tree
an Opinno company

ManpowerGroup™

Introduzione a C#

Iniziare a programmare in C# su Unity

Iniziamo da un paio di concetti base.

Cos'è un algoritmo?
Un algoritmo è un *elenco di passaggi per svolgere un compito*, ad esempio la ricetta di un dolce è un algoritmo.

Cosa vuol dire programmare?
Programmare vuol dire *tradurre un algoritmo* in un linguaggio che una macchina possa capire. Ciò avviene sfruttando i linguaggi di programmazione.

In Unity utilizzeremo il linguaggio di programmazione C# per programmare i vari algoritmi che serviranno a far girare il gioco.

Iniziare a programmare in C# su Unity

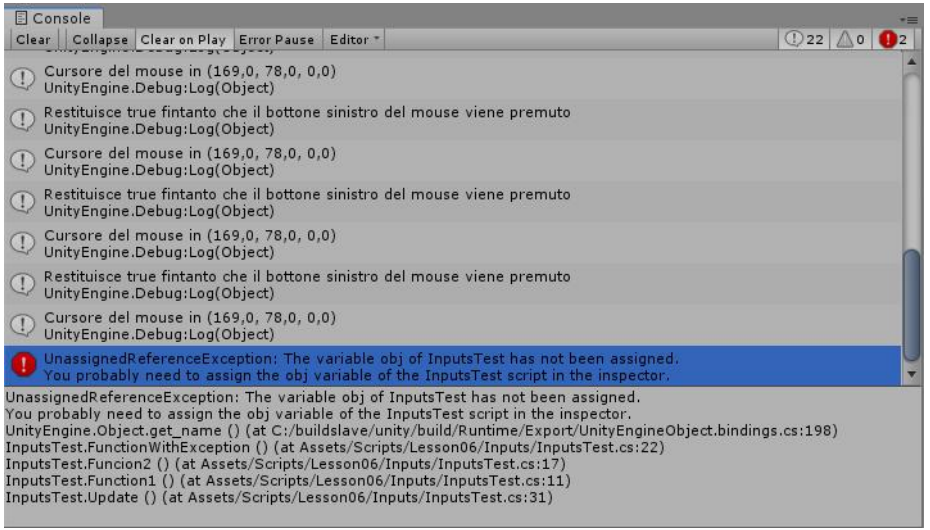
Prima di passare a programmare complesse meccaniche di gioco, programmeremo delle piccole cose in modo tale da acquisire dimestichezza con i nostri strumenti.

I risultati di queste cose verranno visualizzati sulla console di Unity.

La console

Una finestra di Unity che ci comunica errori e avvertimenti e sulla quale possiamo scrivere tramite codice. Si consiglia di aggiungerla/posizionarla sempre in un punto visibile del layout.

- Messaggi di debug, warning ed errore
- Call stack al momento dell'errore



Possiamo mostrare/nascondere i tre tipi di messaggi (normale, warning ed error), accorpare i messaggi identici ed altre cose.

Creare un nuovo Script

Per iniziare a programmare abbiamo bisogno di un file in cui scrivere il nostro codice.
Possiamo creare uno Script (file di tipo .cs)
cliccando col tasto destro sulla finestra Project e selezionando *Create* > *C# Script*
All'interno di uno Script potremo scrivere molte cose:

- nuove componenti di Unity da aggiungere ai GameObject
- nuovi tipi di assets
- architetture dati da utilizzare durante il gameplay

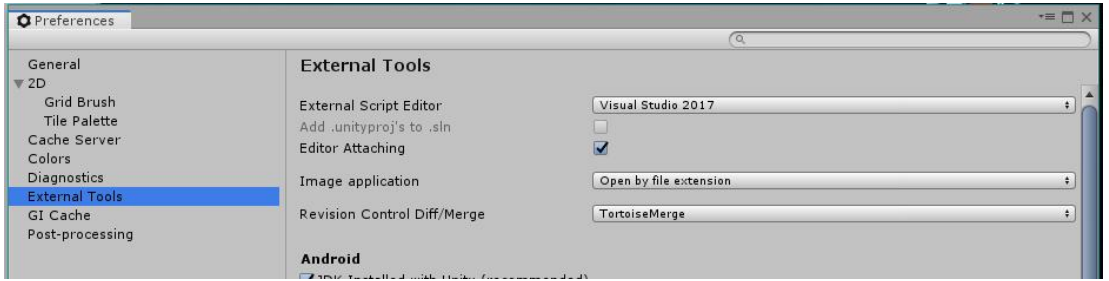
NB: nei primi due casi possiamo scriverne uno solo per file. Ad ogni modo è *quasi* sempre consigliabile creare un file nuovo quando si crea un codice nuovo.
Negli esempi che vedremo il file creato è stato nominato NuovoScript
NB: i nomi degli script NON possono iniziare con numeri né contenere spazi o caratteri speciali eccetto l'underscore

