

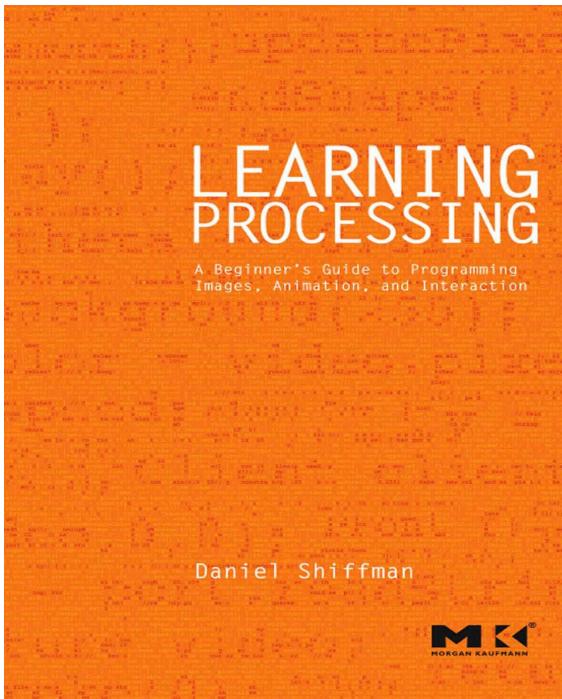


# FONDAMENTI DI INFORMATICA

Alma Artis – Anno Accademico 2017/2018  
Salvatore Rinzivillo (ISTI, CNR)

Introduzione a Processing

# LIBRI E RIFERIMENTI



- Introduzione
- Capitolo 1-3
- Registrarsi al sito:  
<http://almaartis.rinziv.it/>

Learning Processing– Second Edition  
Daniel Shiffman  
Available here: <http://learningprocessing.com/>



# SOMMARIO

# INFORMAZIONE

- Tutto ciò che può essere rappresentato all'interno di un computer
  - Numeri, caratteri, immagini, suoni
  - Comandi e sequenze di comandi che il calcolatore esegue per trasformare l'informazione
- Il **computer** (elaboratore elettronico) è lo strumento per rappresentare ed elaborare l'informazione

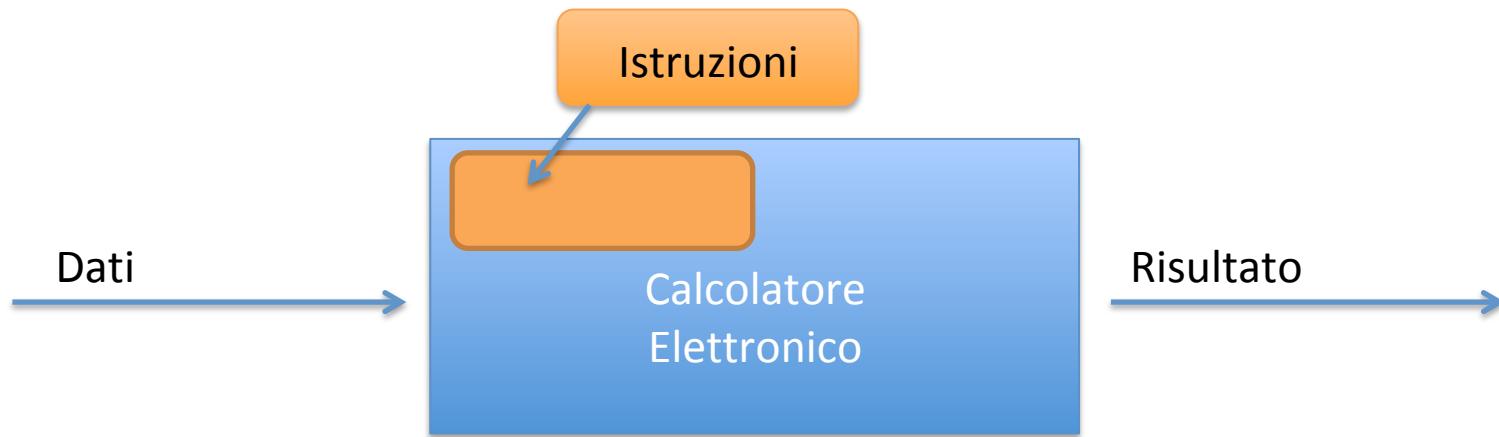
# CALCOLATORE ELETTRONICO

Una macchina che:

- Ha un meccanismo di input per acquisire le richieste
- Ha un dispositivo per memorizzare le informazioni
- Ha la capacità di elaborare i dati
- Ha un dispositivo per comunicare al risposta

# CALCOLATORE ELETTRONICO UNIVERSALE

- Il calcolatore può essere programmato per eseguire un insieme di azioni sui dati per risolvere un dato problema (o funzione)



# FUNZIONI CALCOLABILI

- Di tutte le funzioni possibili, solo un sottoinsieme sono **calcolabili**, ovvero esiste un **algoritmo** che è in grado di fornire un risultato per i dati in input
- L'informatica si occupa di trovare risposte a domande del tipo:
  - Quante e quali sono le funzioni calcolabili?
  - Quale sono le funzioni che una data macchina può calcolare?
  - Esiste una macchina che calcoli tutte le (infinite) funzioni calcolabili?

# ALGORITMO

- E' un insieme ordinato di azioni che risolve un dato problema (funzione) P
- L'esecuzione dell'algoritmo è affidata ad un **esecutore** (non necessariamente un calcolatore) in grado di decifrare le azioni nella sequenza
- Nella vita quotidiana tutti eseguiamo algoritmi:
  - Montare un mobile componibile
  - Cambiare la cartuccia della stampante
  - Prendere un caffè alla macchina a gettoni

# ESERCIZIO

- Definire una attività quotidiana come un algoritmo astratto
  - Definire le istruzioni elementari
  - Definire l'algoritmo come sequenza di istruzioni elementari



# LNGUAGGI DI PROGRAMMAZIONE

# ALGORITMI E LINGUAGGI DI PROGRAMMAZIONE

- Nel caso di un elaboratore elettronico è necessario:
  - Conoscere l'insieme di istruzioni che è in grado di interpretare
  - Conoscere i tipi di dati che può rappresentare
- Questi due aspetti sono centrali nella definizione di un **Linguaggio di Programmazione**

# ALGORITMI E PROGRAMMI

- Dato un **problema P**, la sua soluzione si identifica secondo la seguente procedura
  - Individuare un **metodo risolutivo**
  - Trasformare il metodo in un insieme ordinato di azioni: **algoritmo**
  - Rappresentare dati e azioni attraverso un formalismo comprensibile dal calcolatore (il linguaggio di programmazione): **programma**
- Una volta che il programma è stato creato, potrà essere utilizzato per risolvere ogni istanza del problema P



# QUALE LINGUAGGIO DI PROGRAMMAZIONE?

- I linguaggi di programmazione sono tutti equivalenti
- Dati due linguaggi  $L_1$  e  $L_2$  e due programmi che risolvono la stessa funzione  $f$  nei due linguaggi  $D_{L_1}(f)$  e  $D_{L_2}(f)$
- Allora esiste una funzione calcolabile che traduce  $D_{L_1}$  in  $D_{L_2}$  e viceversa

# IL LINGUAGGIO MACCHINA

- Il linguaggio direttamente eseguibile da un calcolatore si chiama linguaggio macchina
- E' un linguaggio poco comprensibile per gli umani
- Le operazioni disponibili sono molto semplici
- Sono specificate in notazione binaria: sequenze di 1 e 0
- Lo codifica di una funzione complessa richiede la scrittura di un lungo programma, a volte incomprensibile
- In caso di errori o malfunzionamenti, è difficile trovare gli eventuali errori

# SOMMA DEI PRIMI 10 NUMERI

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

# SOMMA DEI PRIMI 10 NUMERI

1. Store the number 0 in memory location 0.
2. Store the number 1 in memory location 1.
3. Store the value of memory location 1 in memory location 2.
4. Subtract the number 11 from the value in memory location 2.
5. If the value in memory location 2 is the number 0,  
continue with instruction 9.
6. Add the value of memory location 1 to memory location 0.
7. Add the number 1 to the value of memory location 1.
8. Continue with instruction 3.
9. Output the value of memory location 0.

# SOMMA DEI PRIMI 10 NUMERI

```
Set “total” to 0.  
Set “count” to 1.  
[loop]  
  Set “compare” to “count”.  
  Subtract 11 from “compare”.  
  If “compare” is zero, continue at [end].  
  Add “count” to “total”.  
  Add 1 to “count”.  
  Continue at [loop].  
[end]  
Output “total”.
```

Questo linguaggio primitivo permette di verificare se un valore è zero

Necessità di inserire etichette per controllare la sequenza di operazioni

# SOMMA DEI PRIMI 10 NUMERI

```
var total = 0, count = 1;  
while (count <= 10) {  
    total += count;  
    count += 1;  
}  
console.log(total);  
// → 55
```

Non è necessario specificare  
i salti all'esterno del ciclo di  
somme

# SOMMA DEI PRIMI 10 NUMERI

```
console.log(sum(range(1, 10)));  
// → 55
```

Introduzione di nuovi operatori: range e sum



**PIXELS**

# COORDINATE

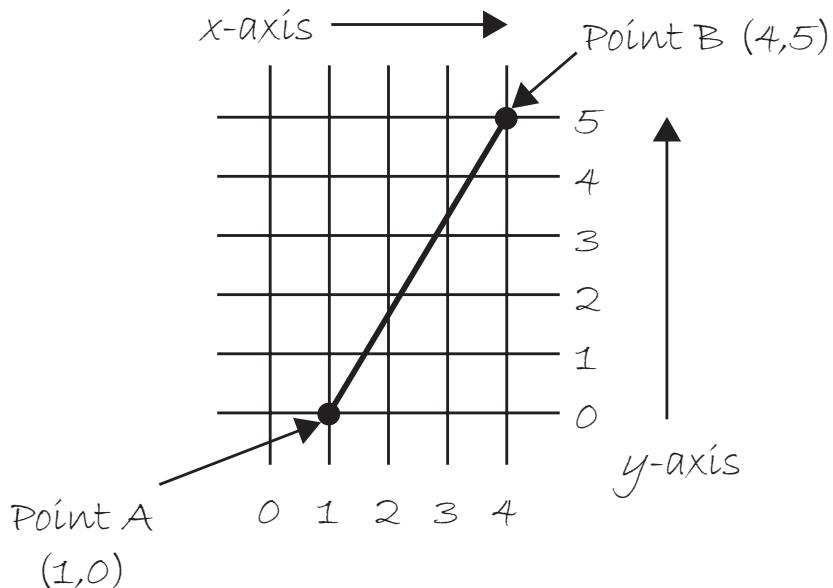


fig. 1.1

```
line(1,0,4,5);
```

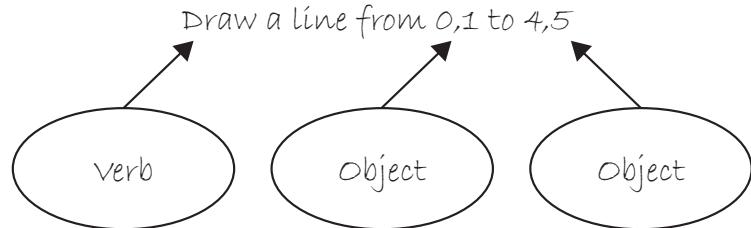
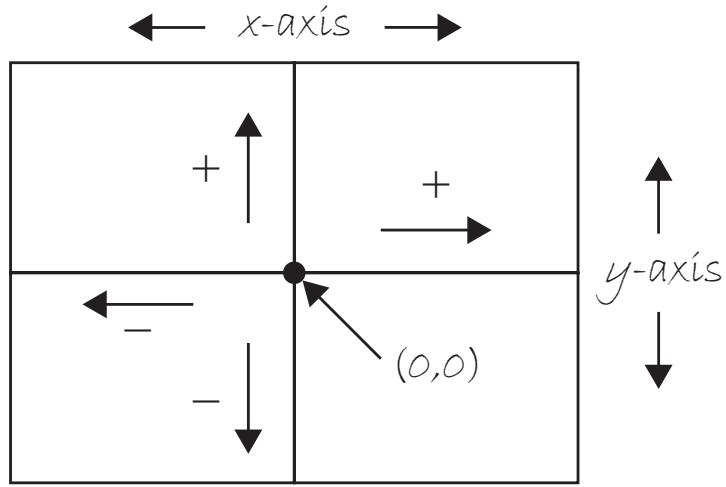


