Appunti su Scala e Kojo (draft)

Massimo Maria Ghisalberti - pragmas.org

2016-03-05

Indice

1 Kojo

Kojo[1] è sviluppato da Lalit Pant (Himjyoti school, Dehradun - India) ed è utilizzato in varie scuole indiane, statunitensi, inglesi e svedesi.
L'approccio usato nella piattaforma Kojo è più ampio dei soliti ambienti per l'insegnamento. Può essere rivolto a più livelli di apprendimento ed è dotato di parti specifiche, per esempio per la sperimentazione in ambito matematico con un laboratorio basato su GeoGebra[2]. Il linguaggio utilizzato è Scala[3].

[1] http://www.kogics.net/kojo
[2] http://www.geogebra.org/cms/it/
[3] http://www.scala-lang.org/

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com

Scala è un linguaggio estremamente potente e multiparadigma (Orientato agli oggetti, funzionale) che può essere utilizzato a vari livelli, sufficientemente semplice nelle sue basi da poter essere insegnato in età scolare (dalla classe 4° primaria). La sua caratteristica di linguaggio funzionale lo fa particolarmente utile nella risoluzione di problemi matematici.

1.0.1 Installazione

Kojo è un applicativo scritto in Scala a cui serve che sia installata una Java Virtual Machine.
Grazie a questa tecnologia può funzionare sui maggiori sistemi operativi.

Oggi quasi tutti i sistemi operativi dovrebbero avere già al loro interno una macchina virtuale Java.
In ogni caso la trovate sul sito web di Oracle. A Kojo basta il runtime per funzionare, ma vi consiglio di installare il Java Development Kit, che è la distribuzione completa del minimo per far funzionare i programmi scritti in questo linguaggio più altri strumenti che volendo evolvere potrebbero essere interessanti (il compilatore per il linguaggio Java).

Consultando questo link JDK8 download[4] nella parte: *Java SE Development Kit 8u77*, potrete scaricare la versione adatta al vostro sistema operativo.
Windows x86 è la versione per i sistemi Microsoft a 32bit e quindi XP sicuramente e per gli altri quelli a 32bit.
Windows x64 è per sistemi a 64bit. Se avete un computer sufficientemente recente molto probabilmente sarà a 64bit. Cercate le informazioni sul sistema nel pannello di controllo se non sapete a che profondità di bit è il vostro sistema.
Ricordate che la versione a 32bit girerà in un sistema a 64bit ma non viceversa.

Una volta che avrete scaricato il file di installazione lanciatelo e seguite le istruzioni che fondamentelmente saranno: avanti, avanti, avanti… fatto.

A questo punto andate sul sito di Kojo[1] Anche qui avete gli installatori per i vari sistemi operativi, scegliete il vostro e scaricate il file. Potete anche scaricare il pacchetto dei Media Files che sono una serie di immagini e suoni derivati dal progetto Scratch[5].

Lanciate l'installatore di Kojo ed avanti, avanti… fatto.

Una volta installato il tutto, avrete due icone: Kojo-web e Kojo-desktop. Lanciate Kojo-desktop. Kojo-web usa una particolare tecnologia Java che permette di caricare il programma direttamente da Internet.
Usate Kojo-desktop.

2 Scala

Scala (da Scalable Language) è un linguaggio di programmazione di tipo general-purpose multi-paradigma studiato per integrare le caratteristiche e funzionalità dei linguaggi orientati agli oggetti e dei linguaggi funzionali. La compilazione di codice sorgente Scala produce Java bytecode per l'esecuzione su una JVM.
Scala è stato progettato e sviluppato a partire dal 2001 da Martin Odersky e dal suo gruppo all'École polytechnique fédérale de Lausanne (EPFL). È stato distribuito pubblicamente a gennaio 2004 sulla piattaforma Java e a giugno dello stesso anno sulla piattaforma .NET (ora non più supportata). La seconda versione del linguaggio è stata distribuita a marzo del 2006.

3 Documenti

Documentazione Kojo in italiano[6]
Documentazione Kojo in italiano (pdf)[7]
Questo documento in pdf[8]
Questo documento in epub[9]

[4] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
[5] https://scratch.mit.edu
[6] http://minimalprocedure.pragmas.org/writings/kojo-italiano-doc/kojo-it-doc.html
[7] http://minimalprocedure.pragmas.org/writings/kojo-italiano-doc/kojo-it-doc.pdf
[8] http://minimalprocedure.pragmas.org/writings/kojo-scala-appunti/kojo-scala-appunti.pdf
[9] http://minimalprocedure.pragmas.org/writings/kojo-scala-appunti/kojo-scala-appunti.epub

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com

3.1 Altri documenti sulla programmazione in altri linguaggi adatti ai bambini

Programmazione elementare in Ruby[10]

3.2 Documenti su Kojo e Scala

Challenges with Kojo (Sfide con Kojo) - Björn Regnell, Lund University, 2015[11]

3.3 Altri documenti di taglio più avanzato

Paradigmi di programmazione (presentazione)[12]
Programmazione funzionale, una semplice introduzione.[13]

4 Appunti

Quello che segue sono degli appunti in ordine sparso sui temi affrontati durante il corso tenuto per le ultime classi di scuola elementare. Gli esempi non sono spesso idiomatici per il linguaggio di programmazione Scala ma si è cercato di renderli il più comprensibili possibile per bambini intorno ai dieci/undici anni di età.
Consiglio di provare gli esercizi e le sfide del documento indicato sopra: *Challenges with Kojo (Sfide con Kojo) - Björn Regnell, Lund University, 2015*[14]. Molte sfide ed esempi sono stati tradotti per *Kojo in italiano* e mantenuti in lingua originale quando ritenuto indispensabile.

4.1 I computer sono macchine stupide

Gli elaboratori elettronici sono delle macchine stupide ed hanno bisogno di essere guidati come un bambino molto piccolo. A loro deve essere detto tutto quello che devono fare quasi fin nei minimi particolari. Il computer non ha ricordi persistenti e nasce quando si accende e muore quando si spenge. I suoi ricordi devono essere letti ogni volta da un sistema di memorizzazione alla accensione, un po' come se noi dovessimo alla mattina rileggere tutto il libro della nostra vita prima di uscire.

Programmare un computer significa doverlo istruire ogni volta per un compito o più da svolgere. Per cercare di eliminare o diminuire questa attività ripetitiva e noiosa sono stati inventati numerosi sistemi e *linguaggi di programmazione*.

Un linguaggio di programmazione è, come per i linguaggi naturali dell'uomo, una serie di regole e di parole che messe insieme con un criterio specifico permetterà alla macchina di imparare ad eseguire compiti più o meno complessi.

Il linguaggio di programmazione utilizzato per l'applicativo Kojo si chiama Scala.

4.2 Scala e i linguaggi di programmazione in generale

Scala è un linguaggio di programmazione per computer elettronici ed è considerato un linguaggio di *alto livello*. *Alto livello* significa che è più vicino a noi che alla macchina, quindi inventato per rendere il meno complesso possibile la programmazione dei computer.

Come detto prima è un vero e proprio linguaggio anche se *artificiale* (inventato direttamente dall'uomo), ha delle regole ed un suo vocabolario di parole anche se molto piccolo. Come noi formuliamo delle frasi usando le regole della nostra lingua lo facciamo in Scala od ogni altro linguaggio di programmazione.

Algoritmi + strutture dati = programmi - Niklaus Wirth

In questo titolo di un famoso libro si riassume abbastanza bene il cosa sia programmare.

[10] http://minimalprocedure.pragmas.org/writings/programmazione_elementare_ruby/corso_elementare_ruby.html
[11] https://bitbucket.org/bjornregnell/scaboo/raw/3744f6b18c582e36f8a726173738091b4e6cd6b4/kojobook/tex/book-it.pdf
[12] http://minimalprocedure.pragmas.org/writings/programming_paradigms/index.html
[13] http://minimalprocedure.pragmas.org/writings/programmazione_funzionale/programmazione_funzionale.html
[14] https://bitbucket.org/bjornregnell/scaboo/raw/3744f6b18c582e36f8a726173738091b4e6cd6b4/kojobook/tex/book-it.pdf

Noi manipoliamo *numeri (interi o con la virgola)*, *parole (chiamate stringhe di caratteri)* ed altri tipi, attraverso specifici *algoritmi (ovvero delle procedure o modi di fare)* per scrivere dei *programmi* che faranno eseguire al computer quello che vogliamo. Salvo errori.

4.3 Le istruzioni

Quelle che chiamiamo *istruzioni* sono i comandi che vengono dati al computer per insegnargli (istruirlo) su un particolare compito. Possono essere varie cose: funzioni, dichiarazioni, definizioni, parole chiave… Tutto quello che viene scritto è una *istruzione*.

4.4 Operatori

Tutti noi conosciamo le calcolatrici elettroniche o meno. I linguaggi di programmazione sono come delle calcolatrici molto potenti o meglio, permettono di trasformare una macchina stupida in una calcolatrice molto potente. Se avete un computer (anche vecchio) e poche nozioni di un linguaggio di programmazione moderno potete risparmiare soldi per una calcolatrice scientifica.

Tutti i linguaggi conoscono bene le operazioni aritmetiche e si comporteranno come ci hanno insegnato a scuola. L'*addizione* con il + si comporterà come previsto. La *sottrazione* con il -, la *moltiplicazione* con il * e la divisione con /.
La divisione in Scala è particolare e si comporta come quando facciamo le divisioni in colonna con o senza resto:

La divisione tra numeri interi darà come risultato un numero intero, per trovare il resto dobbiamo usare un altro operatore che è % (modulo).

```
scala> 23 / 4
res0: Int = 5 //quoziente della divisione intera

scala> 23 % 4
res1: Int = 3 //resto della divisione intera

scala> 24 / 4
res3: Int = 6 //quoto della divisione intera

scala> 24 % 4
res0: Int = 0 //resto della divisione intera
```

Per avere come risultato un numero con la virgola almeno uno degli operandi deve essere un numero con la virgola, in questo modo Scala saprà cosa noi desideriamo.

```
scala> 23 / 4.0
res1: Double = 5.75
```

Non esistono solo operatori *aritmetici*, ma anche *logici* o *booleani* come && che sta per *and* come nella frase: questo *e* quello; || che significa *or* nel senso di: questo *o* quello.
Gli operatori *logici* sono usati nei processi decisionali: se *questo è vero* fai qualcosa *altrimenti* fai qualcos'altro (if (vero) {…} else {…}). Esiste anche un operatore di *negazione*: !, ovvero *non*.

```
scala> !true
res27: Boolean = false
```

Altri operatori sono per esempio *maggiore* >, *minore* <, *uguaglianza* ==, *maggiore o minore e uguale* >= <=.

4.5 Valori, variabili e tipi di dati

Per meglio utilizzare i dati che andremo a manipolare il linguaggio ci fornisce la possibilità di assegnare dei *valori* o delle *variabili*.

```
scala> val pere = 12 // valore
pere: Int = 12

scala> var mele = 12 // variabile
mele: Int = 12
```

Pere è un valore dichiarato con la parola chiave *val* a cui è assegnata la quantità 12. Un valore non potrà più cambiare per tutta la vita del programma, si dice quindi che sarà costante (immutabile).
Mele invece è una variabile, *var*, quindi potrà cambiare per esempio assegnandogli un'altra quantità a mio piacere.

```
scala> mele = 32
mele: Int = 32 // successo, mele è una variabile

scala> pere = 32
<console>:11: error: reassignment to val // fallimento, pere è un valore
       pere = 32
```

Si possono assegnare i tipi di dati che vogliamo, una stringa di caratteri per esempio:

```
scala> val nome = "Anacleto Mitraglia"
nome: String = Anacleto Mitraglia
```

Una volta che abbiamo assegnato una quantità o una stringa ad una variabile questa sarà marchiata con quel tipo di dato:

```
scala> var nome = "Anacleto Mitraglia"
nome: String = Anacleto Mitraglia // nome è di tipo string e potrà contenere solo string

scala> nome = 12
<console>:11: error: type mismatch; // tentativo di riassegnamento con un intero
 found   : Int(12)
 required: String
       nome = 12
              ^
```

Scala è un linguaggio di programmazione *fortemente tipizzato* (al contrario di altri meno rigidi).

4.5.1 I tipi di dati

I numeri come abbiamo visto possono essere di diversi tipi: interi (*Int*) o con la virgola (*Double*). In realtà ci sono altri tipi fondamentali:

- Boolean true o false
- Byte 8-bit con segno (-2^7 to 2^7-1)
- Short 16-bit con segno (-2^15 to 2^15-1)
- Int 32-bit con segno (-2^31 to 2^31-1)
- Long 64-bit con segno (-2^63 to 2^63-1)
- Float 32-bit IEEE 754 singola precisione in virgola
- Double 64-bit IEEE 754 doppia precisione in virgola
- Char 16-bit senza segno (0 to 2^16-1)
- String sequenza di Char

Ne esistono molti perché sono svariate le cose possibili da fare. Il numero di *bit* indica la lunghezza in bit che servono per rappresentarli nella memoria del computer.
Il *Byte* per esempio è rappresentato con una sequenza di zeri ed uno (8-bit): 00000001 = 1, 000000010 = 2.
Il tipo *String* è una stringa di caratteri e cioè una sequenza di caratteri alfanumerici: dall'1 al 9, dalla a alla z e dalla A alla Z (i caratteri maiuscoli e minuscoli sono diversi). Essendo una sequenza è *indicizzata* e si può accedere al carattere che si vuole che compone la parola dato il suo indice. Come tutti i *tipo sequenza* l'indice partirà da zero:

```
scala> val nome = "pippo"
nome: String = pippo

scala> nome(0)
res12: Char = p

scala> nome(1)
res13: Char = i

scala> nome(2)
res14: Char = p

scala> nome(3)
res15: Char = p

scala> nome(4)
res16: Char = o

scala> nome(5)
java.lang.StringIndexOutOfBoundsException: String index out of range: 5
  at java.lang.String.charAt(String.java:658)
  at scala.collection.immutable.StringOps$.apply$extension(StringOps.scala:38)
  ... 33 elided
```

Cercando di accedere al numero di indice 5 si avrà un errore. Le stringe hanno un metodo *length* che restituisce la lunghezza della stringa e cioè il numero di caratteri di cui è composta e non va confuso con l'indice. Il valore massimo di indice sarà sempre uguale a *length - 1* perché il suo valore iniziale è zero.

```
scala> val nome = "pippo"
nome: String = pippo

scala> nome.length
res19: Int = 5
```

Per esempio una funzione (le funzioni le vedremo meglio dopo) didattica che somma le cifre di un numero (per implementare la prova del nove) dopo averlo convertito in una stringa:

```
def sommaCifre(numero: Int): Int = {
  val cifra = numero.toString
  var accumulatore = 0
  for (indice <- 0 to cifra.length - 1) {
    accumulatore = accumulatore + cifra(indice).asDigit
  }
  if (accumulatore > 9)
    accumulatore = sommaCifre(accumulatore)
  accumulatore
}
```

In seguito con una conoscenza migliore di Scala (ma si potrebbe forse anche fare di meglio):

```
def sommaCifre(numero: Int): Int = {
  val risultato = numero.toString.map(_.asDigit).foldLeft(0)(_ + _)
  if (risultato > 9) sommaCifre(risultato)
  else risultato
}
```

Funzione che sarebbe meglio scrivere semplicemente conoscendo un po' di aritmetica modulare:

```
def sommaCifre(numero: Int): Int = {
  numero % 9
}
```

4.6 Commenti

Nel codice che scriviamo possiamo sempre inserire dei commenti ed anzi, vi esorto a farlo per annotare quello che il vostro codice nei vari punti fa.