Creare un nuovo Script

Una volta creato il file basterà fare doppio click su di esso e Unity lo aprirà automaticamente in Visual Studio (che dovrebbe esser installato con Unity) collegandolo direttamente al progetto (sarà possibile vedere gli altri file del progetto dalla finestra Solution Explorer di Visual Studio).

Se ciò non dovesse avvenire, su Unity controllate in *Edit* > *Preferences* nella sezione **External Tools** se Visual Studio è impostato come **External Script Editor**.

Se compare altro, usate il menù di ricerca e assegnategli l'eseguibile devenv.exe presente nella cartella di installazione di Visual Studio.



Se ancora non funziona correttamente, su Visual Studio aprire *Strumenti* > *Ottieni Strumenti e Funzionalità* ed assicuratevi che sia installato il pacchetto **Sviluppo di giochi con Unity**.

Apriamo un file appena creato

Unity genera automaticamente un file con una porzione di codice già scritta.

Questi file sono già predisposti per essere delle **componenti** (MonoBehaviour).

Possiamo già trascinare il file dalla finestra Project in uno dei nostri GameObject in scena (o aggiungerlo tramite il menù *Add Component* in fondo all'**Inspector**).

Così facendo la nostra nuova componente comparirà nell'**Inspector** del GameObject.

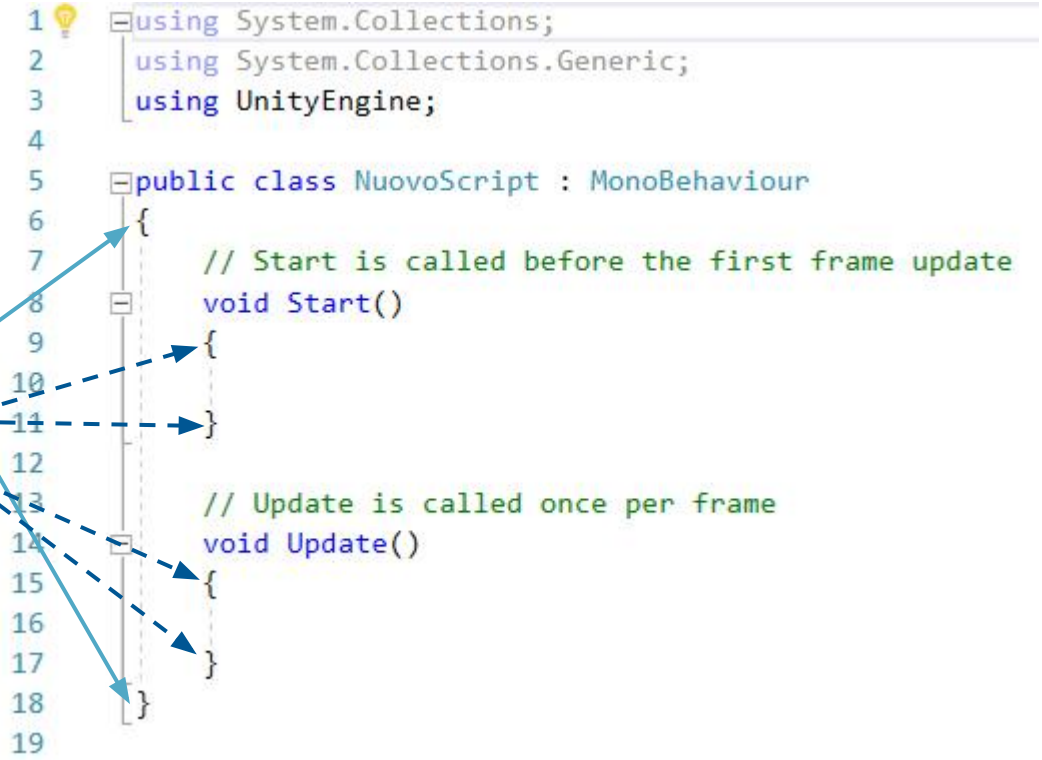
```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class NuovoScript : MonoBehaviour
6  {
7      // Start is called before the first frame update
8      void Start()
9      {
10
11     }
12
13     // Update is called once per frame
14     void Update()
15     {
16
17     }
18 }
19
```