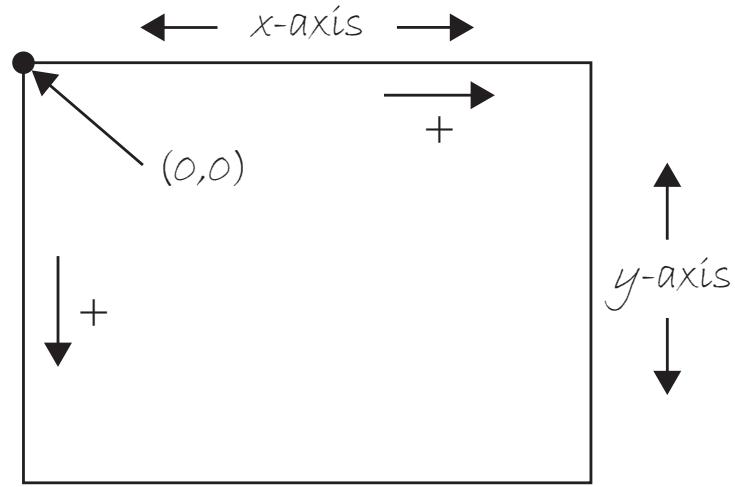
fig. 1.2

# SISTEMA DI RIFERIMENTO



Eighth grade

fig. 1.3



Computer

# FORME DI BASE



Point

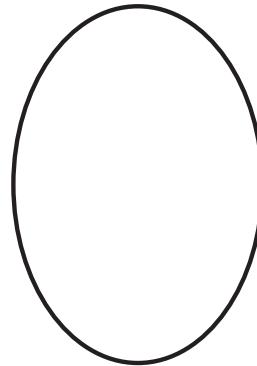
*fig. 1.4*



Line

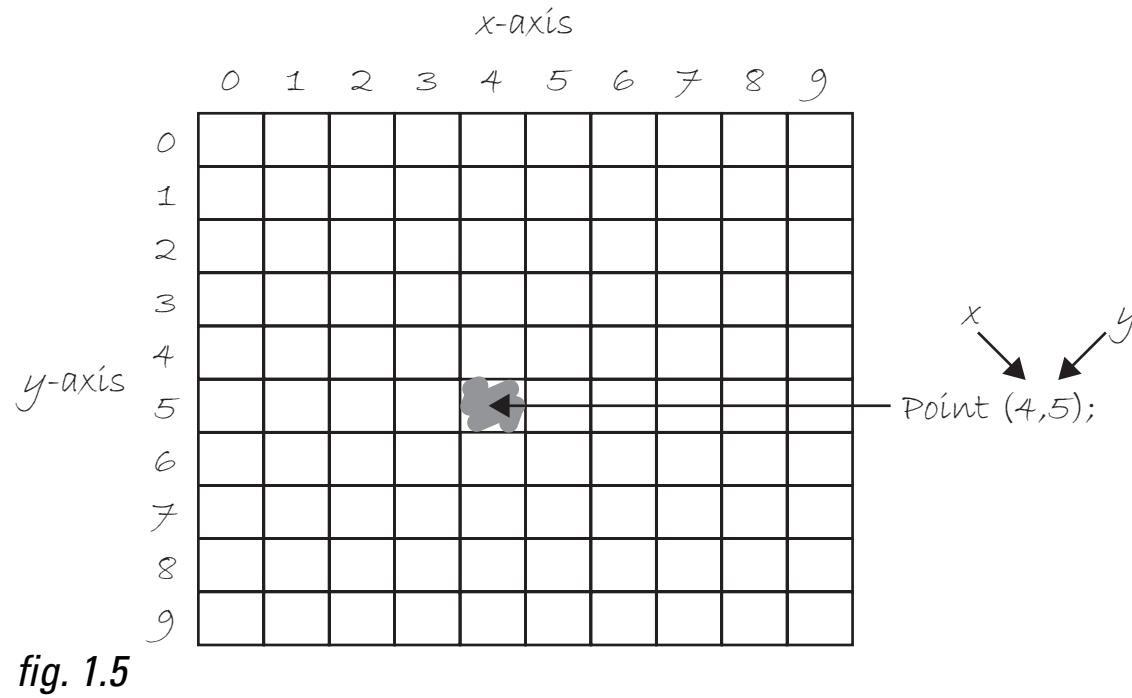


Rectangle



Ellipse

# FORME DI BASE: POINT



# FORME DI BASE: LINE

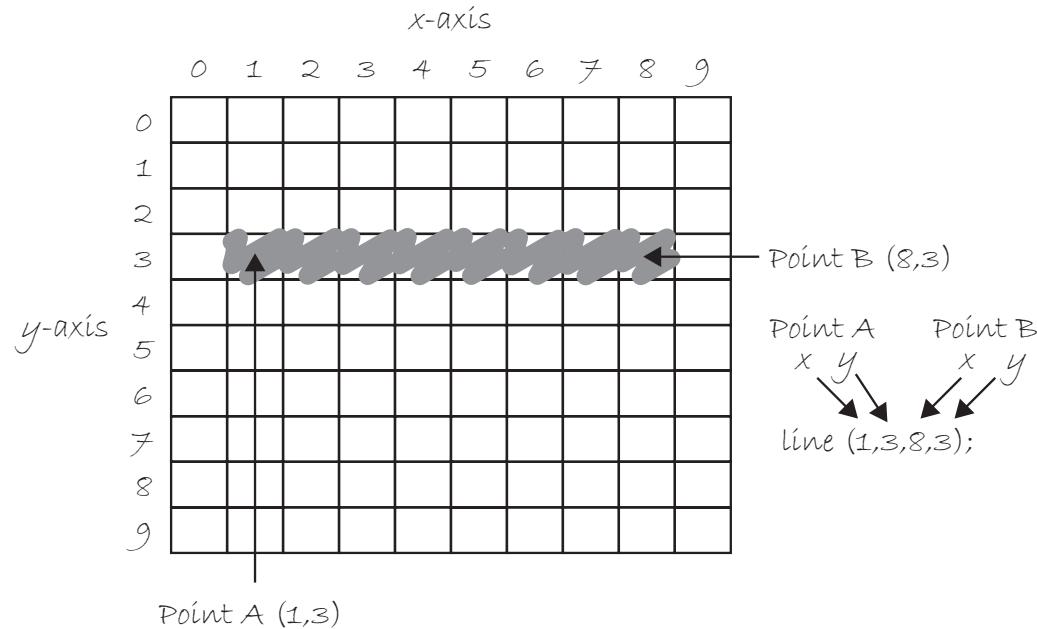


fig. 1.6

# FORME DI BASE: RECT

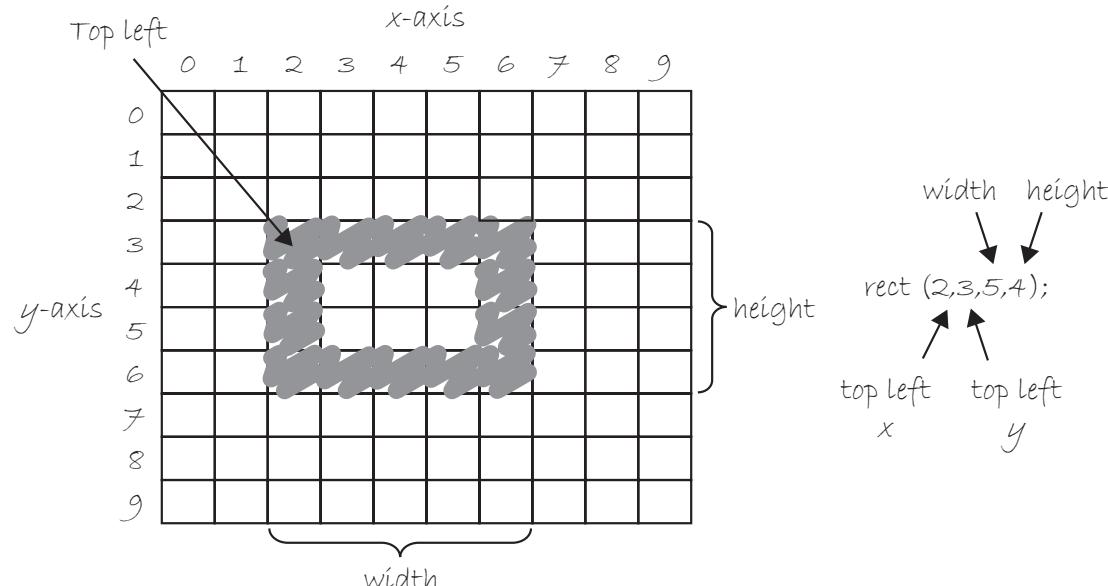


fig. 1.7

# FORME DI BASE: RECT (2)

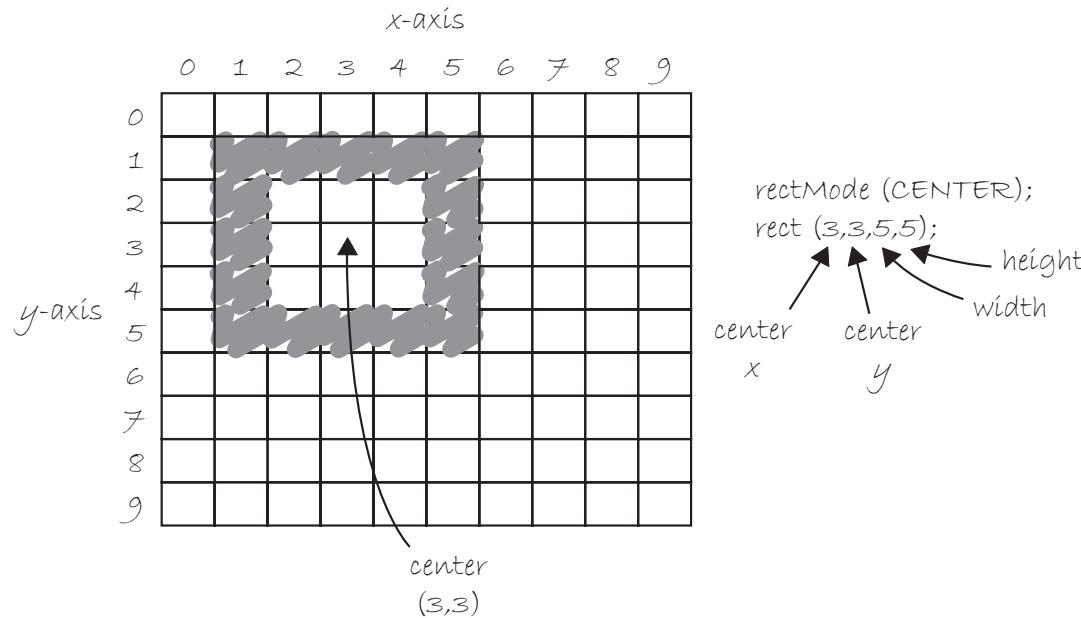


fig. 1.8

# FORME DI BASE: RECT (3)

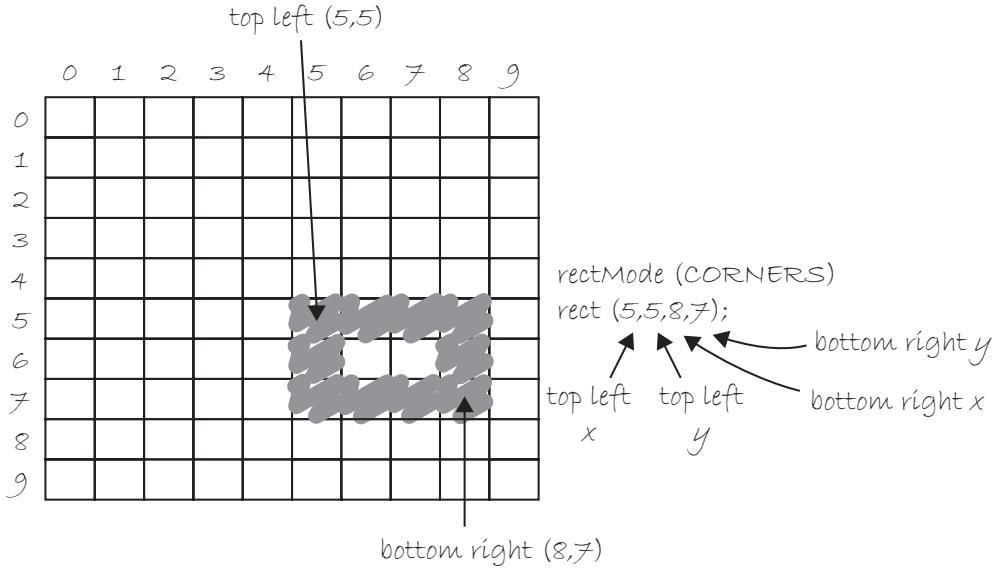
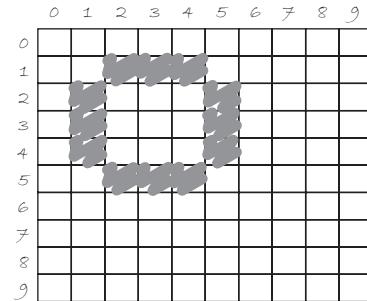
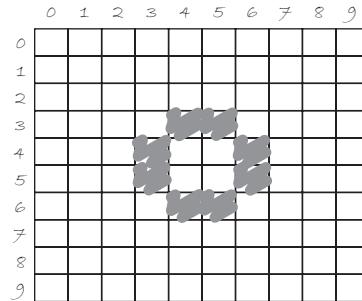


fig. 1.9

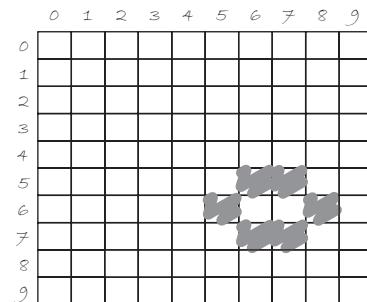
# FORME DI BASE: ELLIPSE



```
ellipseMode (CENTER);  
ellipse (3,3,5,5);
```



```
ellipseMode (CORNER);  
ellipse (3,3,4,4);
```

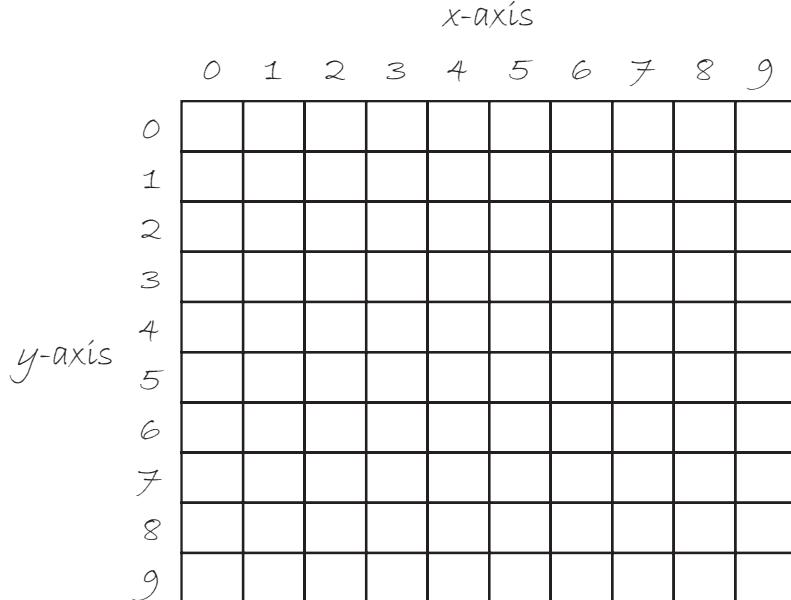


```
ellipseMode (CORNERS);  
ellipse (5,5,8,7);
```

fig. 1.10

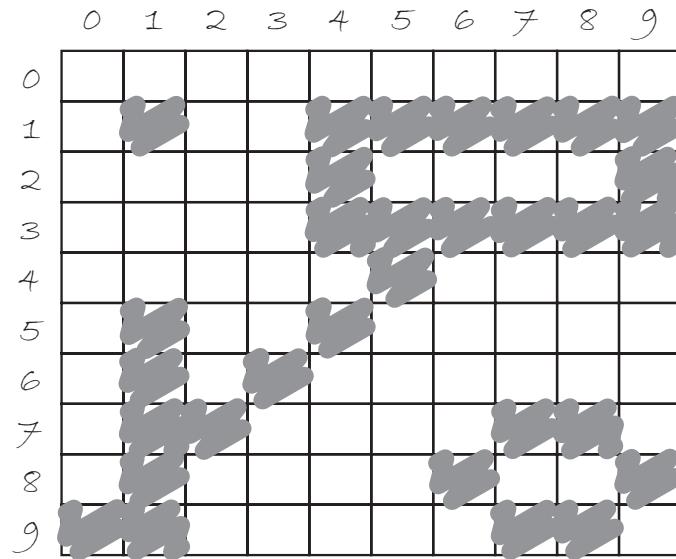
# ESERCIZIO

```
line(0,0,9,6);  
point(0,2);  
point(0,4);  
rectMode(CORNER);  
rect(5,0,4,3);  
ellipseMode(CENTER);  
ellipse(3,7,4,4);
```



