizzate strutture tipiche di Scala funzionale, come ad esempio il *foreach* per iterare su un *ListBuffer* oppure *filter* per selezionare un sottoinsieme di oggetti che soddisfano specifiche condizioni. Alcuni dei costrutti tipici di Scala funzionale utilizzati nel progetto vengono presentati in *Fig. 20*.

Classe: LibreriaManager

```
def printArticolo(articoloID: Int) = {

  def findArticolo(articoloID: Int) = {
    val tempList = vettArticoli.filter(x => x.getArticoloID == articoloID)

-------------------------------------------------------------------------------------------------------


def printAllArticoli = {
  vettArticoli.foreach(println)
}

-------------------------------------------------------------------------------------------------------


ListBuffer[T] = {
      var vettSottoCat: ListBuffer[T] = new ListBuffer()
      vettArticoli.foreach(x => if(x.getTipo==sottoCategoria) vettSottoCat += x.asInstanceOf[T])
      vettSottoCat
    }
```

Fig. 20: esempio di costrutti di Scala funzionale utilizzati nel progetto