I commenti in Scala, come in altri linguaggi, sono del testo racchiuso tra `/*` e `*/`, oppure su tutta la riga con all'inizio `//`.

```
// commento che finisce in fondo alla riga
val a = 1

/* commento che può stare su
più righe
*/
val b = 2
```

4.7 Le funzioni

Il lavoro di insegnare ai computer cosa fare è spesso lungo, noioso e ripetitivo. Per questo motivo i linguaggi di programmazione ci mettono a disposizione molte *strutture* e possibilità di organizzare le istruzioni per poterle recuperare più e più volte.
le *funzioni* che a volte vengono chiamate anche *procedure*, *metodi*, *routine* a seconda del linguaggio che utilizziamo sono uno dei mattoni fondamentali della programmazione.
In Scala sono dichiarate con la parola chiave *def*: `/def/ nome (parametri)`.

```
scala> def somma(a: Int, b:Int) = a + b
somma: (a: Int, b: Int)Int

scala> somma(34, 12)
res2: Int = 46

scala> def uno_più_due() = 1 + 2 // funzione senza parametri in entrata
uno_più_due: ()Int

scala> uno_più_due()
res4: Int = 3
```

Le funzioni sono come delle stanze con due porte una per entrare ed una per uscire:

```
scala> def salone_di_bellezza(mamma: String) =
     | // succede di tutto
     | "mamma bella"
salone_di_bellezza: (mamma: String)String

scala> salone_di_bellezza("mamma brutta")
res5: String = mamma bella
```

La *mamma* entra *brutta* nel salone di bellezza e ne esce *bella*.
Le funzioni quindi possono prendere valori in entrata e dopo averli manipolati restituire un valore in uscita.
Le funzioni possono non avere nessun valore in entrata e restituire un valore oppure nessuno.
In scala una funzione che non restituisce un valore si dice che è di tipo *Unit*

```
scala> def nessun_valore_in_uscita() = println("scrivo qualcosa")
nessun_valore_in_uscita: ()Unit
```

Il tipo di ritorno *Unit* è un tipo speciale e per adesso va semplicemente ignorato.

Le funzioni in Scala restituisco l'ultimo valore assegnato e non c'è bisogno di una parola chiave *return* come in altri linguaggi.

```
scala> def f(): Int = {
     |    val a = 1
     |    val b = 2
     |    a
     | }
f: ()Int

scala> f
res26: Int = 1
```

Il valore *a* è alla fine della funzione e verrà restituito.

4.8 Cicli

Si fa uso molto spesso dei cicli per far compiere al computer operazioni ripetitive più volte. Per disegnare un quadrato in Kojo in italiano si usa il comando *ripeti*:

```
ripeti(4) {
  avanti(100);
  destra(90);
}
```

Il codice *avanti(100)* e *destra(90)* verrà ripetuto 4 volte e un quadrato di lato 100 sarà disegnato.
Raggruppando in una funzione:

```
def quadrato(lato: Int) = {
  ripeti(4) {
    avanti(lato)
    destra(90)
  }
}

quadrato(50)
quadrato(150)
quadrato(300)
```

Possiamo riutilizzare quel codice più volte e disegnare diversi quadrati. L'istruzione *ripeti* in Kojo in italiano è solo un modo semplice per scrivere. Usando le sole istruzioni di Scala (il ciclo *for*):

```
def quadrato(lato: Int) = {
  for (i <- 1 to 4) { // il valore i prenderà prima il valore 1, poi 2... infine 4
    avanti(lato)
    destra(90)
  }
}
```

Per curiosità su cosa significhi *1 to 4*:

```
scala> 1 to 4
res6: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4)

scala> 1 until 4
res7: scala.collection.immutable.Range = Range(1, 2, 3)
```

L'espressione *1 to 4* restituisce un tipo *Range* e cioè una sequenza di numeri: 1,2,3,4.
Sapendo questo potrei scrivere la mia funzione per il quadrato come:

```
def quadrato(lato: Int) = {
  (1 to 4).foreach { x =>
    avanti(lato)
    destra(90)
  }
}
```

Le sequenze possono essere molto utili.

```
scala> val nome = "pippo"
nome: String = pippo

scala> 0 to nome.length -1
res30: scala.collection.immutable.Range.Inclusive = Range(0, 1, 2, 3, 4)
```

Con l'espressione *0 to nome.length -1* ho la sequenza degli indici della stringa *"pippo"*.
Tutto questo per ribadire il concetto che ci possono essere molti modi di fare la stessa cosa.

4.9 Le condizioni

Quando programmiamo dobbiamo far prendere al nostro computer molte scelte, se succede *questo* allora…
In Kojo in italiano, abbiamo il comando: *se (questo è vero) {blocco di codice} altrimenti {blocco di codice alternativo}*.
Questa è una facilitazione di Kojo introdotta dal traduttore in italiano e non presente attualmente nelle altre lingue.

```
val colore = "rosso"

// istruzione su Kojo in italiano
se(colore == "verde") {
  println("il colore è verde")
} altrimenti {
  println("il colore non è verde")
}

// Struttura di Scala
if (colore == "verde") {
  println("il colore è verde")
} else {
  println("il colore non è verde")
}

// sarà stampato il risultato: "il colore non è verde"
// perché il test: colore == "verde" è falso essendo colore impostato a "rosso"
```

L'istruzione *se-altrimenti* corrisponde (con alcune limitazioni) alla struttura decisionale *if-else* di Scala e di molti altri linguaggi di programmazione.

La struttura standard Scala è più flessibile visto che posso scrivere:

```
val colore = "rosso"

// questo modo è troppo annidato ed è facile cadere nell'errore
se(colore == "verde") {
  println("il colore è verde")
} altrimenti {
  se(colore == "giallo") {
    println("il colore è giallo")
  } altrimenti {
    println("il colore non è verde e né giallo")
  }
}

// più compatto e facile da leggere
if (colore == "verde") {
  println("il colore è verde")
} else if (colore == "giallo") {
  println("il colore è giallo")
} else {
  println("il colore non è verde e né giallo")
}
```

Come si può vedere bene, l'uso di *if-else-if* è migliore dell'annidamento di *se-altrimenti*. La forma tradotta è stata proposta solo come introduzione al concetto di scelta alternativa in un linguaggio conosciuto, ma deve essere abbandonata il più presto possibile in favore delle strutture standard del linguaggio. L'istruzione *se-altrimenti* è una funzione come tante altre definibili ed in parte dimostra anche la potenza del linguaggio Scala nella creazione di costrutti per estendere il linguaggio di programmazione stesso.

Per mera curiosità, questo il frammento della implementazione.

```
class IfClauseExpression[T1, T2](fn1: => T1, fn2: => T2){
  lazy val pred = fn1
  lazy val compl = fn2
}

class IfThenClauseExpression[T](cond: => Boolean, thenFn: => T)
    extends IfClauseExpression(cond, thenFn) {
```

```
  def altrimenti[T](elseFn: => T) = {
    if(cond) thenFn else elseFn
  }
}

implicit def se[T](cond: => Boolean)(thenFn: => T) = {
  new IfThenClauseExpression(cond, thenFn)
}
```

4.10 Espressioni ed istruzioni

Una cosa interessante di Scala, insieme ad altri linguaggi di programmazione simili, rispetto ad altri linguaggi che vengono chiamati *imperativi* è la differenza su come si comportano le *istruzioni*.
Le istruzioni come *if-else* ma anche altre, sono in realtà delle espressioni e come tali restituiscono valori.

```
val colore = "bianco"

val risultato = if (colore == "blu")
  "il colore è blu"
else
  "il colore non è blu"

println(risultato)
```

Il blocco *if-else* restituisce un valore ed in questo caso la stringa: *il colore non è blu*. Valore che viene assegnato a *risultato*. Nel caso in cui non fosse stata una *espressione* avremmo dovuto scrivere:

```
val colore = "bianco"
var risultato = ""

if (colore == "blu")
  risultato = "il colore è blu"
else
  risultato = "il colore non è blu"

println(risultato)
```

Come vedete si è dovuto utilizzare una variabile e poi assegnarla con una stringa o con l'altra in base alla decisione.
Avere queste strutture in forma di *espressione* piuttosto che di semplici istruzioni è un grosso vantaggio perché ci permette di scrivere meno e di mantenere una forma del codice più logica e forse più vicina al nostro modo di pensare, almeno a mio avviso.

Si potrebbe scrivere a questo punto inserendo l'espressione *if-else* direttamente come parametro della funzione per scrivere *scriviLinea* o *println* (print line):

```
val colore = "bianco"

scriviLinea(
  if (colore == "blu")
    "il colore è blu"
  else
    "il colore non è blu"
)
```

Il concetto è che una espressione è trattabile come un valore e questo è un *modo matematico* di pensare. Il linguaggio di programmazione Scala è un linguaggio detto *funzionale* e cioè che mette in primo piano il concetto di *espressione*, *valore* e *funzione* nel modo matematico di concepirli. Per chi volesse approfondire un po' di più rimando ad una parte di un mio documento sulla programmazione funzionale in cui si spiega il più semplicemente possibile. Il documento indicato presuppone un *target* più elevato dei bambini a cui è rivolto questo.

4.11 Disegnamo delle figure geometriche con la tartaruga.

Abbiamo visto prima quando sono stati descritti i cicli, una funzione che disegna un quadrato.

```
def quadrato(lato: Int) = {
  ripeti(4) {
    avanti(lato)
    destra(90)
  }
}
```

Le *operazioni minime* per far disegnare un quadrato alla tartaruga sono il portarla in *avanti* per quanti passi vogliamo sia lungo il lato e poi di farla voltare a *destra* o a *sinistra* di 90° gradi. Queste operazioni le *ripetiamo* per quattro volte.
Lo stesso meccanismo lo possiamo applicare per disegnare un qualunque poligono regolare di qualunque numero di lato fino a… un cerchio.
Se ci riflettete bene un cerchio non è altro che un poligono composto da una miriade di punti e con altrettanti angoli.
Questa è una cosa che non va detta alla maestra, che a ragione, si è sforzata di insegnarvi che un cerchio non è un poligono ma una figura piana delimitata da una circonferenza e che questa è una linea curva chiusa i cui punti sono tutti equidistanti da un cerchio.

Programmare però significa avere un approccio pratico al problema e per noi questa *praticità* adesso, è considerare un cerchio come un poligono formato da parecchio lati e parecchi angoli.
Quanti però sono?

```
ripeti(360) {
  avanti(1)
  destra(1)
}
```

Prima il ciclo veniva ripetuto per quattro volte quanti sono i lati del quadrato, per il cerchio trecentosessanta quanti sono i gradi di una angolo giro. La tartaruga avanzerà di un punto e si girerà di un grado e così via per trecentosessanta volte.
Abbiamo il cerchio.
Abbiamo trattato il cerchio come un poligono ed eccolo qua.

Se lo volessi fare più grande o più piccolo?

```
ripeti(360) {
  avanti(2)
  destra(1)
}
```

Provate e guardate che succede, ma se invece aumentassi l'angolo? Osservate le cose interessanti…
Per approfondire e vedere come alla fine il cerchio sia solo un caso limite di poligono regolare vi rimando a questo articolo su wikipedia anche se ostico per i bambini.
Insomma, con questo piccolo pezzetto di codice potremmo disegnare tutti i poligoni che vogliamo.

Un triangolo:

```
ripeti(3) {
  avanti(100)
  destra(120)
}
```

Il triangolo non ha qualcosa di strano? La somma degli angoli di un triangolo è 180° e noi giriamo a destra di 120° per ben tre volte.
La nostra tartaruga avanza inizialmente per cento passi poi si volta a destra di 120°. L'angolo di cui si gira non è quindi l'angolo interno di 60° ma bensì quello esterno:

```
120° + 60° = 180°
```

La tartaruga si gira quindi sempre considerando l'angolo esterno e sapendo questo:

```
ripeti(6) {
  avanti(100)
  destra(60)
}
```

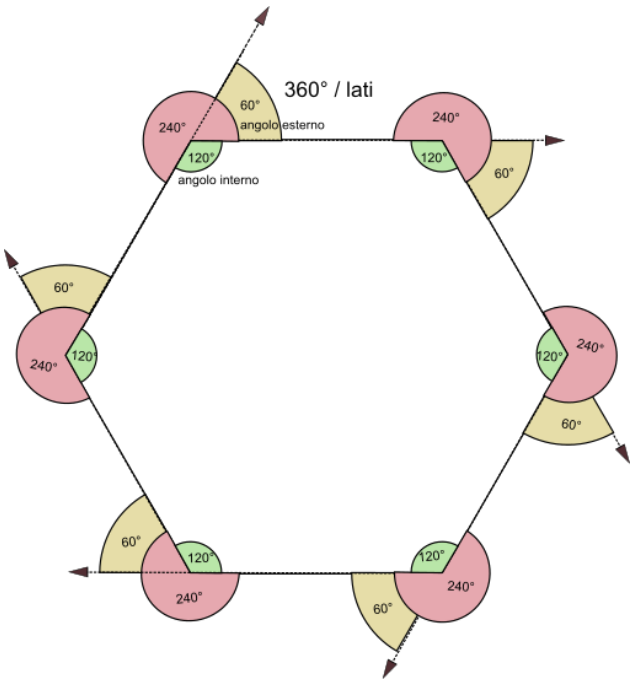
Ecco un esagono. Tutto però perché dovremmo sapere che l'angolo interno di un esagono è 120° e quindi `180° - 120° = 60°`.