Disegnare le forme descritte dal codice a sinistra

# ESERCIZIO



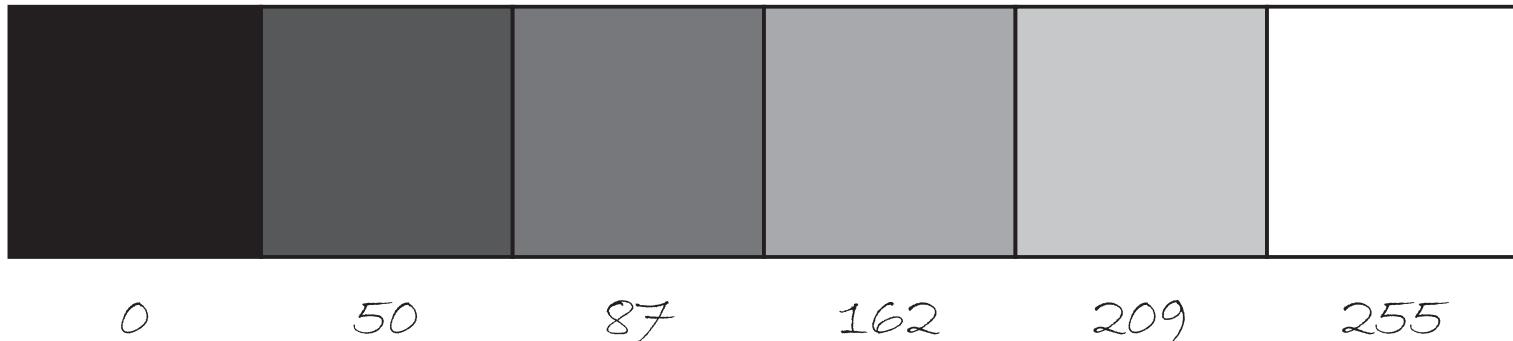
Note: There is more than one correct answer!

Scrivere un codice per riprodurre il diagramma



**COLORI**

# SCALA DI GRIGIO: LUMINOSITÀ



*fig. 1.13*

I livello di luminosità è rappresentato come un valore compreso tra 0 (nero) e 255 (bianco)

# PARTI DI UNA FORMA GEOMETRICA

- Stroke: è la parte che definisce i contorni di una forma
  - Esempio: per un rettangolo sono i 4 lati
- Fill: è la parte interna della forma
  - Esempio: per un rettangolo, l'area racchiusa dai 4 lati

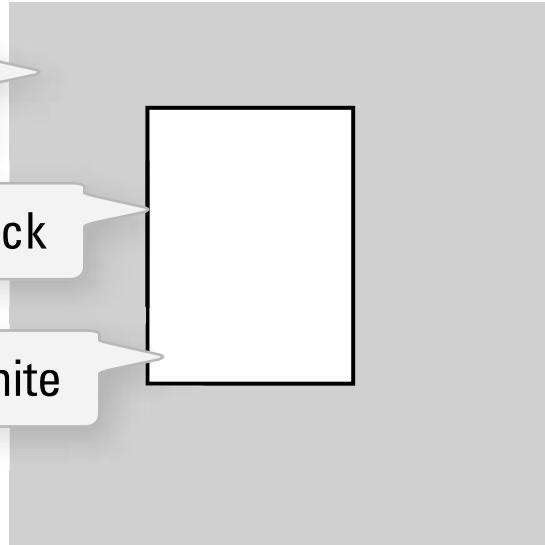
# VALORI DI DEFAULT PER stroke E fill

```
rect(50,40,75,100);
```

The background color is gray.

The outline of the rectangle is black

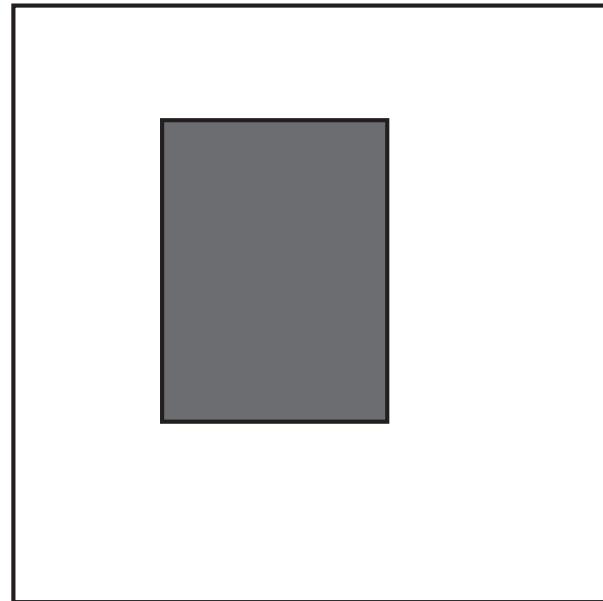
The interior of the rectangle is white



*fig. 1.14*

# stroke **E** fill

```
background(255) ;  
stroke(0) ;  
fill(150) ;  
rect(50,50,75,100) ;
```



*fig. 1.15*

# stroke & fill

```
background(255);  
stroke(0);  
noFill();  
ellipse(60,60,100,100);
```

***nofill()*** leaves the shape  
with only an outline

If we draw two shapes at one time, *Processing* will always use the most recently specified ***stroke()*** and ***fill()***, reading the code from top to bottom. See Figure 1.17.

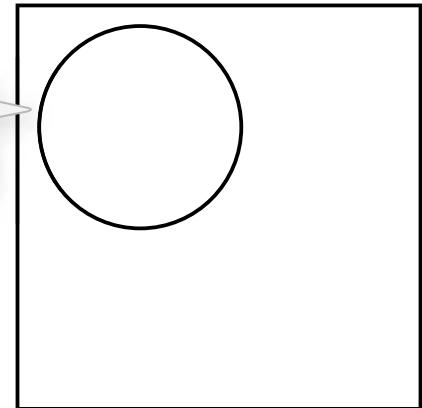


fig. 1.16

# stroke & fill

```
background(150);  
stroke(0);  
line(0, 0, 100, 100);  
stroke(255);  
noFill();  
rect(25, 25, 50, 50);
```

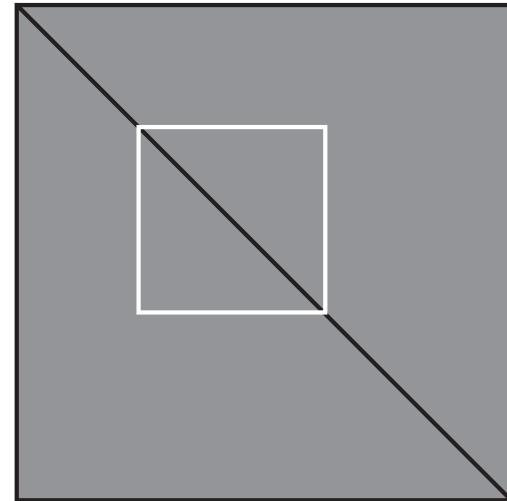
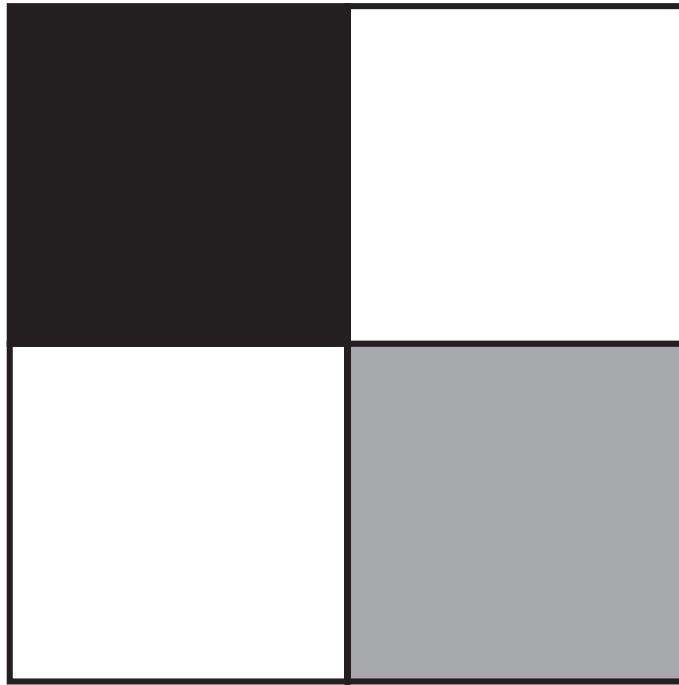


fig. 1.17

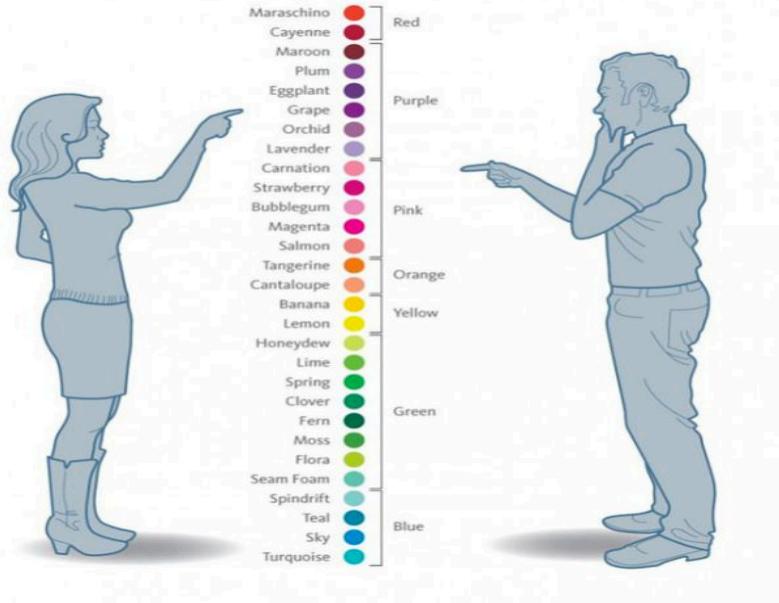
# ESERCIZIO



- Scrivere le istruzioni per creare il diagramma a sinistra

# HOW MANY COLOR?

Female



Male

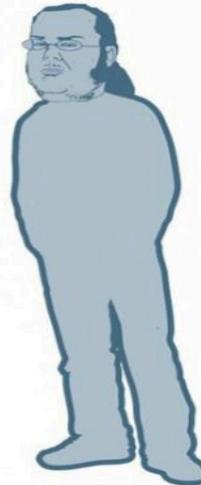


Dog



Programmer

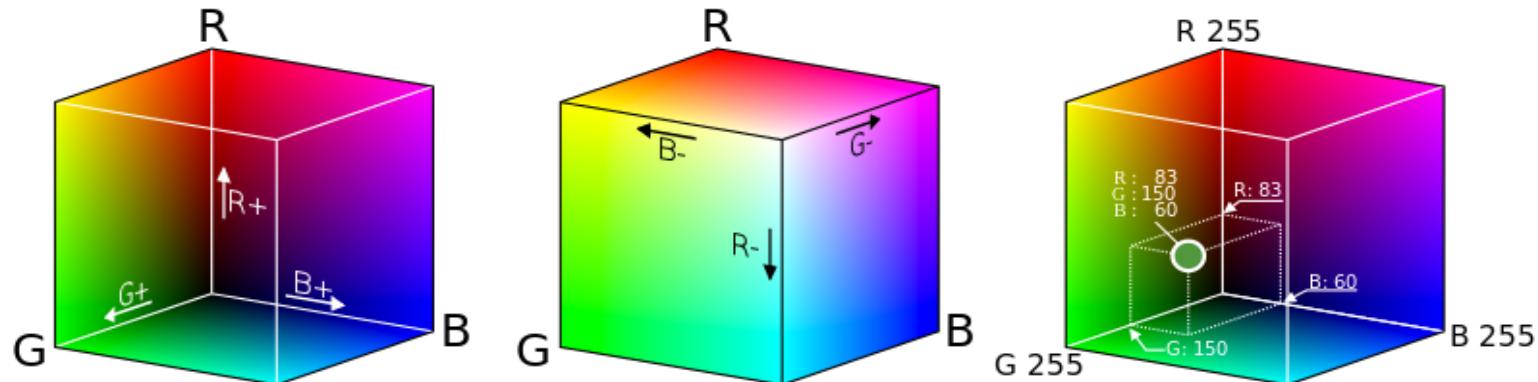
Gray #f94433  
Gray #ac203b  
Gray #85343d  
Gray #874994  
Gray #663c84  
Gray #8c2590  
Gray #a16799  
Gray #af99c7  
Gray #f38da3  
Gray #d2157b  
Gray #ec90b7  
Gray #e90086  
Gray #f57d7e  
Gray #f27727  
Gray #fc9b7b  
Gray #f7d305  
Gray #f1e311  
Gray #ccdf62  
Gray #68bd46  
Gray #0aae4f  
Gray #069665  
Gray #057054  
Gray #3ba246  
Gray #abcf37  
Gray #68c3b2  
Gray #8bccd0  
Gray #0687a7  
Gray #078dca  
Gray #0fb8b5



WeKnowMemes

# RGB COLOR MODEL

- Basato sui colori primari
- È un modello additivo: il colore risultante è la somma dei componenti di base



# RGB COLOR MODEL

- I valori R,G,B possono essere espressi come numeri nell'intervallo [0,1]
- Alcune applicazioni (e anche processing) usano l'intervallo [0,255]
- Non è un modello intuitivo. Come definire i valori per un marrone chiaro?

