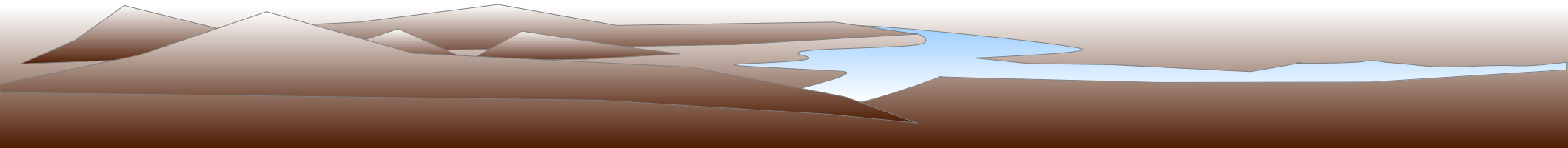
Master in Bioinformatica

Programmazione in Perl
Introduzione

Andrea Sterbini
sterbini@di.uniroma1.it



Informazioni sul corso

- Docente: Andrea Sterbini
- Email: sterbini@di.uniroma1.it
- Ricevimento: mercoledì ore 14-18
- Ufficio: via Salaria 113, 3° piano, stanza 309
- Telefono: 06-4991-8538
- Sito web: **http://twiki.di.uniroma1.it**
  - Cercate l'area **Bioperl**
- Orari lezioni: giovedì 14.30–17 e venerdì 17–19

Perchè il Perl?

- E' un linguaggio di **scripting**
  - Non va compilato -> è facile scrivere programmi
- Ottima **manipolazione di stringhe** di testo
  - Es. ricerca e manipolazione di stringhe di dna
- **Gestione di memoria "trasparente"** x l'utente
  - Costruzione/distruzione automatica di strutture in memoria
- Ha molti modi/stili per fare le stesse cose
  - (è detto "la motosega della programmazione")
- Esiste la libreria di programmi **BioPerl**!!!

Riferimenti utili

- Un ottimo libro di testo per principianti (Perl):
  - "**Beginning Perl for Bioinformatics**",
    James D. Tisdall, O'Reilly
- Per esperti e per proseguire (OOP e cenni di BioPerl):
  - "**Mastering Perl for Bioinformatics**",
    James D. Tisdall, O'Reilly
- Perl per windows: **www.activestate.com**
- La libreria BioPerl: **www.Bioperl.org**
  - BioPerl per ActivePerl si installa facilmente da rete

Livelli di programmazione in Perl

- Livello 1 (the "**quick and dirty**" way)
  - Mettere in sequenza le istruzioni senza preoccuparsi di definire funzioni, moduli o di dichiarare le variabili
  - Perl cerca di fare "quello che vogliamo" (DWIM)
- Livello 2 (programmazione **per funzionalità**)
  - Definire procedure/funzioni da riutilizzare
  - Definire tutte le variabili per migliorare i controlli sintattici
- Livello 3 (programmazione **ad oggetti**)
  - Strutture dati unite alle funzionalità che vi agiscono
  - Programma = interazione e dialogo tra oggetti

Concetti di base

- **Valori**: numeri interi, numeri decimali, testo
- **Strutture dati**: gruppi di valori organizzati
  - Es. scheda di una agenda: nome, cognome, n.tel.
- **Variabili**: nomi che individuano zone di memoria in cui sono inseriti e manipolati i dati
- **Procedure**: sequenze di istruzioni che possono essere riusate (chiamate) in più parti del programma
- **Funzioni**: procedure che calcolano un valore
- **Oggetti**: strutture dati + azioni che la manipolano

Valori

- Valori che possono essere rappresentati
  - **Interi**: numeri interi col segno (es. 1, 2, 3, -42)
  - **Decimali**: numeri con la virgola (es. 3.1415)
  - **Caratteri**: lettere/cifre di un testo (es. 'A', 'B', 'C')
  - **Stringa**: sequenza di caratteri circondata da “ o '
    - “testo con **$nomi** di variabile” (doppi apici: interpolato)
    - 'testo non interpolato' (apici singoli: non interpolato)
  - **vero/falso**: 0, “”, undef = falso, altri valori = vero
  - **undef**: valore non definito (non ancora assegnato)

Tipi di valori contenuti nelle variabili

- Tipi: scalare, vettore/lista, tabella hash
  - **Scalare**: contiene un intero, decimale o stringa

```
my $scalare = 42;
```

  - **Vettore/lista**: contiene una sequenza di valori

```
my @vettore = (1, 2, 3, 4);
```

  - **Tabella di hash**: contiene coppie chiave => valore
    - La **chiave** è una **stringa**, il **valore** è uno **scalare**
    - Vengono utilizzate per realizzare strutture complesse

```
my %tabella = {  nome      => 'Andrea'    ,
                 cognome   => 'Sterbini' ,
                 telefono  => '0649918538' };
```

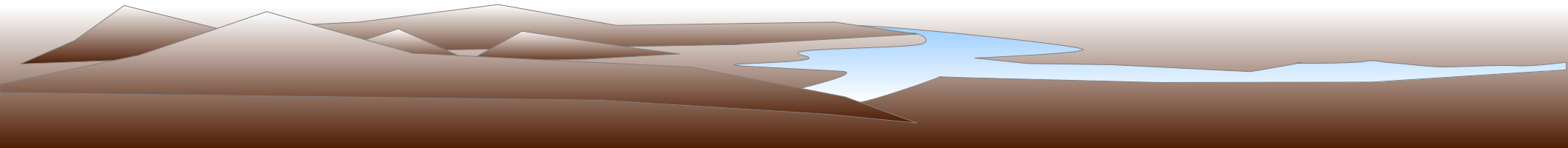
Le variabili e lo stile quick-and-dirty

```
$distanza = sqrt($dx * $dx + $dy * $dy);
$testo = "Ho preso " . $voto;   # (concatenato)
$testo = "Ho preso $voto";      # (interpolato)
@altezze = (186, 160, 175);
$somma = $altezze[0] + $altezze[1] + $altezze[2];
$media = $somma/3;
```

DWIM: Perl crea automaticamente lo spazio per le variabili in memoria (attimo, no?)