Le parentesi graffe

Le parentesi graffe delimitano una porzione di codice.

Il codice di questo file è composto da tre porzioni:

la classe NuovoScript,
la funzione Start()
e la funzione Update()

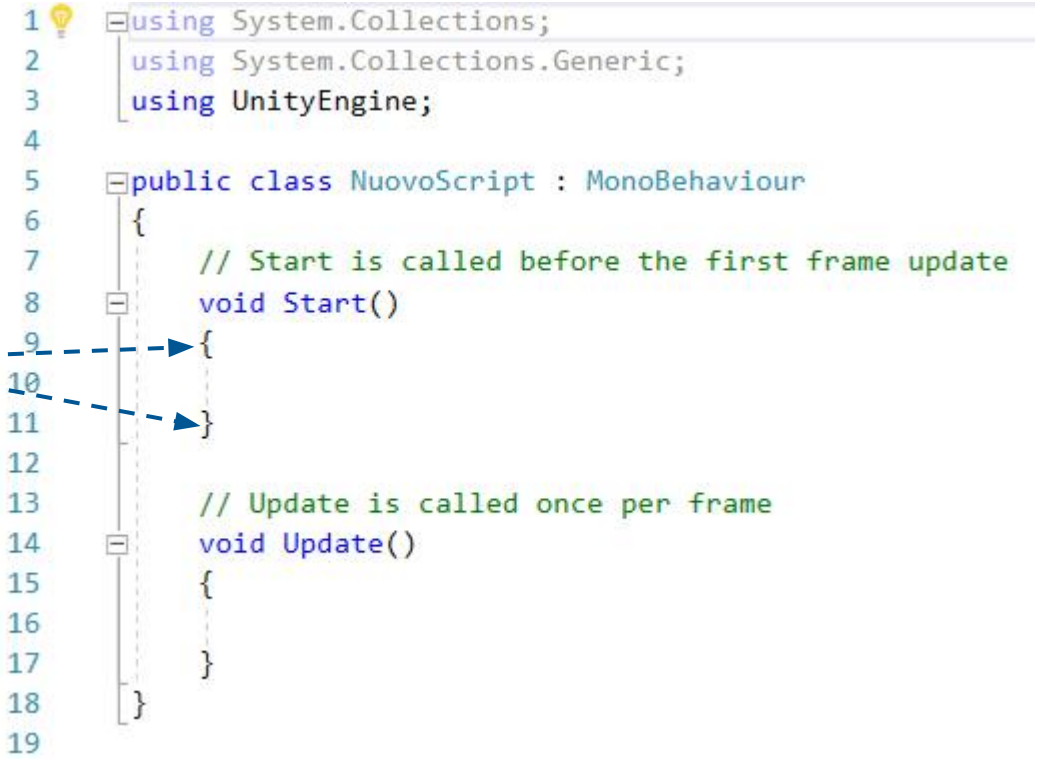


NuovoScript racchiude Start() e Update() poiché sono scritte all'interno delle parentesi graffe che si aprono subito dopo la riga
public class NuovoScript
e si chiudono solo alla riga 19

Le parentesi graffe

Ad esempio se volessimo far eseguire qualcosa alla porzione di codice Start() dovremo scrivere qualcosa all'interno delle parentesi graffe (che in questo momento si aprono e si chiudono alle righe 9 e 11).