# COLORI RGB

## Example 1-3: RGB color

```
background(255) ;  
noStroke() ;  
  
fill(255, 0, 0) ;  
ellipse(20, 20, 16, 16) ;  
  
fill(127, 0, 0) ;  
ellipse(40, 20, 16, 16) ;  
  
fill(255, 200, 200) ;  
ellipse(60, 20, 16, 16) ;
```

- Bright red
- Dark red
- Pink (pale red).



fig. 1.18

# TRASPARENZA

## Example 1-4: Alpha transparency

```
background(0);  
noStroke();  
  
fill(0,0,255);  
rect(0,0,100,200);  
  
fill(255,0,0,255);  
rect(0,0,200,40);  
  
fill(255,0,0,191);  
rect(0,50,200,40);  
  
fill(255,0,0,127);  
rect(0,100,200,40);  
  
fill(255,0,0,63);  
rect(0,150,200,40);
```

No fourth argument means 100% opacity.

255 means 100% opacity.

75% opacity

50% opacity

25% opacity



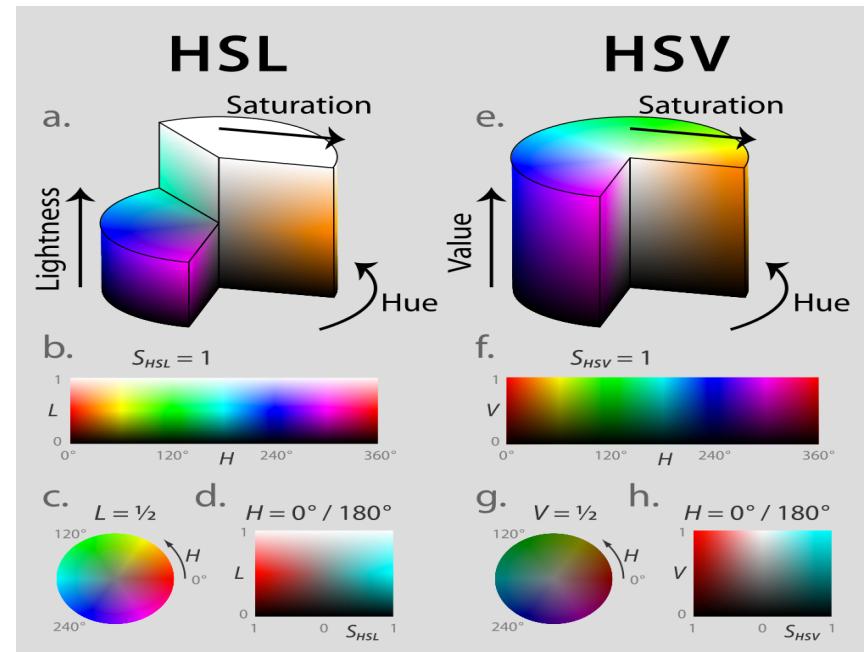
fig. 1.20

# COLOR MODE

- I range di colori sono riconosciuti solo come intervalli tra 0 e 255
- E' possibile definire una mappa diversa utilizzando l'istruzione `colorMode`
  - `colorMode(RGB,100)`
- Alternative color model: HSB
  - Hue: il tipo di colore da usare (varia tra 0 e 360)
  - Saturation: vivicità del colore (varia tra 0 e 100)
  - Brightness: luminosità (varia tra 0 e 100)

# HSV COLOR MODEL

- Basato sui concetti intuitivi di
  - tinta
  - saturazione
  - Intensità o valore



"Hsl-hsv models" by Jacob Rus - Own work. Licensed under CC BY-SA 3.0 via Wikimedia Commons - [http://commons.wikimedia.org/wiki/File:Hsl-hsv\\_models.svg#/media/File:Hsl-hsv\\_models.svg](http://commons.wikimedia.org/wiki/File:Hsl-hsv_models.svg#/media/File:Hsl-hsv_models.svg)

# ESERCIZIO

Descrivere una creatura usando semplici forme geometriche

Example 1-5: Zoog

```
ellipseMode(CENTER);  
rectMode(CENTER);  
stroke(0);  
fill(150);  
rect(100,100,20,100);  
fill(255);  
ellipse(100,70,60,60);  
fill(0);  
ellipse(81,70,16,32);  
ellipse(119,70,16,32);  
stroke(0);  
line(90,150,80,160);  
line(110,150,120,160);
```

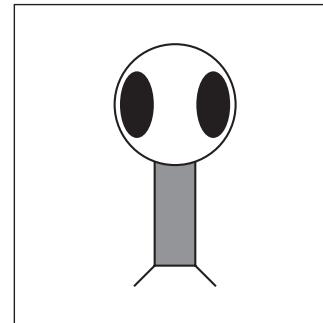


fig. 1.21



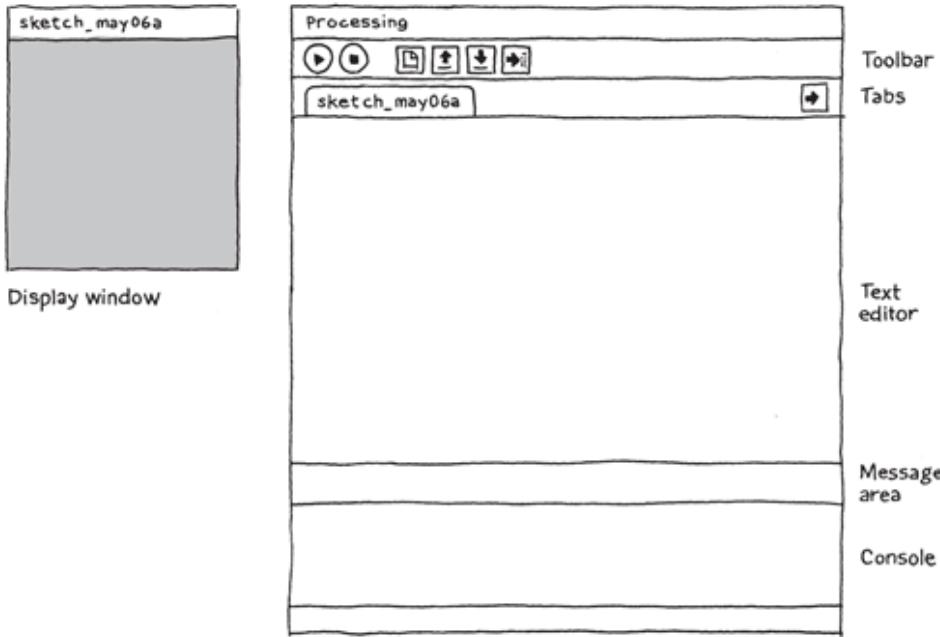
A decorative header pattern at the top of the page consists of six horizontal panels. From left to right: 1. A panel of light blue hexagonal tiles on a dark navy background. 2. A panel featuring nested diamond shapes in light blue, medium blue, and white. 3. A panel with a grid of triangles in white, medium blue, and dark navy. 4. A panel of light blue hexagonal tiles on a dark navy background. 5. A panel featuring nested diamond shapes in light blue, medium blue, and white. 6. A panel with a grid of triangles in white, medium blue, and dark navy.

# INTRODUCING PROCESSING

# PROCESSING.ORG INSTALLATION

- Download IDE from <http://processing.org>
- Expand archive file
- Execute the application

# ANATOMY OF CODE EDITOR



# ANATOMY OF CODE EDITOR



Execute current Sketch



Stop execution of current Sketch



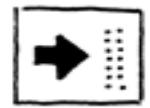
Save current Sketch



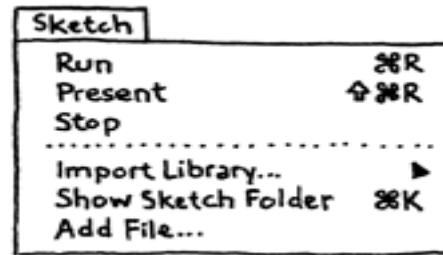
Load an existing Sketch from disk



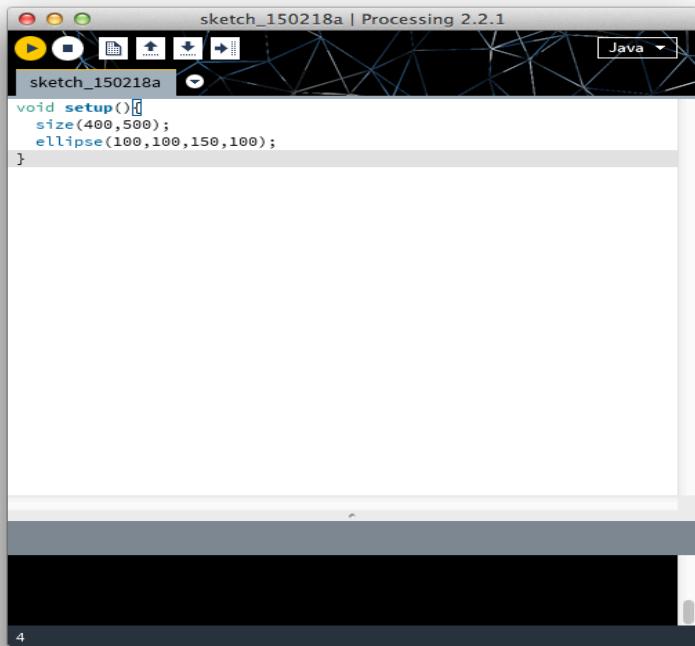
Start a new Sketch



Export current Sketch as a standalone app



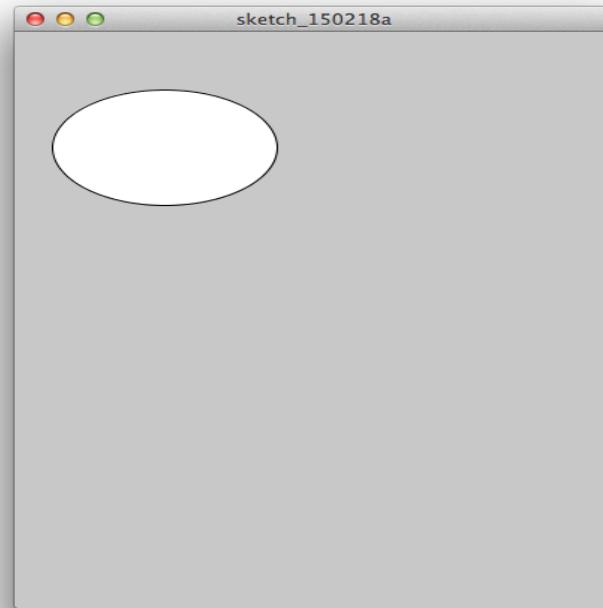
# OUR FIRST SKETCH



The screenshot shows the Processing 2.2.1 IDE interface. The title bar reads "sketch\_150218a | Processing 2.2.1". The toolbar includes icons for play, stop, file, and export. A dropdown menu says "Java". The code editor contains the following Java code:

```
void setup(){  
    size(400,500);  
    ellipse(100,100,150,100);  
}
```

The preview window is empty.



# CODICE IN PROCESSING

- Possiamo scrivere tre tipi di istruzioni
  - Chiamare delle funzioni
  - Assegnare un valore ad una variabile
  - Strutture di controllo

The diagram shows a single line of Processing code: `Line (0,0,200,200);`. A bracket labeled "Arguments in parentheses" spans from the opening parenthesis to the closing parenthesis. An arrow labeled "Function name" points to the word "Line". Another arrow labeled "Ends with semi-colon" points to the final semi-colon.

Line (0,0,200,200);

Function name

Arguments in parentheses

Ends with semi-colon

fig. 2.4

# MESSAGGI DI ERRORE

The screenshot shows the Processing IDE interface with the title bar "MyFirstProgram | Processing 0135 Beta". Below the title bar is a toolbar with various icons. The main area displays a sketch titled "MyFirstProgram" containing the following code:

```
size(200,200);
background(255);
rectMode(CENTER);
stroke(0);
fill(150);
rect(100,100,20,100);

fill(255);
ellipse(100,70,60,60); ← Line 9 highlighted

fill(0);
ellipse(81,70,16,32);
ellipse(119,70,16,32);

stroke(0);
line(98,150,80,160);
line(118,150,120,160);
println("Take me to your leader!");
```

An arrow points from the text "Line 9 highlighted" to the line `ellipse(100,70,60,60);`. Another arrow points from the text "Error message" to the error message in the status bar: "No method named "ellipse" was found in type "Temporary\_3082\_2001". However, there is an accessible method "ellipse" whose name closely matches the name "ellipse"." A third arrow points from the text "Error message again!" to the same error message repeated in the status bar.

fig. 2.6

# ESERCIZIO: TROVA L'ERRORE

*Exercise 2-6: Fix the errors in the following code.*



size(200,200); \_\_\_\_\_

background(); \_\_\_\_\_

stroke 255; \_\_\_\_\_

fill(150) \_\_\_\_\_

rectMode(center); \_\_\_\_\_

rect(100,100,50); \_\_\_\_\_



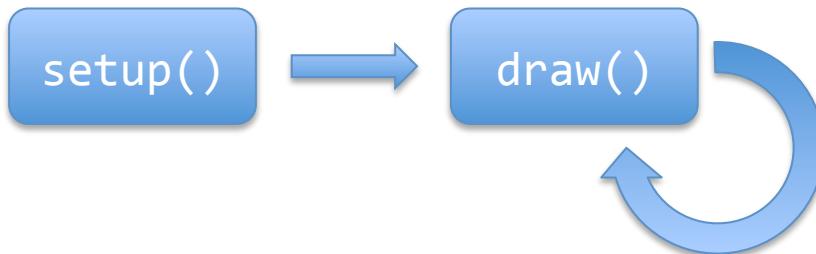
# INTERACTION

# FLUSSO DI UN PROGRAMMA PROCESSING

- Due fasi principali
  - SETUP
    - Questa parte viene eseguita all'inizio e ha lo scopo di configurare lo sketch per l'esecuzione
  - DRAW
    - Questa parte disegna la finestra ciclicamente, più volte al secondo. Aggiornando la visualizzazione si aggiorna anche lo stato del sistema
- Esempio: considera la corsa di una maratona
  - SETUP: indossa le scarpe e l'abbigliamento sportivo
  - DRAW: metti un piede davanti all'altro. Dopo 42km fermati.