Quello che vorrei fare però è avere una funzione che mi disegni un qualunque poligono regolare, dai tre lati fino al nostro caso limite: il cerchio.

```
def poligono(lato: Double, lati: Int) {
  val angolo = 360 / lati.toDouble;
  // controllo se il numero dei lati è pari o dispari
  seVero(lati % 2 >= 0) {
    sinistra(90 - angolo);
  }

  ripeti(lati) {
    avanti(lato);
    destra(angolo);
  }
}

poligono(100, 3) // triangolo
poligono(100, 5) // pentagono
poligono(1, 360) // cerchio
```

Qui bisogna sapere che ci sono delle relazioni matematiche nei poligoni regolari, tra il numero dei lati ed i suoi angoli esterni ed interni.
L'angolo esterno che è racchiuso dal prolungamento ideale del lato precedente con il lato seguente è, considerando l'angolo piatto formato tra esso e l'angolo interno, pari a 360° diviso il numero dei lati che compongono il poligono. Un po' di magia matematica insomma.
Ci sono alcuni modi per dimostrare questo ma si può partire dal fatto che ogni poligono regolare ha all'interno una serie di triangoli il cui vertice è il centro del poligono, componendo queste figure si può risalire secondo le regole dei triangoli all'angolo interno e da questo a quello esterno. Per una dimostrazione vi rimando questo articolo su wikipedia. Sapendolo diventa facile scoprire l'angolo e far girare la tartaruga di conseguenza.

Questo uno schema degli angoli di un esagono:



In quella funzione c'è un trucchetto ulteriore, controllo se il numero dei lati è pari o dispari. Lo faccio usando una operazione di modulo % e cioè quella che mi restituisce il resto della divisione intera: se il risultato è maggiore di zero il numero sarà dispari altrimenti pari. A che serve questo? Mi piaceva avere la base più o meno sempre orizzontale.
Togliete il test *seVero* e guardate che succede.
Ho usato per la lunghezza del lato un valore di tipo *Double*, con la virgola quindi, perché la lunghezza potrebbe essere una frazione di un intero mentre per il numero dei lati no. Un poligono di sei lati e mezzo non s'è ancora mai visto.

4.12 FizzBuzz

Questo è un giochino che serve ad insegnare le divisioni e la divisibilità.

La regola del gioco è semplice: ci si siede in cerchio e si inizia a turno a dire i numeri: 1, 2…
La difficoltà consiste nel dire al posto dei numeri divisibili per tre: Fizz; mentre per quelli divisibili per cinque: Buzz. Non è finita però, per quelli divisibili per tre e per cinque: FizzBuzz.

Insomma sarà… 1, 2, Fizz, 4, Buzz, Fizz… per il 15 sarà quindi FizzBuzz.

Cerchiamo di ragionare sul come fare un programma che scriva una serie di numeri insieme a Fizz, Buzz e FizzBuzz.

Intanto dobbiamo fare un ciclo per prendere un numero per volta no? Allora:

```
val sequenzaDiNumeri = 1 to 100

ripetiPerOgniElementoDi(sequenzaDiNumeri) { numero =>
  scriviLinea(numero)
}
```

Questo scriverà tutti i numeri da 1 a 100. La forma `1 to 100` l'abbiamo già vista e serve per creare una sequenza di numeri che parte da 1 ed arriva fino a 100, se avessimo scritto `5 to 10` sarebbe stata da 5 a 10. Non è difficile.

Ora che abbiamo il nostro ciclo, doppiamo mettere dei controlli per sapere se i numeri sono divisibili per tre o cinque o tutti e due.

```
def divisibile(num: Int, div: Int): Boolean = {
  (num % div) == 0
}

val sequenzaDiNumeri = 1 to 100

ripetiPerOgniElementoDi(sequenzaDiNumeri) { numero =>

  if (divisibile(numero, 3) && divisibile(numero, 5))
    scriviLinea("FizzBuzz")
  else if (divisibile(numero, 3))
    scriviLinea("Fizz")
  else if (divisibile(numero, 5))
    scriviLinea("Buzz")
  else
    scriviLinea(numero.toString)

}
```

Per prima cosa abbiamo bisogno, per rendere il codice più pulito di una funzione che controlla se un numero è divisibile per un altro e come si fa?
Cosa fa un numero divisibile per un altro? Lo abbiamo visto già prima: un numero è divisibile per un altro se la divisione intera non da resto.

```
def divisibile(num: Int, div: Int): Boolean = {
  // num diviso div da resto? Ovvero il suo resto è zero?
  (num % div) == 0
}
```

Questa funzione restituirà il valore *vero (true)* se è divisibile altrimenti restituirà *falso (false)*. Sono dei valori di tipo *Boolean (booleani)* e possono essere solo di due quantità: vero o falso. Scrivere questa funzione mi rende il codice più pulito e capibile perché ci sono meno simboli tutti insieme.

```
val sequenzaDiNumeri = 1 to 100

ripetiPerOgniElementoDi(sequenzaDiNumeri) { numero =>

  // senza funzione di appoggio
  if ((numero % 3) == 0 && (numero % 5) == 0)
    scriviLinea("FizzBuzz")
  else if ((numero % 3) == 0)
    scriviLinea("Fizz")
  else if ((numero % 5) == 0)
    scriviLinea("Buzz")
  else
    scriviLinea(numero.toString)

}
```

Questo è un caso semplice e comunque si potrebbe scrivere ancora meglio, guardate gli esempi in fondo al documento.
Comunque… Si controlla che il numero per prima cosa sia divisibile per tre e per cinque, se non lo è si controlla se lo è per tre, se non lo è si controlla per cinque, se non è nemmeno questo il caso si scrive il numero. Nella *cascata delle possibilità* siamo partiti dal caso più *largo* (tutti e due i numeri) a quello più stretto come fosse un imbuto.

Ci sono molti modi di risolvere questo tipo di problemi e questo è forse il più semplice e meno elegante, anche nell'ambito del linguaggio che stiamo usando.
Intanto usando il ciclo *for* di Scala e non la funzione di Kojo si sarebbe potuto scrivere:

```
val sequenzaDiNumeri = 1 to 100

// comando Kojo
ripetiPerOgniElementoDi(sequenzaDiNumeri) { numero =>
  scriviLinea(numero)
}

// ciclo for di Scala
for (numero <- sequenzaDiNumeri) {
  scriviLinea(numero)
}

// ma anche usando sempre Scala...
sequenzaDiNumeri.foreach { numero =>
  scriviLinea(numero)
}
```

L'ultima forma sarebbe quella preferibile e significa: per ogni elemento di quella sequenza fai qualcosa. È molto simile al comando di Kojo ma scritta in quella forma forse rende meglio l'idea di che cosa faccia, a parte la lingua italiana.

Vi mostro un'altra forma più complicata all'apparenza ma utile nel caso in cui i test fossero molti di più.

```
def fizzBuzz(da: Int, a: Int) {
  for (numero <- da to a) {

    val resti = (numero % 3, numero % 5)

    val messaggio = resti match {
      case (0, 0) => "FizzBuzz"
      case (0, _) => "Fizz"
      case (_, 0) => "Buzz"
      case (_, _) => numero.toString
    }
    scriviLinea(messaggio)
  }
}
```

La variabile resti conterrà i resti della divisione del numero per tre e per cinque. Quindi se il `numero = 5` sarà `(2, 0)` perché `5 / 3 = 1 con il resto di 2`, mentre `5 / 5 = 1 con il resto di 0`.
La forma *match* serve per controllare che il valore sia uno dei vari casi rappresentato da *case*.

Se prendiamo l'esempio di prima: `resti = (2, 0)`:

- nel caso in cui `resti corrisponde a (0, 0) allora FizzBuzz`
- nel caso in cui `resti corrisponde a (0, _) allora Fizz`
- nel caso in cui `resti corrisponde a (_, 0) allora Buzz`
- nel caso in cui `resti corrisponde a (_, _) allora numero`

Il carattere _ serve come segnaposto e vuol dire *qualunque valore*, quindi viene usato per indicare un valore diverso da zero.
Facciamo un esempio con una tabella.

numero	numero % 3	numero % 5	scrivo
1	1	1	1
2	2	2	2
3	0	3	Fizz
4	1	4	4
5	2	0	Buzz

Se il valore a sinistra è zero si scrive *Fizz* mentre se quello a destra è zero si scrive *Buzz* e se lo sono tutti e due *FizzBuzz*.
Se prima il paragone era con l'imbuto ora dovrebbe essere con lo scolapasta o meglio con lo *staccio* o *setaccio*.

Come ho detto più volte ci sono sempre molti modi di scrivere le cose e si dovrebbe però scegliere quello più adatto a risolvere i problemi. Abbiamo molti strumenti diversi ed alcuni sono più adatti di altri.

Possiamo stringere il bullone del tubo che perde con le pinze o una chiave inglese, con le pinze a lungo andare rovineremo il bullone e saremo costretti prima o poi a cambiarlo.
Con la chiave giusta durerà molto di più.

Una variante del gioco è il *FizzBuzzWoof* e cioè divisibile per tre, per cinque e per sette e loro combinazioni.

```
1  def fizzBuzzWoof(da: Int, a: Int) {
2    for (numero <- da to a) {
3      val resti = (numero % 3, numero % 5, numero % 7)
4      val messaggio =  resti match {
5        case (0, 0, 0) => "FizzBuzzWoof"
6        case (0, 0, _) => "FizzBuzz"
7        case (0, _, 0) => "FizzWoof"
8        case (0, _, _) => "Fizz"
9        case (_, 0, 0) => "BuzzWoof"
10       case (_, 0, _) => "Buzz"
11       case (_, _, 0) => "Woof"
12       case (_, _, _) => numero.toString
13     }
14     scriviLinea(messaggio)
15   }
16 }
```

Come si vede bene il nostro setaccio adesso è più complicato ma una volta compreso il meccanismo molto più semplice di tanti *if-else-if-else*… Provate a scriverlo con gli *if-else* per esercizio.

4.13 Un personaggio svolazzante

Vediamo come si potrebbe fare un personaggio caricando delle immagini alternative alla solita *tartaruga* ed a farlo muovere con i tasti di direzione che si trovano alla destra della tastiera.
Per fare questo useremo le cose già dette in precedenza ed alcuni concetti nuovi.

Il codice avrà bisogno della versione 2.0.49 di Kojo, quindi nel caso ne abbiate una vecchia aggiornatela. Si potrebbe anche con la 2.0.48 ma con degli accorgimenti mancando una funzione tradotta in italiano.

Intanto vi dico che si può accedere alle funzioni originali della tartaruga, quelle in inglese, scrivendo:

```
// tartaruga originale
englishTurtle.forward(100)

// tartaruga in italiano
avanti(100)
```

Le due righe di comando si equivalgono perché la tartaruga italiana è in un certo senso *figlia* di quella inglese (non lo è nel senso del linguaggio di programmazione dove le parentele hanno un significato ben preciso che non abbiamo ancora affrontato).
Per esempio usando la versione 2.0.48 di Kojo in cui non era ancora stata tradotta la funzione `setCostumeImages`, per poterla usare avreste dovuto scrivere in questo modo:

```
def loadImages(fnameA: String, fnameB: String) {
  import net.kogics.kojo.util.Utils.loadImage
  val a = loadImage(fnameA)
  val b = loadImage(fnameB)

  // la funzione setCostumeImages non è tradotta
  tarta.englishTurtle.setCostumeImages(a, b)
}
```

Mentre invece con la nuova versione:

```
def vestiPersonaggio(personaggio: Tartaruga, immagini: List[Image]) {
  if (COSTUMI != immagini) {
    COSTUMI = immagini

    // la funzione setCostumeImages è tradotta
    personaggio.indossaImmagini(immagini.toVector)
  }
}
```

Le funzioni scritte sopra sono diverse, ma si deve notare la traduzione nelle righe indicate. Il primo esempio era stato scritto per la prima versione del *personaggio svolazzante* che è poi stato migliorato e reso più *generico*.

Per prima cosa cerchiamo di capire cosa significhi `import net.kogics.kojo.util.Utils.loadImage`.
La *direttiva* `import` serve per rendere disponibili al codice che stiamo scrivendo delle ulteriori funzionalità. Immaginate di avere delle scatole che contengono rispettivamente: forbici, colori, nastro adesivo.
Vogliamo usare il foglio per fare una bambolina.

- Prendiamo un foglio.
- Apriamo la scatola delle forbici.
- Ritagliamo le varie parti del corpo.
- Apriamo la scatola dei colori
- Coloriamo i pezzi del corpo
- Apriamo la scatola del nastro adesivo
- Attacchiamo le varie parti.