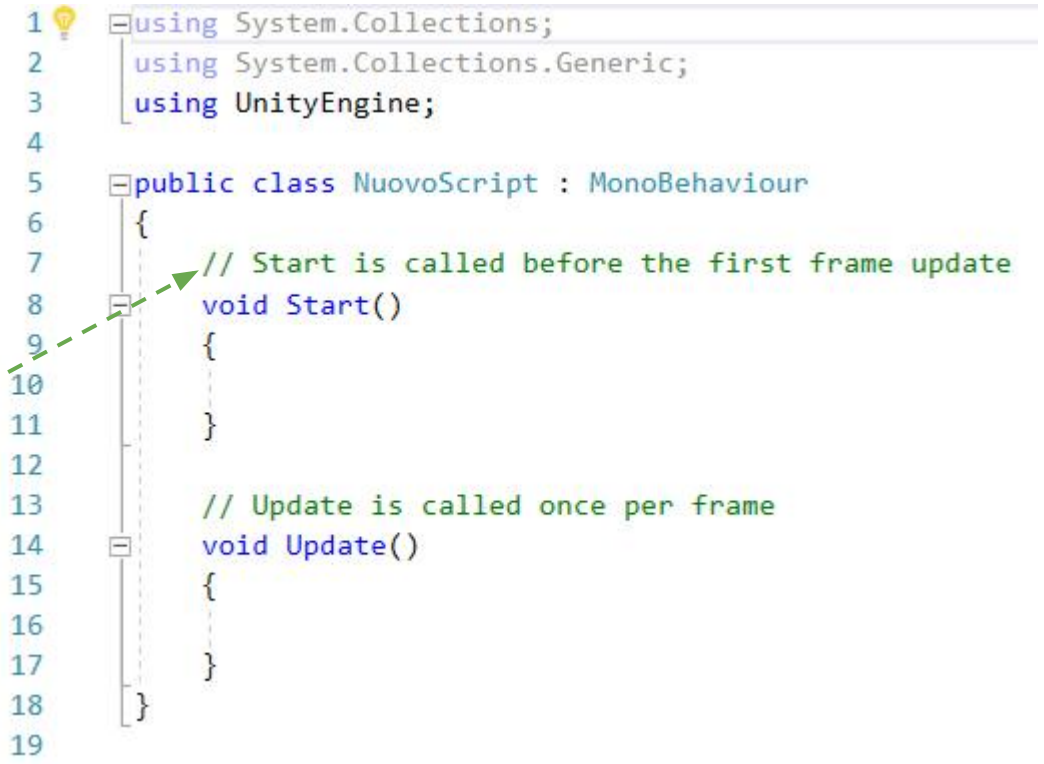
NB: ogni volta che si aprono delle parentesi graffe il codice viene indentato (si sposta di "*un tab*" a destra). Visual Studio esegue questa cosa in automatico quando andate a capo dopo una parentesi graffa aperta.
Per farlo manualmente premete il tasto **tab** (quello a sinistra della **Q** sulla tastiera)



I commenti

Le scritte “in verde” all’interno di Visual Studio rappresentano i commenti, porzioni di codice che non vengono prese in considerazione dalla macchina ma che sono utili per noi umani a comprendere meglio e in minor tempo ciò che un codice dovrebbe fare.
Per inserire un commento
ci basterà scrivere doppio slash (//),
tutto ciò che è a destra degli slash (su quella riga) diventerà un commento.
Se abbiamo bisogno di commentare più righe possiamo farlo usando /* e */ ad esempio:

```
/*
righe commentate…
*/
```



La funzione Debug.Log()

Possiamo scrivere Debug.Log() per stampare nella Console un messaggio.

Ad esempio proviamo a scrivere
Debug.Log(“Hello World!”); // il testo del messaggio è racchiuso tra virgolette!
alla riga 10 del codice creato nei passaggi precedenti.

Una volta aggiunta la componente NuovoScript ad un GameObject nella scena, entrando in modalità Play vedremo comparire nella Console il messaggio
“Hello World!”