# FUNZIONI `setup()` E `draw()`

- `setup()` prepara lo spazio di visualizzazione e inizializza lo stato interno del programma. Viene eseguita solo una volta!
- `draw()` viene chiamata ripetutamente fino a quando il programma è in esecuzione. Ha il compito di aggiornare lo stato interno del programma e la visualizzazione
  - Sfruttiamo le continue esecuzioni di `draw()` per monitorare lo stato del sistema



# BLOCCO DI CODICE

- Un **blocco** è un insieme di istruzioni racchiuse tra parentesi graffe {}
- Un blocco può contenere al suo interno altri blocchi
  - Di solito i blocchi annidati dentro altri sono pure indentati, ovvero hanno una rientranza nell'inizio della linea di codice
- Due blocchi di codice rilevanti sono **setup()** e **draw()**

# FUNZIONI setup() E draw()

What's this?

What are these for?

```
void setup() {  
    // Initialization code goes here  
}
```

```
void draw() {  
    // Code that runs forever goes here  
}
```

fig. 3.1

# FUNZIONI `setup()` E `draw()`

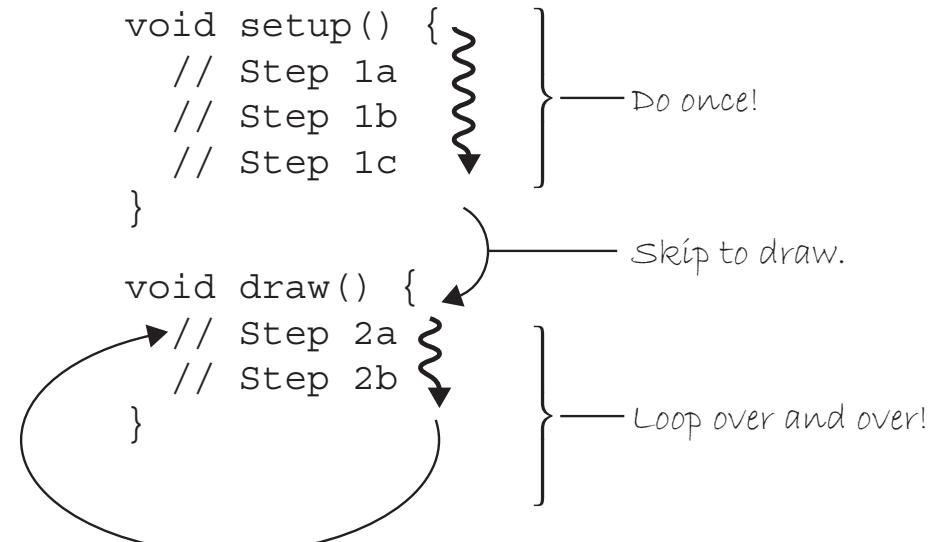


fig. 3.2

# ESERCIZIO: ZOOG

```
void setup(){
    // Set the size of the window
    size(200,200);
}

void draw() {
    // Draw a white background
    background(255);

    // Set CENTER mode
    ellipseMode(CENTER);
    rectMode(CENTER);

    // Draw Zoog's body
    stroke(0);
    fill(150);
    rect(100,100,20,100);

    // Draw Zoog's head
    stroke(0);
    fill(255);
    ellipse(100,70,60,60);

    // Draw Zoog's eyes
    fill(0);
    ellipse(81,70,16,32);
    ellipse(119,70,16,32);

    // Draw Zoog's legs
    stroke(0);
    line(90,150,80,160);
    line(110,150,120,160);
}
```

**setup()** runs first one time. **size()** should always be first line of **setup()** since Processing will not be able to do anything before the window size is specified.

**draw()** loops continuously until you close the sketch window.

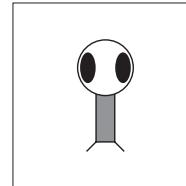


fig. 3.3

# VARIAZIONI CON IL MOUSE

- Le variabili `mouseX` e `mouseY` sono due variabili di sistema.
- Sono aggiornate automaticamente per rappresentare la posizione orizzontale e verticale del puntatore del mouse sullo sketch
- Le variabili `pmouseX` e `pmouseY` sono aggiornate con la posizione del mouse alla iterazione precedente

# VARIAZIONI CON IL MOUSE

## Example 3-2: *mouseX* and *mouseY*

```
void setup() {  
    size(200,200);  
}  
  
void draw() {  
    background(255);  
  
    // Body  
    stroke(0);  
    fill(175);  
    rectMode(CENTER);  
    rect(mouseX,mouseY,50,50);  
}
```

Try moving ***background()*** to ***setup()*** and see the difference! (Exercise 3-3)

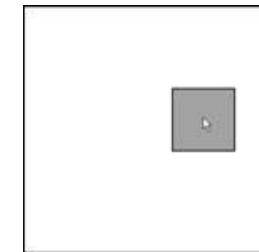


fig. 3.4

***mouseX*** is a keyword that the sketch replaces with the horizontal position of the mouse.

***mouseY*** is a keyword that the sketch replaces with the vertical position of the mouse.

Cosa accade se sposto l'istruzione ***background(255)*** in ***setup()*** ?

# ESERCIZIO: ZOOG INTERATTIVO

Disegna Zoog in modo da seguire la posizione del mouse

Example 3-3: Zoog as dynamic sketch with variation

```
void setup() {  
    size(200,200); // Set the size of the window  
    smooth();  
}  
  
void draw() {  
    background(255); // Draw a white background  
  
    // Set ellipses and rects to CENTER mode  
    ellipseMode(CENTER);  
    rectMode(CENTER);  
  
    // Draw Zoog's body  
    stroke(0);  
    fill(175);  
    rect(mouseX, mouseY, 20, 100);  
  
    // Draw Zoog's head  
    stroke(0);  
    fill(255);  
    ellipse(mouseX, mouseY-30, 60, 60);
```

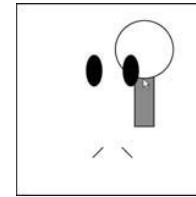


fig. 3.5

Zoog's body is drawn at the location (**mouseX, mouseY**).

Zoog's head is drawn above the body at the location (**mouseX, mouseY-30**).

# ESERCIZIO: ZOOG INTERATTIVO (2)



*Exercise 3-4: Complete Zoog so that the rest of its body moves with the mouse.*

```
// Draw Zoog's eyes  
fill(0);  
ellipse(_____, _____ , 16, 32);  
ellipse(_____, _____ , 16, 32);  
  
// Draw Zoog's legs  
stroke(0);  
line(_____, _____ , _____ , _____ );  
line(_____, _____ , _____ , _____ );
```

# ESERCIZIO: TRACCIA DEL MOUSE

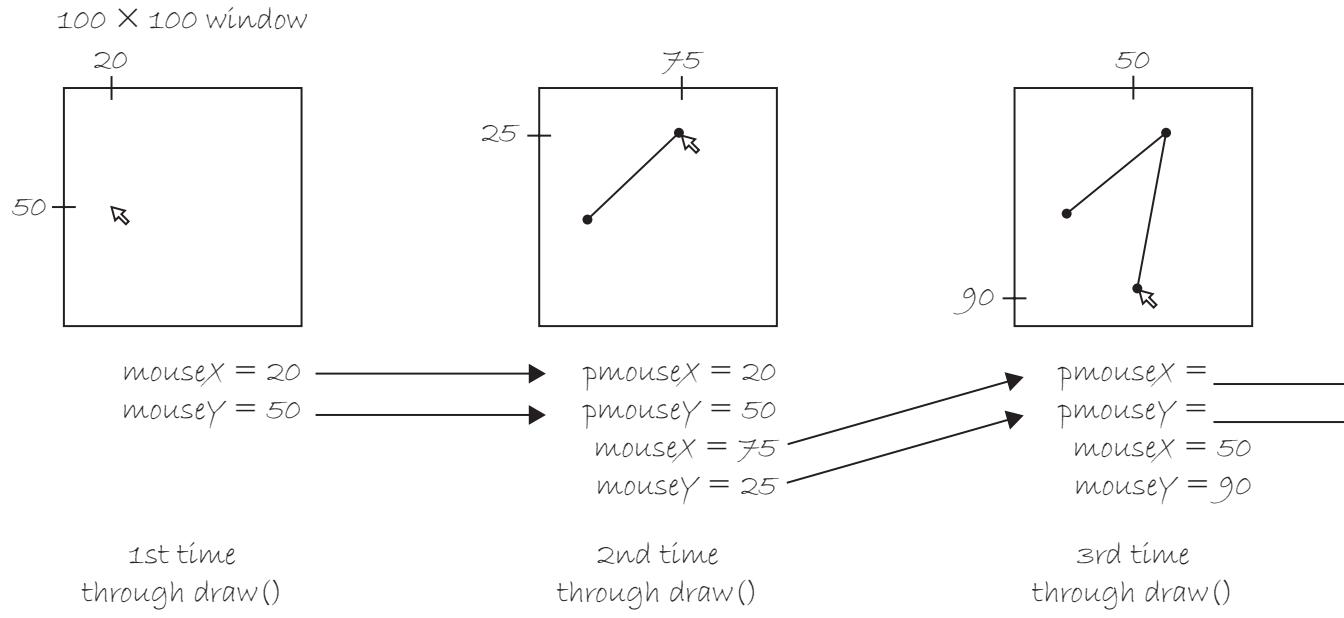


fig. 3.6

# ESERCIZIO: DISEGNO DI UNA LINEA CONTINUA

## Example 3-4: Drawing a continuous line

```
void setup() {  
    size(200,200);  
    background(255);  
    smooth();  
}  
  
void draw() {  
    stroke(0);  
    line(pmouseX,pmouseY,mouseX,mouseY);  
}
```

Draw a line from previous mouse location to current mouse location.

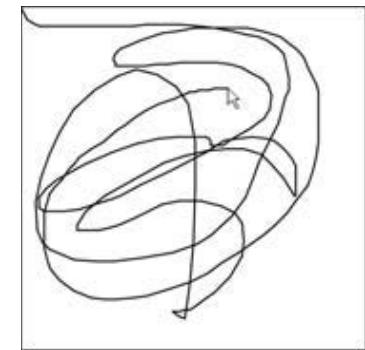


fig. 3.7

# **ESERCIZIO: LINEA CONTINUA CON TRATTO PROPORZIONALE ALLA VELOCITÀ**



# EVENTI: MOUSE E TASTIERA

- Il ciclo di esecuzione di `draw()` ci permette di rilevare la posizione del mouse tramite le variabili `mouseX` e `mouseY`
- Quando il bottone del mouse viene premuto viene generato un **evento**
- Quando si verifica un determinato evento, processing esegue una particolare funzione associata ad esso
- In risposta al click del mouse, viene chiamata la funzione `mousePressed()`
- In risposta alla pressione del tasto della tastiera viene chiamata la funzione `keyPressed()`

# ESERCIZIO: DISEGNO DI PUNTI AL CLICK DEL MOUSE

Example 3-5: *mousePressed()* and *keyPressed()*

```
void setup() {  
    size(200,200);  
    background(255);  
}  
  
void draw() {  
}  
  
void mousePressed() {  
    stroke(0);  
    fill(175);  
    rectMode(CENTER);  
    rect(mouseX,mouseY,16,16);  
}  
  
void keyPressed() {  
    background(255);  
}
```

Nothing happens in *draw()* in this example!

Whenever a user clicks the mouse the code written inside *mousePressed()* is executed.

Whenever a user presses a key the code written inside *keyPressed()* is executed.

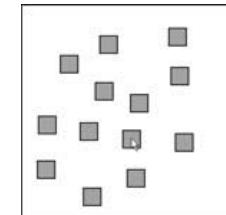


fig. 3.8

# ESERCIZIO: ZOOG INTERATTIVO FINALE

- Zoog segue il mouse
- Il colore degli occhi dipendono dalla posizione del mouse
- Quando si clicca con il mouse viene visualizzato il messaggio “Take me to your leader!”

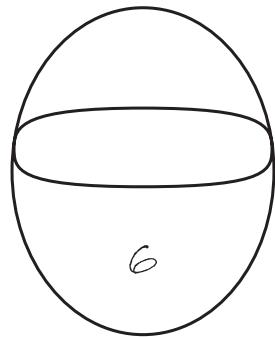
# ASSEGNAMENTO (1)

- Step 1: crea una visualizzazione semplice utilizzando delle primitive di base
- Step 2: rendi la composizione interattiva in base alla posizione del mouse

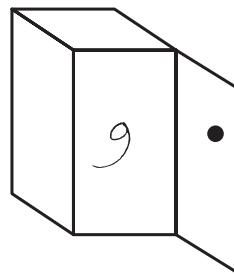


# VARIABILI

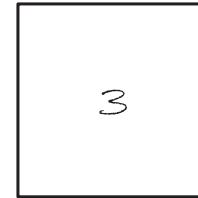
# COSA È UNA VARIABILE?



variable  
bucket



variable  
locker



variable  
post-it

*fig. 4.1*

Un nome per indirizzare una locazione della memoria del computer

# USO DI VARIABILI

Jane's Score	Billy's Score
5	10
30	25
53	47
66	68
87	91
101	98

fig. 4.3

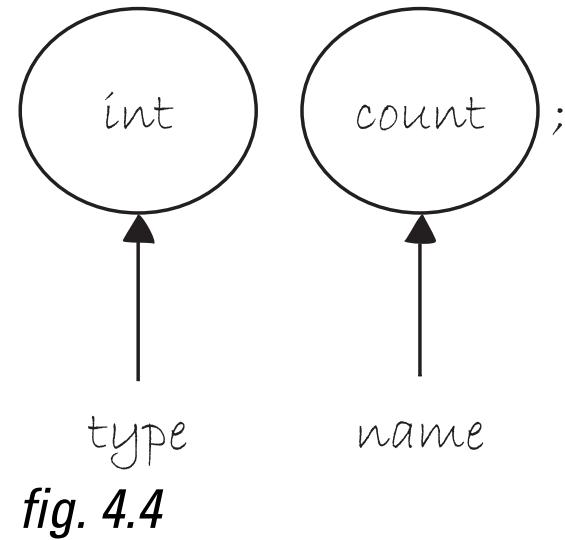
- Oltre a memorizzare un valore, una variabile permette di seguire il cambiamento del suo contenuto

# ESERCIZIO

- Definire le variabili per il gioco  
Pong

# DICHIARAZIONE DI VARIABILI

- Una variabile può memorizzare valori primitivi o riferimenti ad array e oggetti
- Una variabile deve essere dichiarata prima di essere usata
- Per dichiararla, bisogna specificare
  - il tipo: quale dato dovrà contenere
  - Il nome: come chiamiamo la locazione che conserva il dato



# TIPI DI VARIABILE

- boolean: true or false
- char: un carattere , 'a', 'b', 'c', etc.
- byte: un numero piccolo, -128 to 127
- short: un numero un poco più grande, -32768 to 32767
- int: un numero intero, -2147483648 to 2147483647
- long: un numero estremamente grande
- float: numero con cifre decimali, such as 3.14159
- double: numero con tante cifre decimali

# ASSEGNAMENTO DI UN VALORE

```
int count;  
count = 50;
```

Declare and initialize a variable in two lines of code.

```
int count = 50;
```

Declare and initialize a variable in one lines of code.