Questo è quello che succede, `import` importa all'interno dello spazio di lavoro delle funzionalità. Se scrivessimo come *pseudocodice* il nostro lavoro per la bambolina:

```
import Scatole.Forbici

val partiDelCorpo = tagliaIlFoglio(foglio)
/* partiDelCorpo è adesso una lista:
  partiDelCorpo := List[PartiDelCorpo](testa, corpo, gambe, braccia)
*/
import Scatole.Colori

coloraDiRosa(partiDelCorpo(0)) // testa
coloraDiRosso(partiDelCorpo(1)) // corpo
coloraDiVerde(partiDelCorpo(2)) // gambe
coloraDiRosso(partiDelCorpo(3)) // braccia

import Scatole.NastroAdesivoTrasparente

attaccaParti(partiDelCorpo)
```

Questo codice non funzionerà ma è una descrizione più vicina al linguaggio di programmazione della lista di azioni di prima. Questo modo si dice *pseudocodice* è cioè una forma di organizzare gli eventi in una via di mezzo tra il semplice parlarne ed il programmare. È molto utile per quando si voglia elaborare un algoritmo per risolvere un problema.
Come si vede `import` carica dal *pacchetto (package)* delle Scatole le *Forbici, i Colori ed il NastroAdesivoTrasparente* così che noi li potessimo usare.

Per il nostro *personaggio* abbiamo bisogno di alcune funzionalità che sono in varie scatole:

```
import net.kogics.kojo.util.Utils.loadImage
import net.kogics.kojo.picture.ImagePic
import net.kogics.kojo.core.Point
```

Per caricare dal disco rigido delle immagini abbiamo bisogno di utilizzare la funzione `loadImage` che fa quello che ci serve:

```
/*
  Immagini del personaggio per le varie direzioni
*/
val COSTUMI_SU = List[Image](
  loadImage("./immagini/up-a.png"),
  loadImage("./immagini/up-b.png")
)

val COSTUMI_SINISTRA = List[Image](
  loadImage("./immagini/left-a.png"),
  loadImage("./immagini/left-b.png")
)

val COSTUMI_DESTRA = List[Image](
  loadImage("./immagini/right-a.png"),
  loadImage("./immagini/right-b.png")
)

val COSTUMI_GIU = List[Image](
  loadImage("./immagini/down-a.png"),
  loadImage("./immagini/down-b.png")
)
```

La funzione prende un percorso di *file* di immagine e lo carica nella memoria del computer. Il percorso del file è qui preso nella notazione *Unix*. Per chi usa Microsoft Windows può sembrare strana, visto che sarà abituato alla *barre* al contrario: *backslash*. Scala e la Java Virtual Machine su cui si esegue sono *multipiattaforma*, cercano cioè di poter essere eseguiti su più sistemi operativi possibile e quindi sono costretti ad uniformare certi meccanismi tra cui il come indicare i percorsi dei file sul disco rigido (o altro sistema di memorizzazione di massa).

Il modo di indicare i percorsi però non sarà così *alieno* perché se ci fate caso quando *navigate* in Internet anche gli *URL (Uniform Resource Locator)* sul browser sono così. I link alle pagine Web che consultate: `http://minimalprocedure.prag`
`scala-appunti/kojo-scala-appunti.html`.
Si usano le barre normali come vedete. Il link è un percorso di file e questo vuol dire che nel computer che si chiama

`minimalprocedure.pragmas.org` ci sarà una cartella `writings` con una sotto cartella `kojo-scala-appunti` che contiene il file `kojo-scala-appunti.html`.
La parte iniziale `http://` indica il protocollo di trasferimento dati da utilizzare per accedere a quel documento ed in questo caso è quello utilizzato per il World Wide Web (HTTP: HyperText Transfer Protocol).

Abbiamo caricato le immagini dentro una lista che è chiamata `List`, Scala però vuole sapere anche che tipo di lista è ed è molto rigido in questo. Se vi dicessi: "Voglio una scatola!"
Voi mi chiedereste probabilmente: "Una Scatola di cosa?"

Anche noi dobbiamo dire a Scala che quella è una lista di Image, il tipo che denota le immagini.

In questo programma si definiranno parecchie funzioni perché poi le monteremo come ingranaggi di una macchina.

Lasceremo la tartaruga predefinita che nasconderemo subito con il comando `invisibile` e ne andremo a fare un'altra perché che ne serve una un po' più speciale.

```
/*
  Personaggio
*/

val COORDINATE_STAGE = stage.bounds
val COORDINATE_PERSONAGGIO = Point(COORDINATE_STAGE.x + 100, 0.0)
val COORDINATE_AMBIENTE = Point(3000 / 2 -  COORDINATE_STAGE.width / 2, 0.0)

def nuovoPersonaggio(): Tartaruga = {
  val personaggio = nuovaTartaruga(COORDINATE_PERSONAGGIO.x, COORDINATE_PERSONAGGIO.y)
  rettangoloDiCollisionePersonaggio = PicShape.image(loadImage("./immagini/collision-bordo.png"))
  rettangoloDiCollisionePersonaggio.setPosition(personaggio.posizione.x - 50, personaggio.posizione.y - 71)
  draw(rettangoloDiCollisionePersonaggio)
  animaPersonaggio(personaggio)
  personaggio
}

def animaPersonaggio(personaggio: Tartaruga) {
  personaggio.fai { t =>
    ripetiFinché(true) {
      pause(velocitàAnimazionPersonaggio)
      t.prossimoCostume()
    }
  }
}
```

Inizialmente ci sono dei valori *costanti*. La parola chiave `val` sappiamo già che crea un valore ma per rendere più evidente che sono qualcosa che non cambierà mai le mettiamo tutto in maiuscolo. È un modo comune di indicare per far capire a prima vista che quelli sono valori di riferimento.
Lo `stage` è il rettangolo dove la tartaruga cammina e `stage.bounds` mi restituisce i limiti come l'altezza e la larghezza o le coordinate x ed y.
Nella funzione `nuovoPersonaggio` si crea una nuova tartaruga con `nuovaTartaruga` una funzione a cui devono essere fornite le coordinate x ed y di dove collocarla. Queste coordinate vengono calcolate partendo da quelle dello stage, per cui indipendentemente da quando lo *stage* sia grande il personaggio sarà sempre nel lato sinistro e spostato di 100 pixel verso destra: `val COORDINATE_PERSONAGGIO = Point(COORDINATE_STAGE.x + 100, 0.0)`. L'altezza va bene quella standard per cui è `0`. `Point` è un tipo *punto* che contiene due valori: x ed y; ovvero le coordinate di quel punto.

Una cosa interessante è `rettangoloDiCollisionePersonaggio` perché noi vogliamo sapere quando il personaggio colpisce degli oggetti nella scena. Per poterlo fare dobbiamo legare alla nostra tartaruga un oggetto `PicShape` che è in grado di capire se sta toccando qualcosa. Queste sono funzionalità che ci mette a disposizione Kojo attraverso la direttiva `import` di prima.

Per adesso abbiamo quindi una funzione che crea una nuova tartaruga e che chiameremo da ora in poi *personaggio* visto che gli cambiamo l'immagine che lo rappresenta. Abbiamo anche una funzione che lo animerà scorrendo le varie immagini che gli metteremo.
La funzione `animaPersonaggio`, accoglierà come parametro un personaggio creato con la funzione `nuovoPersonaggio` e farà partire un *loop* infinito in cui ne viene cambiata l'immagine visibile (o *costume* nel gergo Kojo).

Per far *indossare* una lista di *costumi* al nostro personaggio:

```
val COSTUMI_SU = List[Image](
  loadImage("./immagini/up-a.png"),
  loadImage("./immagini/up-b.png")
)

def vestiPersonaggio(personaggio: Tartaruga, immagini: List[Image]) {
    personaggio.indossaImmagini(immagini.toVector)
  }
}

// la funzione assegnerà al personaggio la lista di immagini COSTUMI_SU
vestiPersonaggio(personaggio, COSTUMI_SU) {
```

Il *loop* quindi farà vedere, una volta caricata la lista delle immagini, una immagine per volta:

- up-a.png
- aspetta un po'
- up-b.png
- aspetta un po'
- up-a.png
- aspetta un po'
- up-b.png
- aspetta un po'
- up-a.png
- aspetta un po'
- …

Nel nostro caso dove le immagini sono di un pipistrello con le ali su (up-a.png) e con le ali giù (up-b.png) ci sembrerà di vederlo svolazzare.

Se vi siete chiesti perché lo chiami personaggio e non pipistrello è presto detto: voglio fare un programma in cui la logica sia la stessa ma ne si possa cambiare l'aspetto semplicemente sostituendo le immagini. Voglio un programma generico. Se invece di avere un pipistrello che svolazza voglio una astronave con l'antenna che si muove mi basterà sostituire sul disco le immagini e rilanciare il programma. Per capirci, un panino è qualcosa dentro due fette di pane, se ci metto il salame sarà un panino col salame ma se ci metto il formaggio sarà un panino col formaggio.

Questo personaggio lo vogliamo spostare con la tastiera abbiamo detto e quindi vediamo come interagire con quella.

Ogni volta che premete un tasto sulla tastiera il sistema operativo riceve un *codice* che è spesso un numero. Questo *codice* può essere letto dai programmi che facciamo. L'operazione è una cosa un po' noiosa perché spesso consiste nel fare molti test per vedere quale dei tanti tasti è premuto.

Noi controlleremo solo quelli che ci interessano.

```
/*
Direzioni
*/
val STOP = 0
val SINISTRA = 100
val DESTRA = 200
val ALTO = 300
val BASSO = 400
val SINISTRA_ALTO = 500
val SINISTRA_BASSO = 600
val DESTRA_ALTO = 700
```

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com

```
12 val DESTRA_BASSO = 800
13 var DIREZIONE = ALTO
14
15 def direzionePersonaggio(): Int = {
16   DIREZIONE = if (isKeyPressed(Kc.VK_LEFT) && isKeyPressed(Kc.VK_UP))
17     SINISTRA_ALTO
18   else if (isKeyPressed(Kc.VK_LEFT) && isKeyPressed(Kc.VK_DOWN))
19     SINISTRA_BASSO
20   else if (isKeyPressed(Kc.VK_RIGHT) && isKeyPressed(Kc.VK_UP))
21     DESTRA_ALTO
22   else if (isKeyPressed(Kc.VK_RIGHT) && isKeyPressed(Kc.VK_DOWN))
23     DESTRA_BASSO
24   else if (isKeyPressed(Kc.VK_LEFT))
25     SINISTRA
26   else if (isKeyPressed(Kc.VK_RIGHT))
27     DESTRA
28   else if (isKeyPressed(Kc.VK_UP))
29     ALTO
30   else if (isKeyPressed(Kc.VK_DOWN))
31     BASSO
32   else
33     STOP
34   DIREZIONE
35 }
```

La cosa importante di questo frammento di codice è `isKeyPressed`. Questa funzione controlla se un tasto è stato premuto. I tasti si chiamano `Kc.<qualcosa>` e per la verità sono dei semplici numeri, sono dei *nomi di valore* come quelli che facciamo noi. Per esempio guardando la lista sopra la funzione `direzionePersonaggio`, `STOP` è uguale a `0` e possiamo usare quel nome dovunque sia aspettato un numero intero.
Visto che si dovrebbe fare questa lista di test molte volte nel programma ne ho fatto una funzione che mi restituisce una serie di valori che credo siano facilmente intuibili: `STOP`, `SINISTRA`, `DESTRA`, …
Ognuno di quei valori mi indica la direzione in cui va il personaggio.

Mi serve quando devo *vestire* il personaggio a seconda se vada su o giù, sinistra o destra:

```
1  def adattaCostumiPersonaggio(personaggio: Tartaruga) {
2    direzionePersonaggio() match {
3      case SINISTRA       => vestiPersonaggio(personaggio, COSTUMI_SINISTRA)
4      case DESTRA         => vestiPersonaggio(personaggio, COSTUMI_DESTRA)
5      case ALTO           => vestiPersonaggio(personaggio, COSTUMI_SU)
6      case BASSO          => vestiPersonaggio(personaggio, COSTUMI_GIU)
7      case SINISTRA_ALTO  => vestiPersonaggio(personaggio, COSTUMI_SINISTRA)
8      case SINISTRA_BASSO => vestiPersonaggio(personaggio, COSTUMI_SINISTRA)
9      case DESTRA_ALTO    => vestiPersonaggio(personaggio, COSTUMI_DESTRA)
10     case DESTRA_BASSO   => vestiPersonaggio(personaggio, COSTUMI_DESTRA)
11     case _              => vestiPersonaggio(personaggio, COSTUMI_SU)
12   }
13 }
```

Se non avessi usato la strategia di prima avrei dovuto fare la serie di test anche qui mentre invece mi posso limitare ad una lista di azioni da intraprendere. Il codice è più pulito e semplice da mantenere e da leggere.
L'avrei dovuta scrivere in questo modo:

```
1  def adattaCostumiPersonaggio(personaggio: Tartaruga) {
2    if (isKeyPressed(Kc.VK_LEFT) && isKeyPressed(Kc.VK_UP))
3      vestiPersonaggio(personaggio, COSTUMI_SINISTRA)
4    else if (isKeyPressed(Kc.VK_LEFT) && isKeyPressed(Kc.VK_DOWN))
5      vestiPersonaggio(personaggio, COSTUMI_SINISTRA)
6    else if (isKeyPressed(Kc.VK_RIGHT) && isKeyPressed(Kc.VK_UP))
7      vestiPersonaggio(personaggio, COSTUMI_DESTRA)
8    else if (isKeyPressed(Kc.VK_RIGHT) && isKeyPressed(Kc.VK_DOWN))
9      vestiPersonaggio(personaggio, COSTUMI_DESTRA)
10   else if (isKeyPressed(Kc.VK_LEFT))
11     vestiPersonaggio(personaggio, COSTUMI_SINISTRA)
12   else if (isKeyPressed(Kc.VK_RIGHT))
13     vestiPersonaggio(personaggio, COSTUMI_DESTRA)
```

```
  else if (isKeyPressed(Kc.VK_UP))
    vestiPersonaggio(personaggio, COSTUMI_SU)
  else if (isKeyPressed(Kc.VK_DOWN))
    vestiPersonaggio(personaggio, COSTUMI_GIU)
  else
    vestiPersonaggio(personaggio, COSTUMI_SU)
}
```

Quale si legge meglio secondo voi?