PROBLEMA: ma se faccio errori di battitura?
non me ne accorgerò **MAI**!

SOLUZIONE: "**use strict**" e **my** per ogni variabile



Elementi di stile

- Inserite SEMPRE descrizioni di cosa fa il prog.

```
# commento che spiega cosa volete e come
```

- Intestazione

```
#!/usr/bin/perl -w    # interprete del programma
use strict;           # attiva controlli aggiuntivi
```

- Definite SEMPRE tutte le variabili

```
my $nome = valoreiniziale;
```

- Le istruzioni finiscono con ; (punto e virgola)
- I blocchi di istruzioni sono circondati da {} (graffe)

Usare vettori (ovvero liste)

- Li definisco col carattere @ (“at” o chiocciola)

```
my @vettore = (1, 2, 3, 4, 5);
```

- Gli elementi invece li manipolo con **$nome[*indice*]**
  - **ATTENZIONE**: gli indici degli elementi iniziano da **0**
  - Es.: inserisco un nuovo elemento all'indice 7

```
$vettore[7] = 42;  # inserisco l'ottavo
  elemento
```

    - (**DWIM**: Perl crea automaticamente gli elementi intermedi)
  - Es.: leggo (e stampo) un elemento

```
print $vettore[6];
```

Usare tabelle (di hash)

- Le definisco col carattere **%** (percento)

```
my %scheda = {  nome       => 'Andrea',
                telefono   => '06-4991-8538' };
```

- I valori li manipolo con **$tabella{chiave}**
  - Es.: inserimento di un valore

```
$scheda{cognome} = 'Sterbini';
```

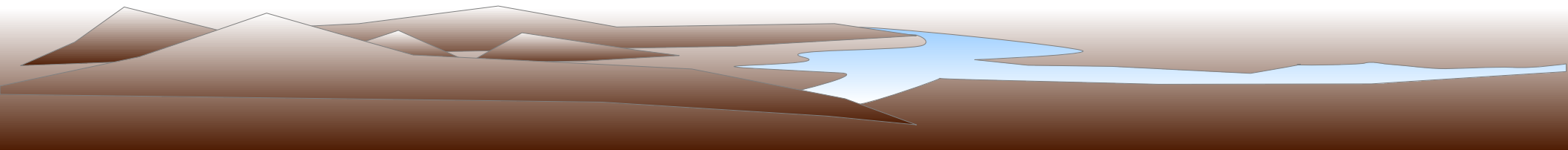
  - Es.: leggo e stampo tutti i “campi” della scheda

```
print “$scheda{nome} $scheda{cognome}
  $scheda{telefono}” ;
```

  - Se un elemento non esiste torna **undef**

Modificare il flusso del programma

- Per modificare il flusso di esecuzione del programma si usano istruzioni che svolgono compiti diversi a seconda di determinate **condizioni**
- **if-then-else** e sue varianti
  - if-then, if-then-elsif-...else
  - Realizza due (o più) percorsi alternativi nel programma
- **switch-case**
  - Semplifica la scrittura di una sequenza di test simili
- Operazioni ripetute (cicli)



If-then-else

- if-then-else

```
if (condizione) {
    # istruzioni eseguite se "condizione" è vera
  } else {
    # istr. eseguite se "condizione" è falsa
  }
```

- Le **condizioni** sono test che danno risultato vero/falso
  - Ricordate che 0, "" e undef significano **falso**, il resto **vero**
- Sintassi alternative dell' **if-then** semplice:

```
istruzione if condizione;       # se vero
istruzione unless condizione;   # se falso
```

Condizioni semplici e complesse

- Valori vero e falso: 0, '' (stringa vuota) e undef sono considerati **falso,** Il resto è considerato **vero**
- Test elementari (danno un risultato vero/falso)

```
0 == $nome          $nome != 0
$nome > 0           $nome <= 0
$nome eq 'Andrea'   $nome ne 'Andrea'
```

- Creo condizioni complesse con NOT, AND e OR:

NOT (nego la condizione): `! condizione`
AND (tutte vere): `condizione1 && condizione2`
OR (almeno una vera): `condizione1 || condizione2`

Ripetere un blocco di istruzioni

- Un numero fissato di volte

```
for ($i=0 ; $i<$limite ; $i++) {
  #istruzioni da ripetere }
```

- Un numero variabile di volte (anche 0 /almeno 1)

```
while (condizione) {
     # istruzioni da ripetere }
do { #istruzioni da ripetere
     } while (condizione);
```

- Per ogni elemento di una lista

```
foreach $elemento (@lista) {
  # istruzioni da ripetere }
```