# ESEMPIO

## Example 4-1: Variable declaration and initialization examples

---

```
int count = 0;           // Declare an int named count, assigned the value 0
char letter = 'a';       // Declare a char named letter, assigned the value 'a'
double d = 132.32;       // Declare a double named d, assigned the value 132.32
boolean happy = false;   // Declare a boolean named happy, assigned the value false
float x = 4.0;           // Declare a float named x, assigned the value 4.0
float y;                 // Declare a float named y (no assignment)
y = x + 5.2;             // Assign the value of x plus 5.2 to the previously declared y
float z = x*y + 15.0;    // Declare a variable named z, assign it the value which
                        // is x times y plus 15.0.
```

# USO DI VARIABILI

```
// tra poco aggiungeremo le nostre  
variabili  
  
void setup(){  
    size(200,200);  
}  
  
void draw(){  
    background(255);  
    stroke(0);  
    fill(175);  
    ellipse(100,100,50,50);  
}
```

# USO DI VARIABILI

```
// tra poco aggiungeremo le nostre  
variabili  
  
void setup(){  
    size(200,200);  
}  
  
void draw(){  
    background(255);  
    stroke(0);  
    fill(175);  
    ellipse(mouseX,mouseY,50,50);  
}
```

# USO DI VARIABILI

```
// dichiariamo le nostre variabili
int circleX = 100;
int circleY = 100;

void setup(){
    size(200,200);
}

void draw(){
    background(255);
    stroke(0);
    fill(175);
    ellipse(circleX,circleY,50,50);
}
```

# ASSEGNAMENTO DI UN NUOVO VALORE

```
// dichiariamo le nostre variabili
int circleX = 0;
int circleY = 100;

void setup(){
    size(200,200);
}

void draw(){
    background(255);
    stroke(0);
    fill(175);
    ellipse(circleX,circleY,50,50);

    circleX = circleX + 1;
}
```

1. Remember `circleX = 0` and `circleY = 100`

2. Run `setup()`. Open a window  $200 \times 200$  →



3. Run `draw()`.

- Draw circle at  $(circleX, circleY) \rightarrow (0, 100)$



$$circleX = 0 + 1 = 1$$

4. Run `draw()`

- Draw circle at  $(circleX, circleY) \rightarrow (1, 100)$



- Add one to `circleX`

$$circleX = 1 + 1 = 2$$

5. Run `draw()`

- Draw circle at  $(circleX, circleY) \rightarrow (2, 100)$



- Add one to `circleX`

$$circleX = 2 + 1 = 3$$

6. And so on and so forth!

fig. 4.5

# ESERCIZIO

- Modifica l'esempio precedente per fare in modo che il cerchio cresca di dimensione

# ESERCIZIO

## *Exercise 4-4*

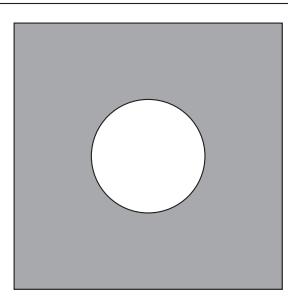
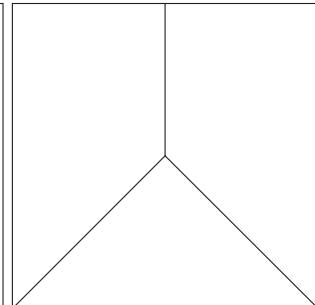
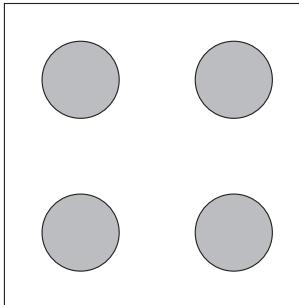


**Step 1:** Write code that draws the following screenshots with hard-coded values. (Feel free to use colors instead of grayscale.)

**Step 2:** Replace all of the hard-coded numbers with variables.

**Step 3:** Write assignment operations in `draw()` that change the value of the variables.

For example, “`variable1 = variable1 + 2;`”. Try different expressions and see what happens!



# VARIABILI DI SISTEMA

- Processing crea delle variabili prima dell'inizio della esecuzione
- Abbiamo già visto un esempio con mouseX e mouseY
- Ecco un elenco di variabili utili:
  - width: la larghezza della finestra
  - height: la altezza della finestra
  - frameCount: il numero di refresh della finestra dall'inizio dell'esecuzione
  - frameRate: numero di frame disegnati al secondo
  - screen.width: larghezza dello schermo
  - screen.height: altezza dello schermo
  - key: codice dell'ultimo tasto premuto
  - keyPressed: valore boolean, true se viene premuto un tasto

# ESEMPIO

## Example 4-5: Using system variables

```
void setup() {  
    size(200,200);  
    frameRate(30);  
}  
  
void draw() {  
    background(100);  
    stroke(255);  
    fill(frameCount/2);  frameCount is used to color a rectangle.  
    rectMode(CENTER);  
    rect(width/2,height/2,mouseX+10,mouseY+10);  
}  
  
void keyPressed() {  
    println(key);  
}
```

The rectangle will always be in the middle of the window if it is located at (width/2, height/2).

# NUMERI RANDOM

- La funzione random() è una funzione speciale che ritorna un valore casuale
- Rispetto alle altre funzioni viste finora (ellipse, line, point), questa risponde con un valore numerico
- La funzione ha bisogno di due numeri: viene ritornato un numero compreso tra i due
- La funzione ritorna un numero con la virgola: float
  - `float w = random(1,00);  
rect(100,100,w,50);`

# RANDOM ELLIPSES

Example 4-7: Filling variables with random values

```
float r;
float g;
float b;
float a;

float diam;
float x;
float y;

void setup() {
    size(200,200);
    background(0);
    smooth();
}

void draw() {
    // Fill all variables with random values
    r = random(255);
    g = random(255);
    b = random(255);
    a = random(255);
    diam = random(20);
    x = random(width);
    Y = random(height);

    // Use values to draw an ellipse
    noStroke();
    fill(r,g,b,a);
    ellipse(x,y,diam,diam);
}
```

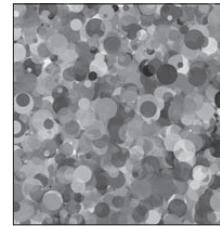


fig. 4.8

Each time through **draw()**, new random numbers are picked for a new ellipse.

# VARIABLE ZOOG





# **ISTRUZIONI CONDIZIONALI**

# VALORI BOOLEANI

- Il tipo boolean prevede solo due possibili valori: true o false
- Sono prodotti come risultato di operazioni di confronto

```
println(3 > 2)
```

```
// → true
```

```
println(3 < 2)
```

```
// → false
```

- Si possono confrontare anche le stringhe

```
println("Aardvark" < "Zoroaster")
```

```
// → true
```

- Il confronto tra stringhe segue l'ordinamento dato dalla codifica Unicode, che assegna un valore numerico ad ogni carattere

# BOOLEANI: OPERATORI DI CONFRONTO

- < (minore)
- > (maggiore)
- <= (minore o uguale)
- >= (maggiore o guale)
- == (uguale)
- != diverso
- Esempio

```
println("Itchy" != "Scratchy")  
// → true
```

# BOOLEANI: OPERATORI LOGICI (1)

- Java supporta tre operatori logici: `&&` (and), `||` (or) e `!` (not)
- I tre operatori sono alla base della Algebra di Boole
  - Permette di usare la logica proposizionale per ragionare su fatti che possano verificarsi o meno
- L'operatore `&&` restituisce `true` se entrambi i suoi argomenti sono `true`

```
println(true && false)
```

// → `false`

```
println(true && true)
```

// → `true`

```
println(false&& false)
```

// → `false`

```
println(false&& true)
```

// → `false`

# BOOLEANI: OPERATORI LOGICI (2)

- L'operatore `||` restituisce `true` se uno dei suoi argomenti è `true`

```
println(false || true)
```

// → true

```
println(false || false)
```

// → false

```
println(true || true)
```

// → true

```
println(true || false)
```

// → true

# BOOLEANI: OPERATORI LOGICI (3)

- L'operatore ! restituisce il valore inverso del suo argomento

```
println(!false)
```

// → true

```
println(!true)
```

// → false

- L'operatore condizionale (cond? val1 : val2) è un operatore ternario. Se il valore a sinistra del simbolo ? è vero ritorna il primo valore, altrimenti il secondo

```
println(true ? 1 : 2);
```

// → 1

```
println(false ? 1 : 2);
```

// → 2

# CONTROLLO DEL FLUSSO

- Quando un programma è formato da più istruzioni, queste vengono eseguite in ordine dall'alto verso il basso
- Il seguente programma è composto da due istruzioni



```
int theNumber = int(random(10, 100));  
println("Your number is " + theNumber);
```

# ISTRUZIONI CONDIZIONALI

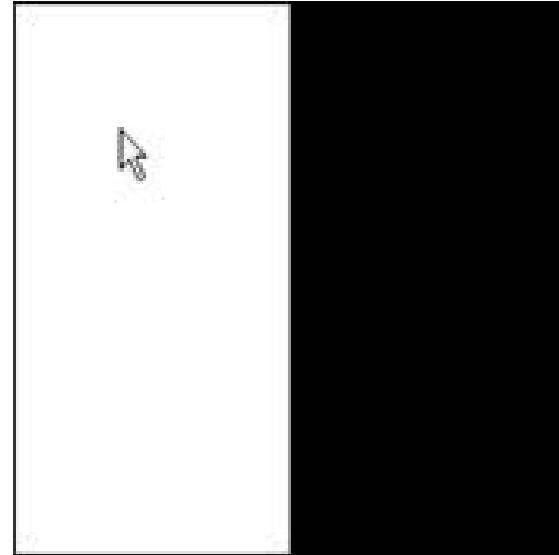
- Tutti i programmi vengono eseguiti in sequenza
- Queste istruzioni di controllo consentono di scegliere due possibili percorsi, sulla base di un valore booleano
- Sintassi:

```
if (espressione)
    istruzione1
else
    istruzione2
```
- Espressione è una espressione booleana
- Istruzione1 rappresenta il ramo eseguito se la valutazione ritorna true
- Istruzione2 rappresenta il ramo eseguito se la valutazione ritorna false



# ESEMPIO

```
if(mouseX < width / 2){  
    fill(255);  
    rect(0,0,wdth/2,height);  
}
```



*fig. 5.1*

# ESEMPIO

```
if(mouseX < width/2){  
    background(255);  
}else{  
    background(0);  
}
```

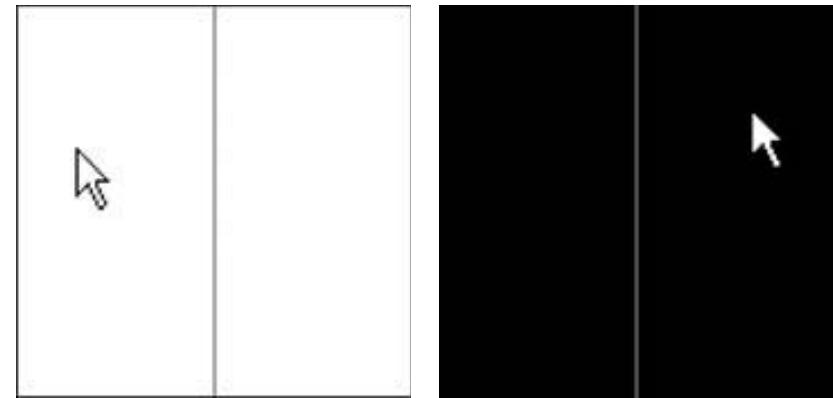


fig. 5.2

# CONDIZIONI MULTIPLE

```
if (boolean expression #1) {  
// code to execute if boolean  
expression #1 is true  
} else if (boolean expression #2) {  
// code to execute if boolean  
expression #2 is true  
} else if (boolean expression #n) {  
// code to execute if boolean  
expression #n is true  
} else {  
// code to execute if none of the  
above  
// boolean expressions are true  
}
```

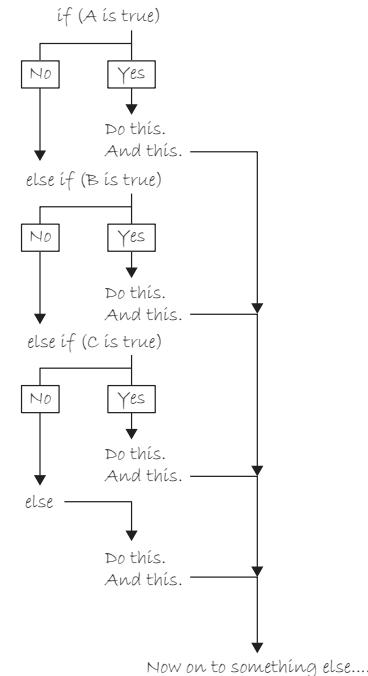


fig. 5.3

# ESEMPIO

```
if (mouseX < width/3) {  
    background(255);  
} else if (mouseX < 2*width/3) {  
    background(127);  
} else {  
    background(0);  
}
```

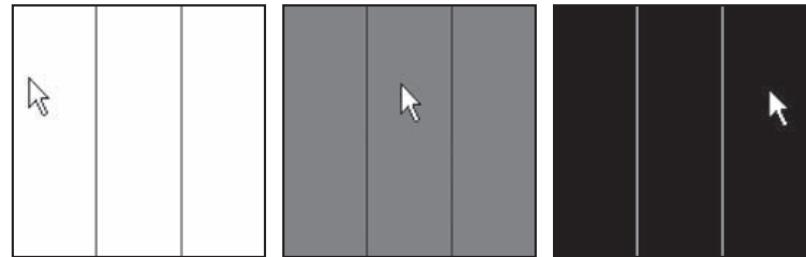


fig. 5.4

# SKETCH CONDIZIONALI

## Example 5-1: Conditionals

```
float r = 150;  
float g = 0;  
float b = 0;
```

1. Variables.

```
void setup() {  
    size(200, 200);  
}
```

```
void draw() {  
    background(r,g,b);  
    stroke(255);  
    line(width/2,0,width/2,height);
```

2. Draw stuff.

```
if (mouseX > width/2) {  
    r = r + 1;  
} else {  
    r = r - 1;  
}
```

3. "If the mouse is on the right side of the screen" is equivalent to "if **mouseX** is greater than width divided by 2."

```
if (r > 255) {  
    r = 255;  
} else if (r < 0) {  
    r = 0;  
}
```

4. If *r* is greater than 255, set it to 255.  
If *r* is less than 0, set it to 0.

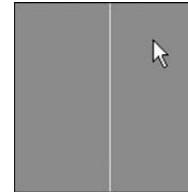


fig. 5.5

# FUNZIONE constrain()

```
if (r > 255) {  
    r = 255;  
} else if (r < 0) {  
    r = 0;  
}  
  
r = constrain(r, 0, 255);
```

Constrain with an “if” statement.