Lo stesso meccanismo sarà per la velocità di animazione del personaggio, lenta quando scende e veloce quando sale o normale se va in linea retta.

```
/*
  Velocità
*/
val VELOCITA_LENTA = 0.4
val VELOCITA_NORMALE = 0.2
val VELOCITA_VELOCE = 0.1
val ACCELERAZIONE_LENTA = 1
val ACCELERAZIONE_VELOCE = 4

def velocitàAnimazionPersonaggio(): Double = {
  direzionePersonaggio() match {
    case SINISTRA       => VELOCITA_NORMALE
    case DESTRA         => VELOCITA_NORMALE
    case ALTO           => VELOCITA_VELOCE
    case BASSO          => VELOCITA_LENTA
    case SINISTRA_ALTO  => VELOCITA_VELOCE
    case SINISTRA_BASSO => VELOCITA_LENTA
    case DESTRA_ALTO    => VELOCITA_VELOCE
    case DESTRA_BASSO   => VELOCITA_LENTA
    case _              => VELOCITA_NORMALE
  }
}
```

Proseguendo anche nella gestione del movimento:

```
def incrementoVelocitàPersonaggio(): Double = {
  if (isKeyPressed(Kc.VK_SHIFT))
    ACCELERAZIONE_VELOCE
  else
    ACCELERAZIONE_LENTA
}

def muoviPersonaggio(personaggio: Tartaruga) {
  val x = personaggio.posizione.x;
  val y = personaggio.posizione.y;
  val posizione = direzionePersonaggio() match {
    case SINISTRA       => (x - incrementoVelocitàPersonaggio, y)
    case DESTRA         => (x + incrementoVelocitàPersonaggio, y)
    case ALTO           => (x, y + incrementoVelocitàPersonaggio)
    case BASSO          => (x, y - incrementoVelocitàPersonaggio)
    case SINISTRA_ALTO  => (x - incrementoVelocitàPersonaggio, y + incrementoVelocitàPersonaggio)
    case SINISTRA_BASSO => (x - incrementoVelocitàPersonaggio, y - incrementoVelocitàPersonaggio)
    case DESTRA_ALTO    => (x + incrementoVelocitàPersonaggio, y + incrementoVelocitàPersonaggio)
    case DESTRA_BASSO   => (x + incrementoVelocitàPersonaggio, y - incrementoVelocitàPersonaggio)
    case _              => (x, y)
  }
  // facciamo spostare anche il rettangolo di collisione insieme al personaggio
  rettangoloDiCollisionePersonaggio.setPosition(posizione._1 - 50, posizione._2 - 71)
  personaggio.saltaVerso(posizione._1, posizione._2)
}
```

Qui volevo fare in modo che premendo il tasto `SHIFT` si potesse aumentare la velocità e quindi c'è una funzione `incrementoVelocitàPersonaggio` che mi serve per incrementare lo spostamento del personaggio.
Nella funzione `muoviPersonaggio` si noti come si sposti anche il rettangolo di collisione insieme al personaggio.
Un altro punto in cui si controlla se è premuto un tasto è la funzione per fermare l'animazione:

```
def fermaGiocoSeEsc() {
  //ferma il loop infinito di animazione
  if (isKeyPressed(Kc.VK_ESCAPE)) {
    stopAnimation()
  }
}
```

Il nostro personaggio però, dovrà svolazzare da qualche parte ed allora dovremmo caricare un qualche sfondo per simulare un ambiente. Lo facciamo allo stesso modo del personaggio usando una tartaruga. Come mai? Ci interessa che questo sfondo si sposti per simulare un attraversamento abbastanza lungo.
Per la verità si sarebbe potuto fare in un altro modo, ma forse questo è il più semplice e sfrutta quello che già conosciamo.
Quindi non doppiamo fare altro che creare una nuova tartaruga con `nuovaTartaruga` ed assegnargli un costume che in questo caso è una immagine di circa 3000 pixel.

```
/*
  Ambiente
*/

def caricaAmbiente(f: String): Tartaruga = {
  val immagine = loadImage(f)
  val ambiente = nuovaTartaruga(COORDINATE_AMBIENTE.x, 0)
  ambiente.indossaImmagine(immagine)
  println(immagine.getWidth(null))
  ambiente
}

val AMBIENTE = caricaAmbiente("./immagini/sfondo2.png")
```

Prima di scrivere il blocco principale della nostra animazione interattiva manca qualcosa, la gestione delle collisioni con i bordi dello *stage*.

```
def controllaCollisioniPersonaggio() {

  if (rettangoloDiCollisionePersonaggio.collidesWith(stageRight)) {
    //println("collide right")
    val x = AMBIENTE.posizione.x - COORDINATE_STAGE.width
    AMBIENTE.saltaVerso(x, AMBIENTE.posizione.y)
    PERSONAGGIO.saltaVerso(COORDINATE_STAGE.x + 100, PERSONAGGIO.posizione.y)
  }

  if (rettangoloDiCollisionePersonaggio.collidesWith(stageLeft)) {
    //println("collide left")
    val x1 = AMBIENTE.posizione.x + COORDINATE_STAGE.width
    val x2 = COORDINATE_STAGE.x + COORDINATE_STAGE.width - 100
    AMBIENTE.saltaVerso(x1, AMBIENTE.posizione.y)
    PERSONAGGIO.saltaVerso(x2 - 100, PERSONAGGIO.posizione.y)
  }
}
```

Qui si devono usare delle funzioni che sono scritte in inglese perché questa parte non è tradotta in italiano, ma il concetto è semplice: `collidesWith` ritornerà un valore `true` o `false` nel caso in cui il nostro `rettangoloDiCollisionePersonaggio` tocchi la parte destra (`stageRight`) o la parte sinistra (`stageLeft`) dello stage. In questo caso lo sfondo verrà spostato a destra o sinistra ed il personaggio riallineato.
In questa gestione *rudimentale* del contatto con i bordi dello stage mancano alcuni controlli perché il nostro personaggio dovrà rendersi conto di quando è in fondo od in cima rispetto allo sfondo. Se continuate ad andare in una direzione prima o poi lo sfondo verrà spostato oltre lo stage. Provate come esercizio a implementare un *riavvolgimento* dello sfondo in maniera da non farlo uscire dallo stage.

Siamo arrivati più o meno all'innesco della animazione:

```
val AMBIENTE = caricaAmbiente("./immagini/sfondo2.png")
val PERSONAGGIO = nuovoPersonaggio()

```

```
animate {
  fermaGiocoSeEsc()
  controllaCollisioniPersonaggio()
  adattaCostumiPersonaggio(PERSONAGGIO)
  muoviPersonaggio(PERSONAGGIO)
}
```

Il blocco `animate` è un loop infinito che eseguirà in continuazione la lista delle operazioni contenute. La prima, `fermaGiocoSeEsc`, controllerà se è stato premuto il tasto `ESC` ed in tal caso fermerà l'animazione; la seconda, `controllaCollisioniPersonaggio`, si assicurerà della posizione del personaggio tramite il suo rettangolo di collisione ed agirà di conseguenza; la terza, `adattaCostumiPersonaggio`, metterà al personaggio i gusti costumi in base alla direzione; l'ultima, `muoviPersonaggio`, farà scorrere il personaggio sullo stage.

Nell'archivio personaggio2.zip troverete il sorgente per Kojo e la cartella con le immagini.

4.14 Documenti segreti

Le bande, chi non ha mai fatto od organizzato una banda? Le riunioni nella capanna di vecchie tavole o sull'albero, le discussioni interminabili sul gioco del momento, gli archi e le fionde… I messaggi segreti.
I messaggi segreti, le tabelle di conversione delle lettere, per non fare capire alla banda rivale gli spostamenti delle truppe.
Ricordi di bambino quando i giochi elettronici non esistevano e ci si sbucciava le ginocchia cadendo dalla bicicletta.

Oggi i computer però li abbiamo e ci mandiamo le email ed i ragazzi alla fine non credo siano cambiati molto. I messaggini se li scambiano lo stesso, tra amici o i primi fidanzatini o fidanzatine. Tutte cose che i genitori devono intuire ma non sapere.

Oggi, impariamo come cifrare un messaggio.

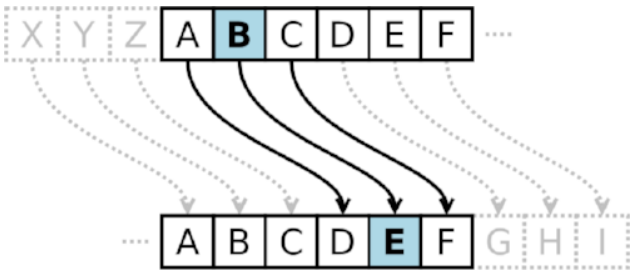
Cifrare i messaggi è una cosa che si fa da tempo ed uno dei primi sistemi lo inventò, pare, Gaio Giulio Cesare. Lo usava per mandare messaggi alle truppe che i nemici non potessero leggere. Il cosiddetto *cifrario di Cesare* era molto semplice e funzionava in questa maniera:

testo in chiaro	a	b	c	d	e	f	g	h	i	l	m	n	o	p	q	r	s	t	u	v	z
testo cifrato	D	E	F	G	H	I	L	M	N	O	P	Q	R	S	T	U	V	Z	A	B	C

Come potete notare ad ogni lettera ne corrisponde un'altra: `a -> D`. Cesare usava *spostare* la lettera di tre posti, per questo la lettera `a` corrisponde a `D` (qui messa in maiuscolo solo per evidenziarla):

- spostando la `a` di un posto diventerà `b`
- spostando la `a` di due posti diventerà `c`
- spostando la `a` di tre posti diventerà `d`

Come in questo schema:



Nel caso in cui si superasse la lunghezza dell'alfabeto si ricomincerà da capo e la `z` diventerà la `c`.
Ecco un esempio:

testo in chiaro	attaccaregliirriducibiligalliallaorasesta
testo cifrato	DZZDFFDUHLONNUUNGAFNENONLDOONDOODRUDVHVZD

L'esempio l'ho preso da wikipedia per prigrizia.

Come si può facilmente immaginare questo sistema è decisamente debole ed è facile decifrare il messaggio anche non conoscendo di quanto vengono spostate le lettere. Questo sistema che fu usato anche da altri, per esempio l'imperatore Augusto, godeva del fatto che per la maggior parte le persone non sarebbero nemmeno state in grado di leggere un testo *in chiaro* (non cifrato). Nell'XI secolo però un matematico arabo chiamato comunemente Alchindus (Al-Kindi) formulò le basi matematiche per decifrare messaggi di questo tipo e quindi il sistema non venne più utilizzato in assoluto.

Quello che noi oggi invece vogliamo utilizzare è il *cifrario di Vigenère*.
Questo sistema è più difficilmente decifrabile, anche se ha le sue basi in quello di Cesare. Fa uso di una password, una parola chiave, che è utilizzata per convertire un testo leggibile in uno non leggibile.

La base è semplice, invece di usare un numero per spostare le lettere si usano le lettere della password.

L'alfabeto è una lista di lettere e chiaramente ogni lettera ha una posizione in questa lista:

- a -> 1
- b -> 2
- c -> 3
- d -> 4

e così via…

Anche la nostra password è formata di lettere e quindi se abbiamo la parola *cane*: `c(3) a(1) n(12) e(5)`.
Con una password *paco*: = `p(14) a(1) c(3) o(13)`
Il testo cifrato sarà quindi *sbqt*:

- `3+14 = 17(s)`
- `1+1 = 2(b)`
- `12+3 = 15(q)`
- `5+13 = 18(t)`

In questo caso sia il testo che la password hanno una uguale lunghezza, in genere però la password è più costa del testo.
Per correggere questo problema la password viene ripetuta più volte fino a formare una stringa lunga quanto il testo e poi viene applicata.

Se la password fosse *pluto*:

```
ci vediamo stasera per giocare a quel gioco che a mamma e babbo non piace?
plutoplutoplutoplutoplutoplutoplutoplutoplutoplutoplutoplutoplutoplutoplut
```

La password così ripetuta viene chiamata *verme* perché si allunga per tutto il testo da cifrare. Ora non ci resta che applicarla come abbiamo fatto prima.
Per decifrare facciamo il passo inverso ed invece che sommare la posizione nell'alfabeto della lettera del testo in chiaro con la posizione nell'alfabeto della lettera della password, sottraiamo la posizione nell'alfabeto della lettera del testo cifrato con la posizione nell'alfabeto della lettera della password:

- `17-14 = 3(c)`
- `2-1 = 1(a)`
- `15-3 = 12(n)`
- `18-13 = 5(e)`

Più complicato parlarne che farlo. Ora facciamolo fare ad un computer.

Per poter implementare questo *algoritmo* abbiamo bisogno di fare almeno due funzioni: una per cifrare il testo ed una per decifrarlo. Oltre questo direi di fare anche una funzione che ricostruisce il *verme* partendo dalla password, dato che dovremmo farlo due volte: quando cifriamo e quando decifriamo.

Partiamo dal *verme*:

```
def calcolaVerme(password: String, lunghezzaTesto: Int): String = {
  val ripetizioni = lunghezzaTesto / password.length
  val resto = lunghezzaTesto % password.length
  val tappo = se(resto > 0) {
    password.slice(0, resto)
  } altrimenti {
    ""
  }
  (password * ripetizioni) + tappo
}
```

Per *allungare* la password e renderla lunga quanto il testo dobbiamo sapere la lunghezza del testo e quindi la inseriamo come parametro della funzione, oltre a questo naturalmente ci serve il testo della password.
Dobbiamo calcolare di quante volte allungare la password per arrivare al *verme* e la cosa da fare è dividere la lunghezza del testo per la lunghezza della password. Come sappiamo questa sarà una divisione intera perché le lunghezze in questo caso sono numeri interi, quindi ci servirà anche il resto di questa divisione.
Nel caso che il resto sia zero, cioè che se ripetiamo la password per le volte calcolate essa sarà lunga perfettamente uguale al testo, allora il nostro *tappo* diventerà una stringa vuota. In caso contrario ci servirà una parte della password per terminare il nostro verme.

Come sempre il tutto si potrebbe scrivere più brevemente conoscendo alcuni metodi dei *numeri* e delle *stringhe*:

```
def calcolaVerme(password: String, lunghezzaTesto: Int): String = {
  val ripetizioni = (lunghezzaTesto.toDouble / password.length).ceil.toInt
  val verme = password * ripetizioni
  verme.slice(0, lunghezzaTesto)
}
```

Per calcolare le ripetizioni a questo punto ci servirà una divisione con la virgola (come rivedremo anche dopo) e poi questo risultato lo innalzeremo al numero superiore con `ceil` riportandolo poi ad un `Int` (numero intero) che potrà essere usato come moltiplicatore per la stringa della password.
Ottenendo poi una stringa troppo lunga questa dovrà essere *affettata* (`slice`) al numero di caratteri del testo in chiaro da cifrare.

Come nell'esempio di sopra:

`ci vediamo stasera per giocare a quel gioco che a mamma e babbo non piace?`
`plutoplutoplutoplutoplutoplutoplutoplutoplutoplutoplutoplutoplutoplutoplut`

alla fine del *verme* non abbiamo *pluto* ma *plut*.

Una volta risolto come preparare il nostro *verme* passiamo alle funzioni per cifrare e decifrare il testo.

Prima vediamo come cifrare i nostri messaggi segreti, vedrete che il contrario sarà molto semplice e praticamente uguale.

```
def cifra(testo: String, password: String): String = {
  val verme = calcolaVerme(password, testo.length)
  var testoCifrato = ""
  ripetizione(testo.length) { indice =>
    val posizione = indice - 1
    val numero = testo(posizione).toInt
    val numeroVerme = verme(posizione).toInt
    val numeroLetteraCifrata = numero + numeroVerme
    val letteraCifrata = numeroLetteraCifrata.toChar
    testoCifrato = testoCifrato + letteraCifrata
  }
  testoCifrato
}
```

La funzione prenderà come parametri il testo da cifrare ed una password e restituirà il testo cifrato. Inizialmente calcoleremo il *verme* da applicare e poi ripeteremo un blocco di codice tante volte quanti sono i caratteri del testo da convertire. L'espressione `testo.length` fornisce il numero dei caratteri, ovvero la lunghezza della stringa, un numero che ci servirà anche per avere la posizione del carattere nel testo.
La posizione ha un valore di `indice - 1` perché l'istruzione `ripetizione` parte da 1 mentre le stringhe sono *indicizzate* a partire da zero, come già detto in precedenza.
Le stringhe sono delle liste di caratteri messi uno dietro l'altro e che hanno un valore di posizione all'interno della

lista stessa. Per accedere ad un elemento della lista usiamo le parentesi tonde: `testo(10)` ci darà il carattere nella posizione 10.
Come visto prima, abbiamo bisogno della posizione della lettera (carattere) all'interno dell'alfabeto e per fare questo usiamo un metodo (un comando) dell'oggetto *carattere (Char)* che è *toInt*. L'espressione `val numero = testo(posizione).toInt` imposta *numero* al numero corrispondente a quel carattere e per esempio:

```
scala> 'a'.toInt
res25: Int = 97
```

invocando `toInt` sul carattere a otteniamo il numero 97 che corrisponde alla posizione della lettera nell'*alfabeto* che usano i computer (molto più lungo di quello che usiamo noi per scrivere).
Lo stesso facciamo per la lettera del nostro *verme* che ormai è lungo come il testo che vogliamo cifrare: `val numeroVerme = verme(posizione).toInt`.
Una volta ottenuti questi numeri li sommiamo con `val numeroLetteraCifrata = numero + numeroVerme` e ritrasformiamo `numeroLetteraCifrata` in un carattere: `val letteraCifrata = numeroLetteraCifrata.toChar`. Il carattere `letteraCifrata` è quello che ci serve e quindi possiamo attaccarlo alla nostra stringa di caratteri che usiamo come *accumulatore*: `testoCifrato`. Fatto questo per tutta la lunghezza della stringa non ci resta che restituirlo.

Abbiamo un testo cifrato con una password:

testo	Caro amico ci vogliamo incontrare stasera per giocare?
password	pippo
testo cifrato	³Êâß⊠ÑÖÙÓÞ⊠ÌÙ⊠åßÐÜÙÐÝØ⊠ÙÝÓØÞääáÑÛÕ⊠âäÊãÕáÑ⊠àÕá⊠ÐÙßÒÑÛÕ¯

Il testo cifrato potreste mandarlo ad un amico che conoscendo poi la password potrebbe così decifrarlo.

Per recuperare il testo originale si dovrà fare una operazione inversa: prima abbiamo sommato ora dovremo sottrarre:

```
def decifra(testoCifrato: String, password: String): String = {
  val verme = calcolaVerme(password, testo.length)
  var testoDecifrato = ""
  ripetizione(testo.length) { indice =>
    val posizione = indice - 1
    val numero = testoCifrato(posizione).toInt
    val numeroVerme = verme(posizione).toInt
    val numeroletteraDecifrata = numero - numeroVerme
    val letteraDecifrata = numeroletteraDecifrata.toChar
    testoDecifrato = testoDecifrato + letteraDecifrata
  }
  testoDecifrato
}
```

Se guardate bene questa funzione non è molto diversa da quella presentata prima (a parte i nomi dei valori e delle variabili che sono stati messi diversi per far capire meglio), l'unica differenza reale è: `val numeroletteraDecifrata = numero - numeroVerme`. Prima sommavamo ed adesso sottraiamo il valore della lettera corrispondente del verme.
É tutto qui.

```
val testo = "Caro amico ci vogliamo incontrare stasera per giocare?"
val password = "pippo"

val testoCifrato = cifra(testo, password)
val testoDecifrato = decifra(testoCifrato, password)

println("testo: " + testo)
println("password: " + password)
println("testo cifrato: " + testoCifrato)
println("testo decifrato: " + testoDecifrato)
```

testo	Caro amico ci vogliamo incontrare stasera per giocare?
password	pippo
testo cifrato	³Êâß⊠ÑÖÙÓÞ⊠ÌÙ⊠åßÐÜÙÐÝØ⊠ÙÝÓØÞääáÑÛÕ⊠âäÊãÕáÑ⊠àÕá⊠ÐÙßÒÑÛÕ¯
testo decifrato	Caro amico ci vogliamo incontrare stasera per giocare?

Nel nostro caso non abbiamo bisogno di operare delle rotazioni dell'alfabeto (se arriviamo in fondo ricominciamo dall'inizio) perché il numero dei caratteri disponibili con Scala è decisamente grosso e sarà improbabile superarlo con i nostri messaggini di testo. Certo è che volendo rendere questo codice capace di cifrare non solo i nostri messaggi ma anche immagini, musica ed altri dati andrebbe modificato.

Come sempre il codice è *didattico* e ci sono altri modi (migliori) per scriverlo e renderlo più compatto, a voi provarci.
Come esempio vediamo come il tutto si potrebbe compattare molto e restringere ad una sola funzione che sarà in grado di cifrare il testo e decifrarlo. Il codice qui è troppo *complesso* per i ragazzi ed è presentato solo per *stimolarne la curiosità* e mostrare come, una volta arrivati alla comprensione del linguaggio di programmazione, si possano affrontare gli stessi problemi ma con soluzioni diverse.

```
1  //Cifrario a scostamento Vigenere/Vernam
2
3  val testo = "Caro amico ci vogliamo incontrare stasera per giocare?"
4  val password = "pippo"
5
6  // una sola funzione che cifra e decifra
7  def cifra(testo: String, password: String): String = {
8    val verme = password * (testo.length.toFloat / password.length).ceil.toInt
9    testo.zip(verme).foldLeft("")((a, b) =>
10     a + (b._1 ^ b._2).toChar.toString)
11 }
12
13 val cifrato = cifra(testo, password)
14 val decifrato = cifra(cifrato, password)
15
16 scriviLinea("testo: " + testo)
17 scriviLinea("password: " + password)
18 scriviLinea("testo cifrato: " + cifrato)
19 scriviLinea("testo decifrato: " + decifrato)
```

Il risultato sarà più o meno questo (non tutti i caratteri del testo cifrato sono stampabili essendo delle operazioni binarie, per questo nel documento se ne vedono pochi):

testo	Caro amico ci vogliamo incontrare stasera per giocare?
password	pippo
testo cifrato	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
testo decifrato	Caro amico ci vogliamo incontrare stasera per giocare?

Come Emacs (l'editor che utilizzo per scrivere questo documento) fa vedere la tabella:

```
|testo         | Caro amico ci vogliamo incontrare stasera per giocare?                 |
|password      | pippo                                                                  |
|testo cifrato | 3^H^B^_O^Q^D^Y^S^Y^P^Y^_^N^\^Y^N^]^FP^Y^A^S^F^^^D^]^Q^[^UP^\^D^H^C^U^]^QI |
|testo decifrato| Caro amico ci vogliamo incontrare stasera per giocare?                 |
```

In questa funzione non ci si preoccupa della lunghezza del verme che può essere più lungo del testo a causa del funzionamento del metodo `zip`. Per questo si fa in modo che sia una divisione in virgola e poi si arrotonda il risultato all'intero superiore.
Con il metodo `zip` si unisce il *testo* al *verme* (il metodo `zip` tronca la stringa più lunga perché accoppia lettera per lettera finché non si esaurisce una stringa ignorandone il resto.) ottenendo una *lista di coppie di caratteri*: uno del testo ed uno del verme.
Con `foldLeft` si applica una operazione (*funzione anonima*) per ogni coppia di caratteri; i valori della coppia sono messi in relazione di `^` (*XOR*) che è un operatore che lavora sui bit dei dati. Il risultato è poi concatenato in una stringa.
L'operazione di *XOR* è interessante perché *reversibile*:

```
1  scala> val x = 'a' ^ 'b'
2  x: Int = 3
3
4  scala> x.toChar
```

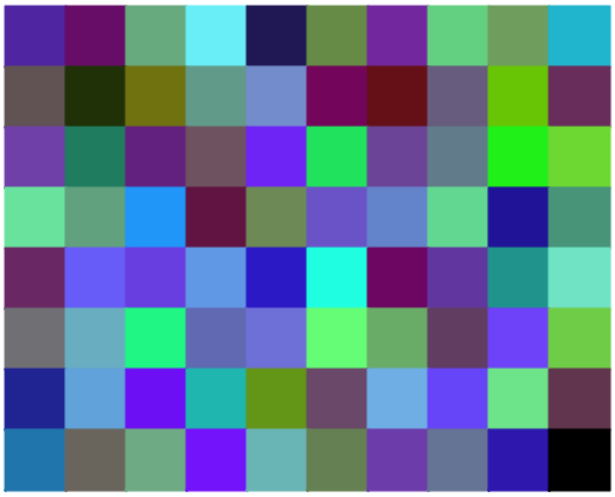
```
res2: Char = ?

scala> (x ^ 'b').toChar
res3: Char = a

scala>
```

Facendo una operazione ^ tra la lettera a e la lettera b otteniamo un intero 3, ma se poi mettiamo mettiamo in *XOR* 3 con b otteniamo il numero 97 che corrisponde ad a. Per questo motivo la funzione `cifra` sopra si può applicare in entrambe le direzioni. L'operazione sarà sempre la stessa e il risultato dipenderà dai dati che gli forniremo. In questo modo si potrebbe per aumentare la sicurezza cifrare più volte (magari con password diverse) e per decifrare rifare i passi inversi.
Sappiate però che il sistema non è comunque particolarmente robusto a meno di non utilizzare una password che sia lunga quanto il testo da cifrare: *one time pad*.
Insomma, in genere più lunga è la password e meglio è.

Buona cifratura a tutti.

4.15 Parole, quadrati e colori

Cosa significa questo?



Qui c'è nascosta una frase.

Ci sono modi per nascondere dei dati all'interno di altri e questa tecnica è chiamata *steganografia* che vuol dire, dal greco, *scrittura nascosta* o *coperta*.
Senza affrontare tecniche molto complesse vediamo come l'immagine sopra è stata generata e come poi poterla rileggere.

Il codice utilizza alcuni blocchi di codice forse troppo complessi per i ragazzi, come alcune operazioni sui bit, quindi prendeteli per come sono anche se cercherò di spiegarle al meglio possibile.

Per iniziare abbiamo bisogno della solita funzione per disegnare dei poligoni regolari, in questo caso dei quadrati, ma la riproponiamo come già vista:

```
/*
 Funzione per disegnare poligoni regolari
 */
def poligono(lati: Int, lato: Int, colore1: Color, colore2: Color) {
  val angolo = 360 / lati.toDouble
  colorePenna(colore1)
```

```
    coloreRiempimento(colore2)
    seVero(lati % 2 >= 0) {
      sinistra(90 - angolo);
    }
    ripeti(lati) {
      avanti(lato)
      destra(angolo)
    }
  }
```

Alla solita funzione vista in precedenza si sono aggiunti due parametri per specificare il colore del bordo ed il riempimento, che in questo caso ci interessano visto che si dovrà far disegnare alla nostra *tartaruga* dei quadrati pieni e colorati.

Volendo convertire del testo in una serie di quadrati colorati, nascondendo in ognuno di essi una lettera, dobbiamo valutare come poterlo fare.
Le lettere sono gestite nei computer come dei codici numerici quindi direi di usare quelli. Le lettere italiane risiedono quasi tutte nei numeri che vanno da 0 a 255 e questo è interessante nel nostro gioco di colori.
Cosa è un colore nel nostro linguaggio di programmazione? Un colore è una istanza dell'oggetto `Color`. Senza entrare troppo nei particolari di alcune cose non ancora affrontate, un colore è la somma dei tre colori fondamentali: rosso, verde, blu. I computer per far vedere i colori a video usano la cosiddetta *sintesi additiva*: la somma del rosso con il verde ed il blu da il bianco. Variando le dosi dei tre colori si avranno tutti gli altri. Ogni colore fondamentale può avere un valore che va da 0 a 255.

- `Color(255, 0, 0, 255)` è il rosso puro
- `Color(0, 255, 0, 255)` è il verde puro
- `Color(0, 0, 255, 255)` è il blu puro
- `Color(255, 255, 255, 255)` è il bianco
- `Color(0, 0, 0, 255)` è il nero

L'ultimo valore è l'*opacità* del colore, quanto cioè sia trasparente: 0 per completamente trasparente e 255 per completamente opaco.