Constrain with the ***constrain()*** function.





# **VALORI, TIPI E OPERATORI**

# RAPPRESENTAZIONE DELL'INFORMAZIONE

- Il calcolatore può leggere, modificare e gestire solo **dati**
- I dati sono rappresentati come **sequenze di bit**, ovvero sequenze di 1 e 0
- I valori 1 e 0 sono rappresentati dal calcolatore come tensioni elettriche, rispettivamente alta e bassa
- Rappresentazione di numeri basata su base 2 invece che su base 10
- Ad esempio, il numero 13:
  - 0 0 0 0 1 1 0 1
  - 128 64 32 16 8 4 2 1



# VALORI E TIPI

- In un calcolatore ci sono sequenze lunghissime di bit
- Sequenze di bit sono organizzate in segmenti logici, chiamati valori
- Diversi valori possono avere diversi ruoli
  - Numeri, stringhe, boolean, oggetti, funzioni, e undefined
- Ogni valore viene creato nel momento in cui viene invocato
- Ogni valore viene conservato nella memoria interna del calcolatore, fino a quando c'è spazio disponibile
- I valori che non vengono utilizzati vengono distrutti e il loro spazio può essere riutilizzato

# NUMERI

- I numeri sono rappresentati da sequenze di cifre
  - Ad esempio il numero 13
- Nel momento in cui viene incontrato il numero 13 in un programma, viene creato uno spazio in memoria in cui sono conservati i bit che lo rappresentano
- Ogni numero in Java viene rappresentato con 64bit
  - Possiamo rappresentare  $2^{64}$  valori differenti per singolo valore
  - Dobbiamo rappresentare anche il segno (negativo o positivo) e la posizione della cifra decimale
  - Operazioni in virgola mobile perdono precisione quando rappresentati su uno spazio finito

# NUMERI: NOTAZIONE

- Numero intero
  - 13
- Numero decimale
  - 9.81
- Notazione scientifica
  - $2.998\text{e}8 = 2.998 * 10^8 = 299,800,000$
- I numeri decimali vanno sempre trattati come approssimazioni del valore reale
  - Ad esempio: confronto del risultato di una divisione

# OPERATORI ARITMETICI

- Gli operatori aritmetici prendono in input due numeri e producono in output un numero
- Gli operatori sono: \*, /, %, +, -
- Gli operatori possono esser concatenati
  - $100 + 4 * 11$
- I simboli + e \* sono operatori
- Gli operatori hanno una precedenza. In questo caso l'operatore di moltiplicazione viene eseguito prima dell'operatore +
- La lista di operatori precedenti è in ordine di precedenza decrescente
- Si può cambiare l'ordine di precedenza utilizzando le parentesi
  - $(100 + 4) * 11$

# OPERATORI ARITMETICI: MODULO

- L'operatore di modulo (%) restituisce il resto della divisione intera di due numeri
- $X \% Y$  è il resto dell'operazione di divisione intera di  $X$  per  $Y$
- Ad esempio:
  - $314 \% 100 = 14$
  - $144 \% 12 = 0$

# STRINGHE

- Le stringhe sono utilizzate per rappresentare testi
- Sono definite tramite l'uso di virgolette singole o doppie
  - "Patch my boat with chewing gum"
  - 'Monkeys wave goodbye'
- Java riconosce qualunque carattere all'interno delle virgolette
- Alcuni caratteri richiedono qualche attenzione
  - Ad esempio, rappresentare il carattere di virgolette dentro la stringa

# STRINGHE: ESCAPING

- Per includere caratteri speciali è necessario utilizzare un operatore di escaping: \
- Quando il carattere \ viene trovato all'interno di una stringa tra virgolette, esso indica che il carattere successivo ha un significato speciale
  - Un carattere con virgolette preceduto da \ non indica la fine della stringa ma una parte di essa
    - ‘Egli disse \’Ciao\’ appena lo vide’
  - Il carattere n preceduto da \ viene interpretato come un rimando a capo
    - "This is the first line\nAnd this is the second"
    - Contiene il seguente testo
    - This is the first line  
And this is the second
  - Per rappresentare il carattere \ dentro una stringa si usano due simboli \ in sequenza
    - "A newline character is written like \"\\n\"."

# STRINGHE: OPERATORI

- Le stringhe non prevedono gli operatori di divisione, moltiplicazione o sottrazione.
- E' possibile utilizzare l'operatore **+**, con il significato di concatenazione
  - `println("con" + "cat" + "e" + "nate");`
  - // “concatenate”

# VALORI INDEFINITI

- Ci sono due valori speciali che vengono usati per denotare l'assenza di valori significativi
  - `null`
  - `undefined`

# PRECEDENZA EGLI OPERATORI LOGICI

- Gli operatori logici `&&` e `||` valutano o convertono in tipo booleano l'operando a sinistra per determinare il risultato del confronto
- In base al risultato di questa prima valutazione, il comportamento può variare
- Questa proprietà può essere utile per assegnare un valore di default in caso uno sia mancante

```
println(null || "user")
// → user
println("Karl" || "user")
// → Karl
```

- L'espressione a destra viene valutata solo se necessario

```
true || x
false && x
```

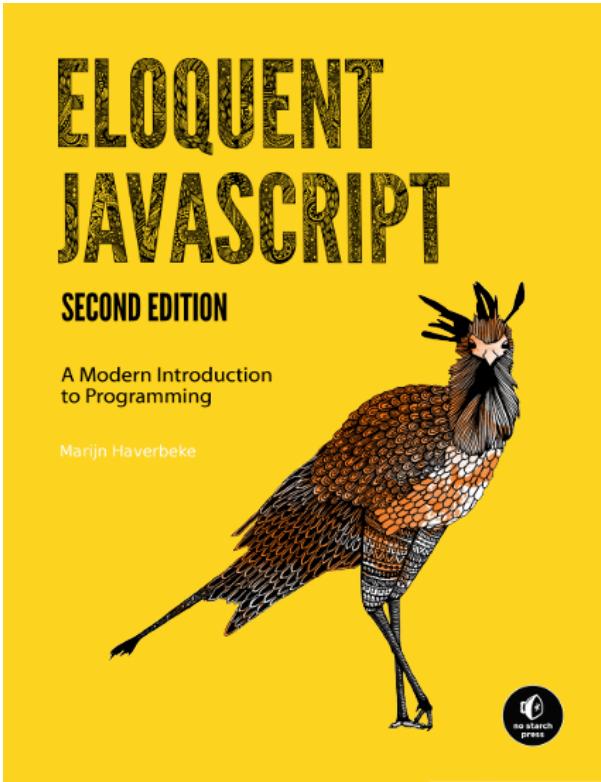
# SOMMARIO

- Linguaggio Java
- Definizione di valori e tipi
- Operatori aritmetici: +, -, \*, /, %
- Concatenazione di stringhe: +
- Confronto: ==, !=, ===, !==, <, >, <=, >=
- Operatori logici: &&, || , !
- Operatori unari: -, !
- Operatori ternari: ( ?: )



# **STRUTTURA DI UN PROGRAMMA JAVASCRIPT**

# LIBRI E RIFERIMENTI



- Capitolo 2

Eloquent Javascript – Second Edition  
Marijn Haverbeke  
Licensed under CC license.  
Available here: <http://eloquentjavascript.net/>

# ESPRESSIONI

- Una espressione è un frammento di codice che produce un valore
- Ogni valore di base è una espressione (es. 22, “programma”, 3.14)
- Una espressione tra parentesi è ancora una espressione
- Un operatore binario applicato a due espressioni produce una espressione
- Un operatore unario applicato ad una espressione produce una espressione
- Espressioni più semplici possono essere combinate per creare una espressione più complessa

# ISTRUZIONI (O STATEMENT)

- L'esempio più semplice di istruzione è una espressione seguita da ";"
  - 1;
  - !false;
- Una istruzione può essere fine a se stessa (vedi i due esempi qui sopra)
- Può ritornare un risultato sullo schermo (vedi istruzione console.log)
- Può cambiare lo stato interno del calcolatore in modo da influenzare le istruzioni che verranno dopo
- In molti casi è essenziale terminare una istruzione con il simbolo ";"
- Ci sono casi in cui non è necessaria. Per evitare problemi, tendiamo a inserirlo sempre

# VARIABILI

- Per mantenere lo stato interno si usano le **variabili**
- Le variabili permettono di gestire lo stato interno e di memorizzare valori
- Le variabili sono create attraverso la keyword (parola chiave) **var**

```
var caught = 5 * 5;
```

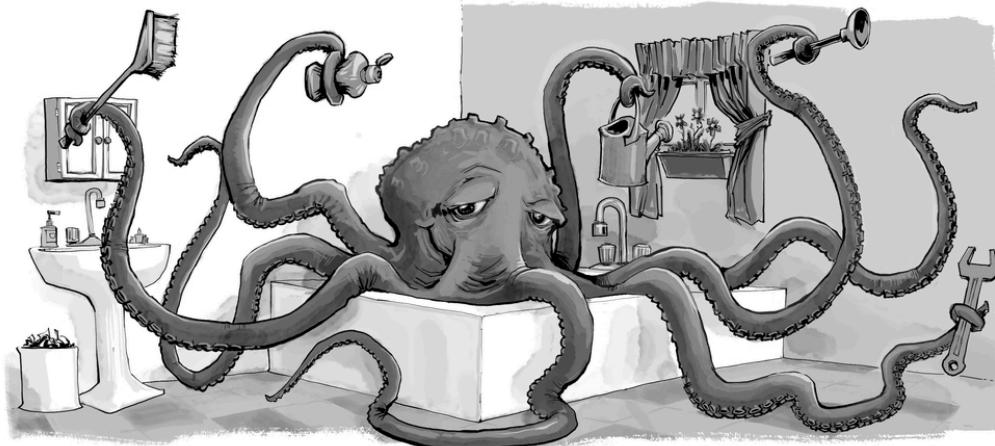
- L'istruzione precedente crea una variabile chiamata `caught` e la usa per contenere il risultato della moltiplicazione di 5 per 5
- Una volta che una variabile è stata definita, il suo nome può essere utilizzato all'interno delle espressioni successive

```
var ten = 10;  
console.log(ten * ten);  
// → 100
```

# NOMI DI VARIABILI

- Il nome di una variabile può essere qualunque parola
  - non può essere una parola chiave (es `var`)
  - non può contenere spazi
  - Si possono utilizzare numeri, ma il nome non può iniziare con un numero
  - Non si può utilizzare simboli di punteggiatura, ad eccezione di \$ e \_
- Una variabile non è legata per sempre ad un singolo valore
- L'operatore = può disconnettere una variabile dal suo valore attuale e assegnarne un altro

```
var mood = "light";
console.log(mood);
// → light
mood = "dark";
console.log(mood);
// → dark
```



# DEFINIZIONE MULTIPLA DI VARIABILI

- Una singola parola chiave var può essere utilizzata per definire più variabili
- Ogni definizione deve essere separata da virgola

```
var one = 1, two = 2;  
console.log(one + two);  
// → 3
```

# KEYWORDS E PAROLE RISERVATE

- Parole con un significato particolare, come le keywords, non possono essere utilizzate come nomi di variabili
- Altre parole sono riservate per uso futuro del linguaggio

break case catch class const continue debugger  
default delete do else enum export extends false  
finally for function if implements import in  
instanceof interface let new null package private  
protected public return static super switch this  
throw true try typeof var void while with yield

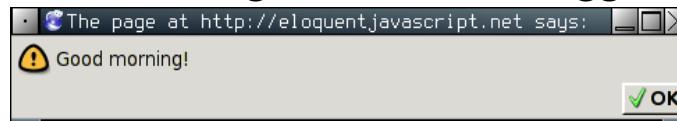
# ENVIRONMENT

- L'insieme delle variabili e i loro rispettivi valori ad un certo istante sono chiamate environment
- L'environment rappresenta lo stato interno del calcolatore
- Quando il programma viene avviato, l'environment non è vuoto ma contiene le variabili che sono parte dello standard del linguaggio
- Alcune di queste variabili permettono di interagire con il sistema esterno

# FUNZIONI

- Una funzione è un frammento di programma encapsulata in un valore
- Queste funzioni possono essere applicati per eseguire quella porzione di codice
- Nell'environment di default di ogni browser è presente la funzione `alert`, che permette di mostrare una finestra di dialogo con un messaggio

```
alert("Good morning!");
```