Per creare un colore nuovo dovremo scrivere in questo modo:

```
val rosso = new Color(255, 0, 0, 255)
```

La parola chiave `new` serve per costruire un nuovo oggetto, questa istruzione fa parte dell'aspetto di *orientamento agli oggetti* di Scala. Prendete questa cosa per buona e ricordate come si costruisce un nuovo colore per adesso.
Sapere come fare dei nuovi colori ci servirà per riempire i nostri quadrati messi uno dietro l'altro.

```
/*
 Funzione per generare un quadrato con il colore di riempimento in base alla lettera. Il codice della lettera
 è immagazzinato nel canale rosso del colore, i rimanenti verde e blu sono casuali.
 */
def letteraInQuadrato(lettera: Char, lato: Int = 50, marca: Boolean = false) {
  val coloreVerde = numeroCasuale(255)
  val coloreBlu = numeroCasuale(255)
  val coloreRosso = lettera.toInt
  val colore = new Color(coloreRosso, coloreVerde, coloreBlu, 255)
  poligono(4, lato, colore, colore)
  if (marca)
    scriviAllaPosizione(lettera.toString, posizione.x + (lato / 2), posizione.y + (lato / 2), nero)
}
```

Eccola qua, la nostra funzione `letteraInQuadrato`, a cui si deve fornire una lettera e la lunghezza del lato. L'ultimo parametro serve solo per far scrivere la lettera stessa all'interno del quadrato per controllare che le cose vadano bene durante lo sviluppo (usando una funzione che è stata scritta per quello scopo e che verrà riutilizzata in altre parti).

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com

```
def scriviAllaPosizione(testo: String, x: Double, y: Double, colore: Color) {
  val t = nuovaTartaruga(x, y)
  t.invisibile()
  t.colorePenna(colore)
  t.scrivi(testo)
}
```

Una cosa che si nota e che non dovreste ancora avere visto sono i parametri di *default* delle funzioni. In questo caso `lato` se non fornito alla chiamata avrà valore 50 e `marca` sarà *falso*. In questo modo si potrà invocare la funzione soltanto come `letteraInQuadrato('a')` e gli altri parametri omessi avranno i loro valori di base.

Ho deciso di immagazzinare il valore della lettera nel *canale rosso* del colore ed assegnare dei numeri a caso tra 0 e 255 agli altri utilizzando la funzione di Kojo `numeroCasuale`. Generiamo il nuovo colore e disegnamo il quadrato.
Abbiamo il primo *mattoncino* per fare quello che vogliamo.

Con questo possiamo disegnare un quadrato partendo da una lettera con un certo colore che è formato dal codice numerico del carattere e da altri valori casuali. Noi però vogliamo *scrivere* delle parole quindi si dovrà chiamare `letteraInQuadrato` per ogni carattere che forma una parola. Ci serve un'altra funzione.

```
/*
 La funzione codifica un testo in una sequenza di quadrati colorati.
 */
def codifica(testo: String, inizioX: Int, inizioY: Int, colonne: Int = 10, larghezzaColonna: Int = 50) {
  saltaVerso(inizioX, inizioY)
  var colonna = 1
  var riga = 0
  ripetiPerOgniElementoDi(testo) { lettera =>
    letteraInQuadrato(lettera, larghezzaColonna)
    colonna += 1
    if (colonna > colonne) {
      colonna = 1
      riga += 1
      saltaVerso(inizioX, inizioY - larghezzaColonna * riga)
    }
    else {
      saltaVerso(posizione.x + larghezzaColonna, posizione.y)
    }
  }
  poligono(4, larghezzaColonna, nero, nero)
}
```

Si dovrà fornire a `codifica` il testo da scrivere ma anche la posizione iniziale x ed y da dove cominciare a disegnare. Gli altri, che sono parametri con valori di base, dicono di quante colonne è composta l'immagine ovvero la lunghezza della riga e la larghezza della colonna (che sarebbe il lato del quadrato). Questi valori ci servono per poter scrivere una riga dopo l'altra e sapere dove interromperle per passare alla prossima.
Nel codice inizialmente spostiamo la tartaruga alle coordinate di partenza e poi imposteremo due variabili: `colonna` e `riga` che serviranno per tenere traccia della posizione (del quadrato) che stiamo disegnando. Nel *loop* `ripetiPerOgniElementoDi(testo)` ci sposteremo di una certa quantità che stata calcolata in base alla larghezza della colonna e controlleremo di non essere arrivati al numero massimo di colonne, in caso contrario ci sposteremo alla riga sotto. Ogni volta, in questo ciclo, invocheremo `letteraInQuadrato` per disegnare la nostra *lettera colorata*.

Una nota nel codice, ho usato un operatore nuovo += che significa: somma alla variabile se stessa più il valore a destra: `colonna +=1` è lo stesso che scrivere `colonna = colonna + 1`.

Abbiamo ora, più o meno tutto per farlo:

```
pulisci()
switchToDefault2Perspective()
ritardo(10)
invisibile()

val X = -300
val Y = 150

```

```
val testo = "Oggi esco a passeggio con la mia amica Gigia, ma poi andiamo al cinema insieme."
codifica(testo, X, Y)
```

Ecco l'immagine che abbiamo visto all'inizio.

L'interessante però è poter recuperare il testo, altrimenti abbiamo fatto solo dell'arte moderna. Recuperare il testo è un po' più complicato e per farlo dobbiamo accedere all'immagine che abbiamo fatto ed analizzare il colore dei *pixel* che la compongono. Questo codice se (notevolmente) migliorato potrebbe essere usato per caricare una immagine dal disco ed analizzarla.

```
/*
 La funzione rilegge l'immagine generata estraendo il testo.
 La larghezza della colonna deve essere variata per adattarla all'immagine, la password insomma.
 */
def rileggi(startX: Int, startY: Int, larghezzaColonna: Int = 50): String = {
  val centroDellaColonna = larghezzaColonna - 2
  val larghezzaImmagine = canvasBounds.getWidth.toInt
  var x = startX
  var y = startY
  var colore = bianco
  var testo = ""
  var colonna = 1
  var riga = 1
  ripetiFinché(colore != nero) {
    colore = prendiColoreAlPixel(x, y)
    if (colore != bianco) {
      testo += colore.getRed.toChar
    }
    if (x >= larghezzaImmagine - centroDellaColonna) {
      colonna = 1
      riga += 1
      x = startX
      y += larghezzaColonna
    }
    else {
      colonna += 1
      x += larghezzaColonna
    }
  }
  testo
}
```

Non è molto diversa dalla funzione `codifica` (e detto tra noi è pure *bruttina* ed andrebbe fatta meglio), ha delle coordinate da dove partire per analizzare l'immagine e le solite righe e colonne. Siccome stiamo analizzando una immagine utilizzeremo la larghezza dell'immagine stessa e non il numero delle colonne come fatto in precedenza. Per leggere il colore nel pixel raggiunto abbiamo una funzione che è un po' complicata quindi prendetela come è:

```
/*
 La funzione legge il colore di un pixel alle coordinate date.
 */
def prendiColoreAlPixel(x: Int, y: Int): Color = {
  val image = stage.canvas.getCamera.toImage(canvasBounds.getWidth.toInt, canvasBounds.getHeight.toInt, stage.myCanvas.
  val rgb = image.asInstanceOf[java.awt.image.BufferedImage].getRGB(x, y)
  val valoreRosso = (rgb & 0x00ff0000) >> 16
  val valoreVerde = (rgb & 0x0000ff00) >> 8
  val valoreBlu = rgb & 0x000000ff
  new Color(valoreRosso, valoreVerde, valoreBlu)
}
```

Per essere semplici, si recupera l'immagine relativa all'area di disegno che risiede nella memoria del computer, poi se ne legge a certe coordinate il valore RGB (Red/Green/Blue) che è un numero intero lungo 24 bit. Con certe operazioni si divide in tre parti ed ognuna di queste sarà il valore del rosso, verde e blu. Prendetela come una delle tante funzioni interne di Kojo ed usatela come è.

In questo programma sono state prese delle decisioni: il bianco significa che non ci sono lettere ed il nero è il quadrato terminale (un po' come il punto in una frase).

Prima di iniziare a *decifrare* l'immagine dobbiamo però trovare da dove partire: le coordinate del primo quadrato colorato.

Noi non sappiamo precisamente dove possa trovarsi il primo quadrato e per questo motivo ci serve una funzione che lo trovi per noi.

```
/*
 La funzione cerca di valutare l'inizio del testo cifrato in colori nel canvas
 */
def trovaInizio(): Point = {
  val larghezzaImmagine = canvasBounds.getWidth.toInt
  val saltoX = 10
  val saltoY = 20
  var x = 0
  var y = 0
  var colore = bianco
  ripetiFinché(colore == bianco) {
    colore = prendiColoreAlPixel(x, y)
    if (x >= larghezzaImmagine - saltoX) {
      y += saltoY
      x = 0
    }
    else x += saltoX
  }
  Point(x, y)
}
```

Questa soluzione è come si dice in gergo: *quick and dirty*, veloce e sporca. Fa una scansione dell'immagine saltando di un certo numero di pixel (per renderla più veloce) partendo dall'inizio. È decisamente *sporca* come implementazione ma in questo caso funziona. La probabilità che trovi il primo quadrato è abbastanza alta ed a noi adesso va bene così. Una volta che ha trovato un pixel che non sia bianco restituisce le coordinate che poi passeremo a `rileggi`.

```
val inizio = trovaInizio()
val testoRiletto = rileggi(inizio.x.toInt, inizio.y.toInt)
```

Caricate il file *lettere_e_colori.kojo* e provate.

La soluzione però non è molto veloce ed abbiamo un *collo di bottiglia* quando accediamo all'immagine dell'area di disegno immagazzinata in memoria.
L'errore? la carichiamo ogni volta che cerchiamo un colore.

In questa funzione:

```
/*
 La funzione legge il colore di un pixel alle coordinate date.
 */
def prendiColoreAlPixel(x: Int, y: Int): Color = {
  val image = stage.canvas.getCamera.toImage(canvasBounds.getWidth.toInt, canvasBounds.getHeight.toInt, stage.myCanvas.
  val rgb = image.asInstanceOf[java.awt.image.BufferedImage].getRGB(x, y)
  val valoreRosso = (rgb & 0x00ff0000) >> 16
  val valoreVerde = (rgb & 0x0000ff00) >> 8
  val valoreBlu = rgb & 0x000000ff
  new Color(valoreRosso, valoreVerde, valoreBlu)
}
```

Impostiamo ogni volta il valore `image` chiamando:

- `stage.canvas.getCamera.toImage(canvasBounds.getWidth.toInt, canvasBounds.getHeight.toInt, stage.myCanvas.getBackground)`

e questo non è ottimale, perché porta via tempo di calcolo al nostro computer.

Dobbiamo trovare prima l'immagine e poi passarla come valore di parametro alle funzioni che la usano. Per questo motivo aggiungeremo una funzione e ne modificheremo tre.

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com

```
/*
 La funzione accede all'immagine in memoria che corrisponde al canvas.
 */

def accediAdImmagineInMemoria() : BufferedImage = {
  val image = stage.canvas.getCamera.toImage(canvasBounds.getWidth.toInt, canvasBounds.getHeight.toInt, stage.myCanvas.
  image.asInstanceOf[BufferedImage]
}
```

Questa prima modifica ci permetterà di immagazzinare l'immagine in un valore che poi forniremo alle funzioni `prendiColoreAlPixel`, `trovaInizio` e `rileggi`.

```
/*
 La funzione legge il colore di un pixel alle coordinate date.
 */
def prendiColoreAlPixel(image : BufferedImage, x: Int, y: Int): Color = {
  val rgb = image.getRGB(x, y)
  val valoreRosso = (rgb & 0x00ff0000) >> 16
  val valoreVerde = (rgb & 0x0000ff00) >> 8
  val valoreBlu = rgb & 0x000000ff
  new Color(valoreRosso, valoreVerde, valoreBlu)
}

/*
 La funzione cerca di valutare l'inizio del testo cifrato in colori nel canvas
 */
def trovaInizio(immagine: BufferedImage): Point = {
  val larghezzaImmagine = canvasBounds.getWidth.toInt
  val saltoX = 10
  val saltoY = 20
  var x = 0
  var y = 0
  var colore = bianco
  ripetiFinché(colore == bianco) {
    colore = prendiColoreAlPixel(immagine, x, y)
    if (x >= larghezzaImmagine - saltoX) {
      y += saltoY
      x = 0
    }
    else x += saltoX
  }
  Point(x, y)
}

/*
 La funzione rilegge l'immagine generata estraendo il testo.
 La larghezza della colonna deve essere variata per adattarla all'immagine, la password insomma.
 */
def rileggi(immagine: BufferedImage, startX: Int, startY: Int, larghezzaColonna: Int = 50): String = {
  val centroDellaColonna = larghezzaColonna - 2
  val larghezzaImmagine = canvasBounds.getWidth.toInt
  var x = startX
  var y = startY
  var colore = bianco
  var testo = ""
  var colonna = 1
  var riga = 1
  ripetiFinché(colore != nero) {
    colore = prendiColoreAlPixel(immagine, x, y)
    if (colore != bianco) {
      testo += colore.getRed.toChar
    }
    if (x >= larghezzaImmagine - centroDellaColonna) {
      colonna = 1
      riga += 1
      x = startX
      y += larghezzaColonna
    }
```

```
57      else {
58        colonna += 1
59        x += larghezzaColonna
60      }
61    }
62    testo
63  }
```

Adesso l'immagine non dovrà essere calcolata per molte volte ma soltanto una con un notevole risparmio di tempo.

```
1  val immagine = accediAdImmagineInMemoria()
2  val inizio = trovaInizio(immagine)
3  val testoRiletto = rileggi(inizio.x.toInt, inizio.y.toInt)
```