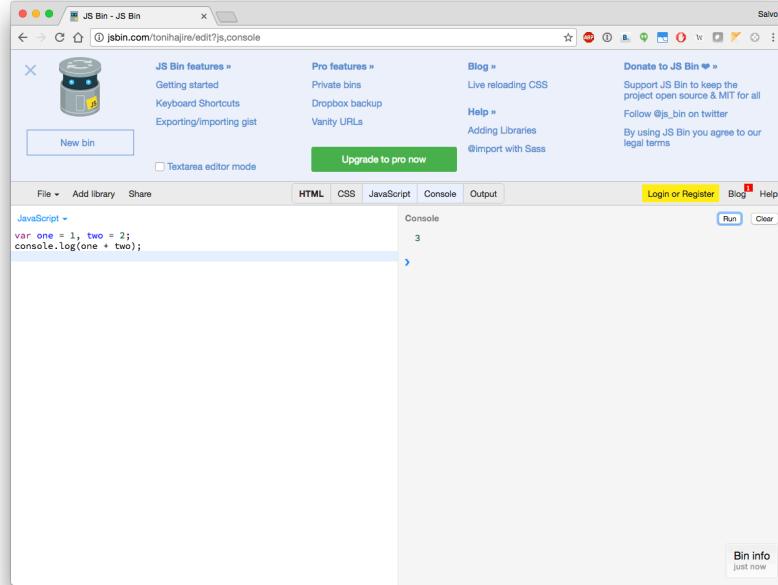


- L'esecuzione di una funzione è detta **invocazione** o **chiamata**
- Una funzione può essere chiamata scrivendo due parentesi alla fine del nome della variabile che la contiene
- Eventuali valori inclusi tra parentesi, chiamati **argomenti**, sono disponibili per il codice della funzione stessa

# LA FUNZIONE console.log

- Questa funzione è disponibile in molti browser moderni
- Permette di mostrare a video il valore di espressioni
- Nei browser, il testo prodotto dalla funzione è mostrato all'interno della **Javascript console**
- Di solito questa parte del browser è nascosta.
  - In molti sistemi si apre premendo il tasto **F12**
  - Nei sistemi basati su Mac Os, si apre con **Command-Option-I**
  - Altrimenti si può cercare nel menu del browser una voce del tipo “web console” o “developer tools”

# JSBIN



- Nella prima parte del corso utilizzeremo una interfaccia web per prendere familiarità con il linguaggio
- <http://jsbin.com/?js,console>

# VALORI DI RITORNO

- Le funzioni possono produrre degli effetti sullo schermo (come la funzione `console.log`)
- Altre invece non hanno effetti esterni visibili ma restituiscono un valore di ritorno
- Ad esempio la funzione `Math.max` prende due numeri e restituisce il maggiore tra i due
  - `console.log(Math.max(2, 4));`  
`// → 4`
- Una funzione che produce un valore è una espressione e, quindi, può essere utilizzata in un contesto più complesso
  - `console.log(Math.min(2, 4) + 100);`  
`// → 102`



# FUNZIONI prompt E confirm

- Queste due funzioni permettono di interagire con l'utente tramite finestre di dialogo
  - confirm permette una scelta tra “Ok” o “Cancel”: ritorna il valore true se l'utente preme OK. Altrimenti ritorna false
  - prompt può essere utilizzata per chiedere una domanda “aperta” all'utente. La funzione ritorna il valore inserito dall'utente
- Queste due funzioni non sono molto utilizzate nelle applicazioni più diffuse
- Possono essere utili in un contesto didattico

# ESERCIZIO

- Scrivere un programma che riceva in input la lunghezza del lato di un quadrato e calcoli il suo perimetro e la sua area

# ESERCIZIO: AREA E PERIMETRO DEL QUADRATO

```
var a = prompt("Dammi la lunghezza del lato");

var area = a * a;
var perimetro = a * 4;

console.log("L'area del rettangolo è " + area);
console.log("Il perimetro del rettangolo è " +
perimetro);
```

# ESERCIZIO

- Scrivere un programma che riceva in input la lunghezza dei lati di un rettangolo e calcoli il suo perimetro e la sua area

# ESERCIZIO: AREA E PERIMETRO DEL RETTANGOLO

```
var a = Number(prompt("Dammi la lunghezza del lato"));
var b = Number(prompt("Dammi la lunghezza del lato
minore"));

var area = a * b;
var perimetro = (a + b) * 2;

console.log("L'area del rettangolo è " + area);
console.log("Il perimetro del rettangolo è " +
perimetro);
```

# ESEMPIO ISTRUZIONE CONDIZIONALE if

```
var theNumber = Number(prompt("Pick a number", ""));  
if (!isNaN(theNumber))  
    alert("Your number is the square root of " +  
        theNumber * theNumber);
```

# ESEMPIO ISTRUZIONE CONDIZIONALE if-else

```
var theNumber = Number(prompt("Pick a number", ""));  
if (!isNaN(theNumber))  
    alert("Your number is the square root of " +  
          theNumber * theNumber);  
else  
    alert("Hey. Why didn't you give me a number?");
```

# ESEMPIO DI SERIE DI CONDIZIONALI

```
var num = Number(prompt("Pick a number", "0"));

if (num < 10)
    alert("Small");
else if (num < 100)
    alert("Medium");
else
    alert("Large");
```

# ESERCIZIO

- Scrivere un programma che prenda in input dall'utente due numeri e ritorni il massimo tra i due

# ESERCIZIO: MASSIMO TRA DUE NUMERI

```
var a = Number(prompt("Dammi il primo numero"), "");  
var b = Number(prompt("Dammi il secondo numero"), "");  
  
if(a > b)  
    console.log("Il massimo è " + a);  
else  
    console.log("Il massimo è " + b);
```

# ESERCIZIO

- Scrivere un programma che chieda in input tre valori che rappresentano le lunghezze di tre lati di un triangolo; decidere se il triangolo è equilatero, isoscele o scaleno e stampare il risultato sulla console

# ESERCIZIO: TIPO DI TRIANGOLO

```
var lato1 = Number(prompt("Lunghezza lato 1"));
var lato2 = Number(prompt("Lunghezza lato 2"));
var lato3 = Number(prompt("Lunghezza lato 3"));

console.log(lato1);
console.log(lato2);
console.log(lato3);

var tipoTriangolo = "scaleno";

// commentom: tutto quell oche mi pare
if(lato1 == lato2)
    tipoTriangolo = "isoscele";
else if(lato1 == lato3)
    tipoTriangolo = "isoscele";
else if(lato2 == lato3)
    tipoTriangolo = "isoscele";

if ((lato1==lato2) && (lato1==lato3))
    tipoTriangolo = "equilatero";

//if ((lato1 != lato2) && (lato1 != lato3) && (lato2 != lato3))
//    tipoTriangolo = "scaleno";

console.log("Il triangolo è " + tipoTriangolo);
```

# ITERAZIONI

- Le istruzioni iterative permettono di ripetere l'esecuzione di parti del programma una o più volte
- Il numero di ripetizioni può essere fissato (iterazione determinata)
- Oppure può dipendere da una condizione (iterazione indeterminata)

# ISTRUZIONE while

- L'espressione `while` è uno dei costrutti di iterazione di Javascript
- Sintassi
  - `while` (espressione)  
    istruzione
- espressione è una espressione booleana (detta **guardia**): se viene valutata a true allora viene eseguita l'istruzione interna (detta **corpo**) e si ripete il ciclo; se l'espressione booleana ritorna false, la ripetizione termina
- Se espressione è false fin dall'inizio, l'istruzione non viene mai eseguita

# ESEMPIO: LISTA DEI NUMERI PARI < 12

```
var number = 0;  
while (number <= 12) {  
    console.log(number);  
    number = number + 2;  
}  
// → 0  
// → 2  
// ... etcetera
```

# BLOCCHI DI ISTRUZIONI

- Nei costrutti che abbiamo visto finora, le espressioni booleane hanno controllato l'esecuzione di singole istruzioni
- Nel caso in cui debbano essere eseguite più istruzioni in sequenza possiamo definire un **blocco**, ovvero una sequenza di istruzioni rinchiusa tra parentesi graffe (`{` e `}`)
- Esempio: scrivere un programma che stampi i numeri pari minori o uguali a 12:

```
var number = 0;  
while (number <= 12) {  
    console.log(number);  
    number = number + 2;  
}  
// → 0  
// → 2  
// ... etcetera
```

# ESEMPIO

- Scrivere un programma che calcoli il valore di  $2^{10}$

```
var result = 1;  
var counter = 0;  
while (counter < 10) {  
    result = result * 2;  
    counter = counter + 1;  
}  
console.log(result);  
// → 1024
```

# ISTRUZIONE do-loop

- Il costrutto do-loop è simile all'istruzione while
- L'unica differenza è che il corpo viene eseguito prima di valutare l'espressione booleana a guardia del ciclo
- Quindi, anche se all'inizio del ciclo l'espressione è false, il corpo viene eseguito almeno una volta

```
do {  
    var yourName = prompt("Who are you?");  
} while (!yourName);  
console.log(yourName);
```

# ISTRUZIONE for

- Alcuni casi di cicli visti finora utilizzano una variabile di controllo per contare il numero di iterazioni eseguite
- Questo tipo di soluzione si presenta molto spesso in fase di programmazione
- Il linguaggio prevede un costrutto apposito per delle iterazioni determinate
- Sintassi:

```
for (expr1; expr2; expr3)
    istruzione
```
- Dove expr1 serve a inizializzare la variabile di controllo; expr2 è la guardia di fine ciclo; expr3 aggiorna la variabile di controllo; istruzione è il corpo del ciclo

# ESEMPIO

- Scrivere un programma che calcoli il valore di  $2^{10}$

```
var result = 1;  
for (var counter = 0; counter < 10; counter = counter + 1)  
    result = result * 2;  
console.log(result);  
// → 1024
```

# INTERRUZIONE DI UN CICLO

- Abbiamo visto che un ciclo viene interrotto quando l'espressione a guardia del ciclo diventa `false`
- E' possibile forzare l'interruzione di un ciclo con l'istruzione `break`
- Esempio: trovare il primo intero che sia maggiore di 20 e divisibile per 7

```
for (var current = 20; ; current++) {  
    if (current % 7 == 0)  
        break;  
}  
console.log(current);  
// → 21
```

- È possibile interrompere l'esecuzione del corpo del ciclo e saltare all'iterazione successiva con l'istruzione `continue`

# INDENTAZIONE DEL CODICE

- In molti esempi potete notare la presenza di uno o più spazi davanti ad alcune istruzioni
- Questi spazi servono ad indentare blocchi di codice
- Questa indentazione serve a enfatizzare la struttura del codice, evidenziando i blocchi contenuti all'interno delle istruzioni condizionali o ai cicli
- L'indentazione è solo una convenzione. Non è necessaria per la corretta esecuzione del programma

# AGGIORNAMENTO SUCCINTO DI VARIABILI

- Abbiamo visto diversi esempi di aggiornamento di varibili all'interno dei cicli
- Ad esempio alcune guardie avevano la seguente sintassi  
`counter = counter + 1`
- Per evitare di scrivere più volte la stessa variabile si può usare la seguente forma compatta  
`counter += 1`
- La stessa forma si può usare per altri operatori e operandi: \*=2, -=1
- Per incrementi o decrementi di singole unità si possono anche usare le espressioni `counter++` e `counter--`

# ISTRUZIONE SWITCH

- In alcuni casi è necessario confrontare una variabile con diversi possibili valori

```
if (variable == "value1") action1();
else if (variable == "value2") action2();
else if (variable == "value3") action3();
else defaultAction();
```

- Per semplificare questo tipo di confronti si può utilizzare il costrutto switch

```
switch (prompt("What is the weather like?")) {
  case "rainy":
    console.log("Remember to bring an umbrella.");
    break;
  case "sunny":
    console.log("Dress lightly.");
  case "cloudy":
    console.log("Go outside.");
    break;
  default:
    console.log("Unknown weather type!");
    break;
}
```

- A volte questo tipo di istruzione può generare errori. Una catena di istruzioni if può fornire una soluzione più facile da gestire

# CAPITALIZATION

- I nomi delle variabili non possono contenere spazi
- In alcuni casi è utile chiamare una variabile con un nome composto
- Ci sono alcune convenzioni per risolvere questo problema

fuzzylittleturtle

fuzzy\_little\_turtle

FuzzyLittleTurtle

fuzzyLittleTurtle

# COMMENTI AL CODICE

- Sebbene il linguaggio sia di alto livello, spesso il solo codice non è sufficiente per rendere un programma comprensibile ad un essere umano
- Spesso si utilizzano delle annotazioni e commenti al codice per spiegare alcune parti di esso
- Il **commento** è una parte del programma ma viene completamente ignorato dall'interprete che lo esegue
- Si possono scrivere due tipi di commenti: su singola riga o a blocchi
- I commenti su singola riga iniziano con due caratteri // e finiscono con la fine della riga

```
var accountBalance = calculateBalance(account);
// It's a green hollow where a river sings
accountBalance.adjust();
// Madly catching white tatters in the grass.
```

# COMMENTI

- Per definire un commento su più righe, si racchiude il testo da commentare tra i due simboli /\* e \*/

```
/*
```

```
I first found this number scrawled on the back of one of  
my notebooks a few years ago. Since then, it has often  
dropped by, showing up in phone numbers and the serial  
numbers of products that I've bought. It obviously likes  
me, so I've decided to keep it.
```

```
*/
```

```
var myNumber = 11213;
```

# SOMMARIO

- Espressioni e istruzioni (o statement)
- Istruzioni condizionali: if, else, switch
- Istruzioni iterative: while, do, for
- Funzioni speciali fornite dall'environment



# **ESERCIZI**

# ESERCIZIO: TRIANGOLO

- Scrivere un programma che stampi un triangolo come il seguente

```
#  
##  
###  
####  
#####  
######  
######
```

# ESERCIZIO: FIZZBUZZ

- Scrivere un programma che stampi tutti i numeri da 1 a 100, con due eccezioni. Invece dei numeri divisibili per 3 deve stampare la scritta “Fizz”. Invece dei numeri divisibili per 5 deve stampare la scritta “Buzz”.
- Modificare il programma precedente, stampando “FizzBuzz” per i numeri che sono divisibili sia per 3 che per 5

# ESERCIZIO: SCACCHIERA

- Scrivere un programma che disegni una scacchiera 8x8 come quella riportata di seguito

```
# # # #
# # # #
# # # #
# # # #
# # # #
# # # #
# # # #
# # # #
```

# ESERCIZIO: SOMMA DI N NUMERI

- Scrivere un programma che legga in input un numero **n** e poi chieda una sequenza di **n** numeri interi. Il programma deve stampare la somma degli **n** numeri

# ESERCIZIO: SOMMA, MASSIMO E MINIMO DI N NUMERI

- Scrivere un programma che legga in input un numero **n** e poi chieda una sequenza di **n** numeri interi. Il programma deve stampare la somma degli **n** numeri, il massimo e il minimo

# ESERCIZIO: NUMERI PRIMI

- Scrivere un programma che legga da input un numero intero **n** e stabilisca se **n** è un numero primo