Guardando bene… Ci si può accorgere come facendo una funzione che carica una immagine dal disco e la inserisce nel valore `immagine` potremmo scrivere delle immagini codificandoci del testo, inviarle ad un amico o amica che potrebbe poi rileggerle.

```
1  /*
2   La funzione carica un file dal disco.
3   */
4  def caricaImmagine(fileName : String) : BufferedImage = {
5    ImageIO.read(new File(fileName));
6  }
```

Nel file *lettere_e_colori4.kojo* invece si è modificata la funzione che raccoglie il colore dell'area di disegno, entrando nell'area stessa e senza fare prima uno *screenshot* in una immagine in memoria:

```
1  /*
2   La funzione prende il colore di un pixel alle coordinate date.
3   Leggendolo dall'area di disegno direttamente
4   */
5  def prendiColoreAlPixel(x: Int, y: Int): Color = {
6    val cameraBounds = stage.canvas.getCamera.getBounds
7    val centerX = cameraBounds.getCenterX.toInt
8    val centerY = cameraBounds.getCenterY.toInt
9    val canvasX = (centerX.toInt + x)
10   val canvasY = (centerY.toInt - y)
11   val pickpath = stage.canvas.getCamera.pick(canvasX, canvasY, 1)
12   val paint = pickpath.getPickedNode.getPaint
13   if (paint != null)
14     paint.asInstanceOf[Color]
15   else
16     Color(255, 255, 255, 255)
17 }
```

Si procede trovando i limiti dell'area di disegno e le coordinate del punto centrale, perché non sono uguali a quelle della tartaruga. Le coordinate dell'area di disegno partono dall'angolo in alto a sinistra (coordinate $x = 0$ ed $y = 0$) mentre, come sappiamo, quelle della tartaruga dal centro. Le altre funzioni che abbiamo utilizzato per rileggere dovranno essere poi modificate per adattarle, ma l'idea nel suo complesso rimarrà la stessa.

```
1  def rileggi(startX: Int, startY: Int, larghezzaColonna: Int = 50): String = {
2    val centroDellaColonna = larghezzaColonna / 2
3    var x = startX + centroDellaColonna
4    var y = startY + centroDellaColonna
5    var colore = bianco
6    var testo = ""
7    ripetiFinché(colore != nero) {
8      colore = prendiColoreAlPixel(x, y)
9      if (colore != bianco) {
10       testo += colore.getRed.toChar
11       x += larghezzaColonna
12     }
13     else {
```

```
14          x = startX + centroDellaColonna
15          y -= larghezzaColonna
16        }
17      }
18      testo
19  }
```

Smessaggiatevi, o meglio *smessaggimmaginatevi*

4.16 La distanza tra le parole

Che vuol dire *distanza tra le parole*? Si misura in metri? In centimetri o millimetri? No, miei cari piccoli lettori si misura in:

- eliminazione di lettere
- inserimento di lettere
- sostituzione di lettere

Adesso i lettori più arguti si staranno chiedendo già a cosa mai possa servire e l'esempio è abbastanza semplice. Questa *distanza* probabilmente la usate ogni giorno quando andate a fare le ricerche con il computer su un *motore di ricerca* online. Mi astengo dal citare il più conosciuto e vi dico soltanto che non è il solo ma ce ne sno altri altrettanto validi e spesso meno *curiosi* di cosa state facendo. Comunque ecco qua, voi cercate una parola e lui ve ne suggerisce di simili (per la verità usa diverse tecniche per fare questo, compreso il ricordarsi le ricerche che avete fatto in precedenza e varie altre cose discutibili).
Voi cercate *pippo* e vi suggerisce anche *pappa*.

Quale sarà la distanza che separa *pippo* da *pappa*, ovvero il *minimo numero di cambiamenti*? La distanza è 2 e vediamo come mai:

- pippo -> p[a]ppo (1)
- pappo -> papp[a] (2)

Per trasformare *pippo* in *pappa* abbiamo cambiato due lettere: due sostituzioni.

- sale -> sole (1)

Possiamo anche aggiungere delle lettere:

- caso -> caso[lare] (4)

Aggiungere e sostituire:

- completo -> comple[s]to (1)
- complesto -> comples[s]o (2)

Sostituire ed elminare:

- casetta -> cas[i]tta (1)
- casitta -> casita (2) eliminazione di una *t*
- casita -> casi[n]a (2)

Questo problema fu *codificato* da un certo **Vladimir Levenshtein** ed infatti va sotto il nome di *distanza di Levenhstein*. Questa è la formula:

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{se } min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{altrimenti} \end{cases} \tag{1}$$

Non vi spaventate subito, vedrete che alla fine è *abbastanza* semplice. Secondo me ha anche una particolarità: è direttamente esprimibile in un codice di programmazione. Non sempre è così ed a volte occorre fare dei *giri* poco intuitivi ma questa non lo è ed in Scala si può implementare più o meno *direttamente*.
Ci sono anche modi più efficienti di farlo perché anche se non sembra, come dice chi programma, è *computazionalmente* onerosa. Insomma per trovare questa fatidica distanza anche su parole corte ci si metterà molto tempo perché saremo costretti a ricalcolare tante volte la stessa cosa.

Nella *formula* troviamo una indicazione:

$$1_{(a_i \neq b_j)} \tag{2}$$

che sta a significare che ad un certo punto, mentre negli altri si aggiunge sempre `1`, lì invece si dovrà aggiungere `0` nel caso in cui le lettere che si controllano in quel momento siano uguali e `1` se sono diverse. Questa parte è interessante ed è uno *stratagemma matematico* che serve ad indicare se quella lettera fa parte dell'insieme delle lettere uguali oppure no. Se ci fate caso è lo stesso motivo perché si aggiunge sempre `1` negli altri casi, ma vediamo.
Il primo blocco:

$$lev_{a,b}(i-1,j)+1 \tag{3}$$

trova quante eliminazioni di lettere servono e se le lettere sono da eliminare devono essere per forza *diverse* (e quindi `1` ci fa sapere che quelle lettere appartengono all'insieme delle lettere diverse.
Il secondo:

$$lev_{a,b}(i,j-1)+1 \tag{4}$$

sarebbe il numero delle operazioni di inserimento che occorrono (stesso discorso di sopra se le lettere vanno inserite saranno nell'insieme delle lettere diverse).
Il terzo:

$$lev_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \tag{5}$$

è la sostituzione e quindi qui le lettere potrebbero essere uguali o diverse.
Insomma con questo stratagemma il nostro *algoritmo* saprà come comportarsi se `a = b`? E se `a = a`?.

Questo algoritmo poi è *ricorsivo*, un concetto che non abbiamo ancora visto. La maggior parte dei linguaggi di programmazione supporta la *ricorsività* ovvero la possibilità di invocare una *funzione* all'interno di sé stessa. La *ricorsività* è un *concetto complesso* ma di uso semplice; può portare a varie problematiche che non staremo qui ad indicare e quindi va usata con *criterio*.
Se implementassimo l'algoritmo in maniera non ricorsiva sarebbe molto più lungo ed anche difficile da capire, quindi seguiremo direttamente la formula.
In questa noi sappiamo che `a` indica la prima delle parole da controllare e `b` la seconda; `i` e `j` sono rispettivamente le lunghezze in numero di lettere delle due parole. Questo algoritmo quindi, lavora sulle posizioni delle lettere nelle parole.

Noi cerchiamo il *minimo numero di cambiamenti* ed essendo tre operazioni, ognuna delle quali ci dice quanti cambiamenti di quel tipo fare, dobbiamo calcolare il *minimo*.
Scala ha una libreria matematica che ci fornisce la funzione giusta: `min`; che è presente nel *package*: `scala.math`.

```
1 | import scala.math.min
```

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com

Adesso la funzione `min` è accessibile.
Noi però dobbiamo trovare il minimo tra tre numeri e non due come fa `min`, dobbiamo scrivere una funzione che lo faccia e che prima calcoli il minimo di due numeri e poi tra il risultato ottenuto ed il numero rimanente:

```
def minimum(a: Int, b: Int, c: Int): Int = {
  val m = min(a, b)
  min(m, c)
}
```

Scrivendolo in maniera più compatta:

```
def minimum(a: Int, b: Int, c: Int): Int = min(min(a, b), c)
```

Dobbiamo anche definire la funzione che ci restituisce il cosiddetto *costo* ovvero quello che indica se le lettere sono uguali o no:

```
def cost(i: Int, j: Int): Int =
  if (source(i - 1) == target(j - 1)) 0 else 1
```

Una volta definito questo, ecco la funzione vera e propria:

```
def lev(i: Int, j: Int): Int = {
  (i, j) match {
    case (i, 0) => i
    case (0, j) => j
    case (i, j) => minimum(
      lev(i - 1, j) + 1,
      lev(i, j - 1) + 1,
      lev(i - 1, j - 1) + cost(i, j)
    )
  }
}
```

Se notate bene è pressoché identica alla formula che Levenhstein scrisse.
Per i vari controlli si è usato il costrutto `match` che meglio si adatta ad operazioni di questo tipo ma si sarebbe potuto anche usare degli `if`:

```
def lev_if(i: Int, j: Int): Int = {
  if (i > 0 && j == 0)
    i
  else if (i == 0 && j > 0)
    j
  else if (i > 0 && j > 0) {
    minimum(
      lev_if(i - 1, j) + 1,
      lev_if(i, j - 1) + 1,
      lev_if(i - 1, j - 1) + cost(i, j)
    )
  } else 0
}
```

A me sembra più chiara la prima versione.
Misuriamola:

```
def time[R](block: => R): R = {
  val t0 = System.nanoTime()
  val result = block
  val t1 = System.nanoTime()
  println("Elapsed time: " + (t1 - t0) / 1000 + "ms")
  result
}
```

Il codice è un po' complicato e non staremo qui ad affrontarlo: è una funzione detta *generica*; cioè che si può applicare a diversi tipi di dati. Di fatto accetta una invocazione di funzione e restituisce il risultato di quel calcolo. Nel mezzo legge l'ora di partenza e quella di fine e ne fa la differenza.
Prendetela per buona così com'è.

Se invochiamo `lev`:

```
val source = "pippo"
val target = "pappa"

lev(source.length, target.length)
```

otterremo la distanza tra le parole *pippo* e *pappa* e cioè 2.
In questa maniera invece, usando la funzione `time`, sapremo anche quanto ci metterà a calcolare questo valore:

```
val source = "pippo"
val target = "pappa"

time { lev(source.length, target.length) }
```

Provate sia con `lev` che con `lev_if` e noterete qualche differenza di velocità. Va chiarito che misurare, specialmente in questa maniera *casareccia*, la velocità di esecuzione, non va presa come riferimento. È solo una indicazione di massima.

Come vedete alla funzione `lev` servono le lunghezze delle parole da controllare, lunghezze che si ottengono con il metodo `length` delle stringhe di caratteri. Negli esempi ci sono oltre al codice qui presentato anche una implementazione più *avanzata* che utilizza una tecnica detta *memoization* e che consiste nel ricordarsi i valori intermedi già calcolati per non stare a perdere altro tempo. Questo sistema per esempio, vi potrebbe permettere di fare confronti di testi molto più lunghi in tempi *ragionevoli* altrimenti non raggiungibili.

L'algoritmo però, si può implementare anche in altri modi e senza usare la *ricorsività* delle funzioni. Vediamo come:

```
def lev_matrix(source: String, target: String): Int = {
  val n = source.length
  val m = target.length
  val d = Array.ofDim[Int](n + 1, m + 1)
  for (i <- 1 to n) d(i)(0) = i
  for (j <- 1 to m) d(0)(j) = j

  for {
    i <- 1 to n
    j <- 1 to m
  } {
    val cost = if (source(i - 1) == target(j - 1)) 0 else 1
    d(i)(j) = minimum(
      d(i - 1)(j) + 1,
      d(i)(j - 1) + 1,
      d(i - 1)(j - 1) + cost)
  }
  d(n)(m)
}
```

Qui è stata usata una *matrice* ovvero una tabella di valori bidimensionale. Il concetto è simile ma vedrete la differenza in velocità. La funzione costruirà lavorando sugli indici questa tabella:

		p	i	p	p	o
	0	1	2	3	4	5
p	1	0	1	2	3	4
a	2	1	1	2	3	4
p	3	2	2	1	2	4
p	4	3	3	2	1	2
a	5	4	4	3	2	2

e l'ultimo valore a destra in basso sarà il risultato cercato.
Essendo la funzione *non ricorsiva* non deve trasportarsi i dati nelle successive chiamate (come le parole da controllare) e quindi può essere *tutta compresa*. Per slegare invece completamente *la versione ricorsiva* dai valori esterni, come abbiamo fino ad ora fatto, possiamo usare la caratteristica di Scala di poter avere funzioni *annidate*:

```
def lev_match(source: String, target: String): Int = {
  def dist(i: Int, j: Int): Int = {
    (i, j) match {
      case (i, 0) => i
      case (0, j) => j
      case (i, j) => minimum(
        dist(i - 1, j) + 1,
        dist(i, j - 1) + 1,
        dist(i - 1, j - 1) + cost(i, j)
      )
    }
  }
  dist(source.length, target.length)
}
```

La distanza di Levenhstein, è utilizzata inoltre non solo per le parole ma può essere applicata anche con certe varianti a molte applicazioni dove si debba ricercare una similitudine tra dati come per esempio le immagini. Il campo applicativo è abbastanza ampio.

Bene, siete al primo passo per fare concorrenza al solito *blasonato* motore di ricerca.

5 Esempi

- disegna_poligoni.kojo
- figura_ripetitiva.kojo
- spirale.kojo
- prova_del_nove.kojo
- fotogrammi.kojo
- pipistrello.zip
- pipistrello.kojo
- fizzBuzz.kojo
- fizzBuzz2.kojo
- fizzBuzzWoof.kojo
- fizzBuzzWoof2.kojo
- personaggio.kojo
- personaggio2.kojo
- personaggio2.zip
- cifrario_vigenere.kojo
- cifrario_vernam.kojo
- lettere_e_colori.kojo
- lettere_e_colori2.kojo
- lettere_e_colori3.kojo
- lettere_e_colori4.kojo
- OpenWeatherMap.kojo
- Levenshtein1.kojo
- Levenshtein2.kojo