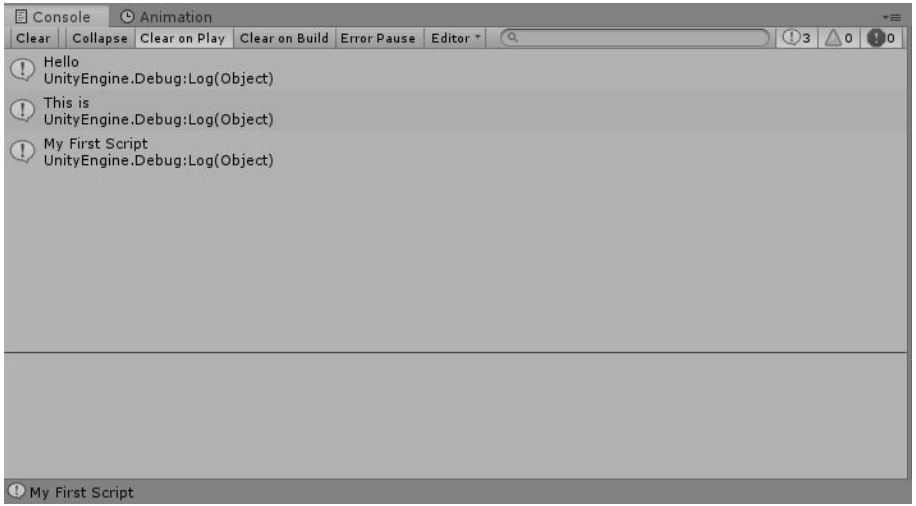
NB: Debug.Log() è un’istruzione (un’azione da far fare al programma), ogni volta che scriviamo un’istruzione dobbiamo **SEMPRE** mettere un ; dopo di essa

Esecuzione del codice

Le istruzioni (i comandi) che scriviamo in un codice vengono eseguite in maniera sequenziale dall'alto verso il basso.
Ad esempio scrivendo le righe:

```
Debug.Log("Hello");
Debug.Log("This is");
Debug.Log("My First Script");
```

Otterremo nella console l'effetto ritratto nell'immagine.



Non possiamo però scrivere queste istruzioni ovunque, dobbiamo scriverle dentro degli appositi blocchi di codice (racchiusi tra parentesi graffe). Iniziamo col più semplice da usare in Unity.

La funzione Start()

La funzione Start() di una componente di Unity rappresenta un blocco di codice che viene eseguito all'inizio della scena. Se mettiamo il nostro NuovoScript su 3 oggetti in scena, appena avvieremo la modalità **Play**, TUTTI E TRE stamperanno gli stessi messaggi.

Per questo motivo nelle slide precedenti è stato consigliato di inserire il richiamo della funzione Debug.Log() all'interno delle graffe di Start().

```
public class NuovoScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        Debug.Log("Hello");
        Debug.Log("This is");
        Debug.Log("My First Script");
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

La funzione Update()

La funzione Update() di una componente Unity viene invece richiamata ad ogni frame, ovvero ogni volta che la scena viene disegnata sullo schermo.

Nel codice qui a fianco, il messaggio verrà stampato continuamente fintanto che il GameObject a cui abbiamo aggiunto NuovoScript sarà attivo.

```
public class NuovoScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        Debug.Log("Ogni frame");
    }
}
```

Errori

Un errore in un programma è chiamato bug.

Il processo di eliminazione errori è chiamato debugging.

Ci sono tre tipi di errori

- Errori di sintassi
- Errori di esecuzione
- Errori logici

Errori di sintassi (syntax errors)

- Sono errori grammaticali le regole di scrittura sono molto “strette”
- Il compilatore rileva gli errori di sintassi e scrive un messaggio di errore
- Esempio: dimenticare un ; alla fine di una istruzione

Errori di esecuzione (runtime errors)

- Errori che vengono individuati mentre si esegue il programma (non durante la compilazione)
- Quando il computer trova un errore, il programma termina e viene stampato sullo schermo un messaggio di errore.
- Esempio: provare a fare una divisione per 0

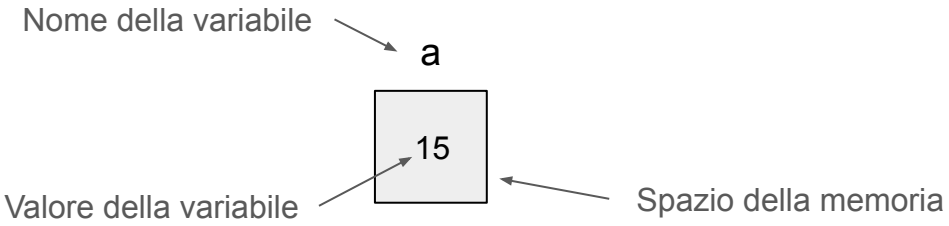
Errori logici (logic errors)

- Errori che non vengono segnalati nè durante la compilazione nè durante l'esecuzione ma che portano il programma a produrre risultati sbagliati.
- DIFFICILI da individuare! Occorre testare il programma con particolari dati in input…

Le variabili

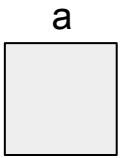
Oltre a stampare messaggi sulla console, nei nostri script sarà importante immagazzinare e modificare dei dati. Questa sarà la base con cui poi potremo scrivere le complesse meccaniche di gioco. Alla base di tutto ci sono le variabili. Una variabile è uno spazio nella memoria della macchina dentro cui mettiamo un valore, possiamo immaginarla come un cassetto. Vediamo come definire una variabile che contiene al suo interno un numero intero:

```
int a;
a = 15;
```



Le variabili

Analizziamo questo primo codice:
int a; // In questa riga DICHIARIAMO una variabile
Dichiarare una variabile vuol dire specificarne il tipo (un numero intero) e il nome, così facendo viene preparato nella memoria un adeguato spazio per inserire poi il valore di questa variabile. Per dichiarare una variabile bisogna sempre scrivere prima il **TIPO** della variabile e poi il suo **NOME**. Il nome possiamo sceglierlo noi, va bene qualsiasi nome (le regole sono: no spazi, solo caratteri alfanumerici e l'underscore, non può iniziare con un numero) è tuttavia usare un nome che abbia senso e aiuti a capire lo scopo di quella variabile. In questo esempio il *tipo* di dato della variabile è un numero intero (**int**) mentre il *nome* della variabile è semplicemente **a**.
A questo punto la variabile è dichiarata ma **non è** definita, ovvero non ha ricevuto ancora nessun valore.



Le variabili

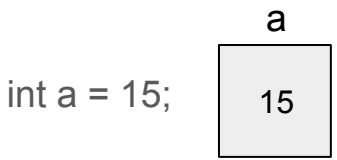
Proseguiamo col codice:

a = 15; // In questa riga ASSEGNIAMO un valore alla variabile a

Quando assegniamo un valore per la prima volta ad una variabile, la stiamo DEFINENDO.

Potremmo dichiarare e definire una variabile nella stessa riga scrivendo:

int a = 15;



NB: quando usiamo il simbolo = non stiamo scrivendo un'equazione come in matematica.

"=" viene chiamato operatore di assegnazione, ciò significa che

"il valore che sta a DESTRA dell'uguale verrà inserito nella variabile alla sua SINISTRA".

Ad esempio **NON POSSIAMO SCRIVERE 15 = a;**

(poiché equivale a dire "mettiamo il valore di a all'interno di 15" e non ha senso)

I tipi di dati

Abbiamo detto che per dichiarare una variabile dobbiamo seguire la formula
tipoDiDato nomeVariabile;

Ecco una lista dei tipi di dati basilari più usati:

int (numero intero, ad esempio 19)

float (numero con la virgola, ad esempio 10.2f ← bisogna aggiungere sempre la **f** per distinguerli dai double)

double (numero con la virgola a doppia precisione rispetto al float, esempio 92.1)

char (un singolo carattere, i valori dei char vanno scritti tra apici come ‘d’)

string (del testo, i valori delle stringhe vanno scritte tra virgolette “in questo modo”)

bool (un valore booleano, ovvero che può avere solo due valori: true o false)

Gli operatori aritmetici

Possiamo lavorare sui dati numerici eseguendo le operazioni matematiche che conosciamo: + - * / %

```
int a = 0;
```



```
a = 5 + 5; // somma, il risultato è 10
```



```
a = 10 - 2; // differenza, il risultato è 8
```

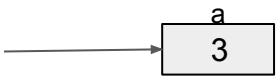


```
a = 3 * 4; // moltiplicazione, il risultato è 12
```



```
a = 23 / 2; // divisione, il risultato è 11 (il resto viene scartato)
```



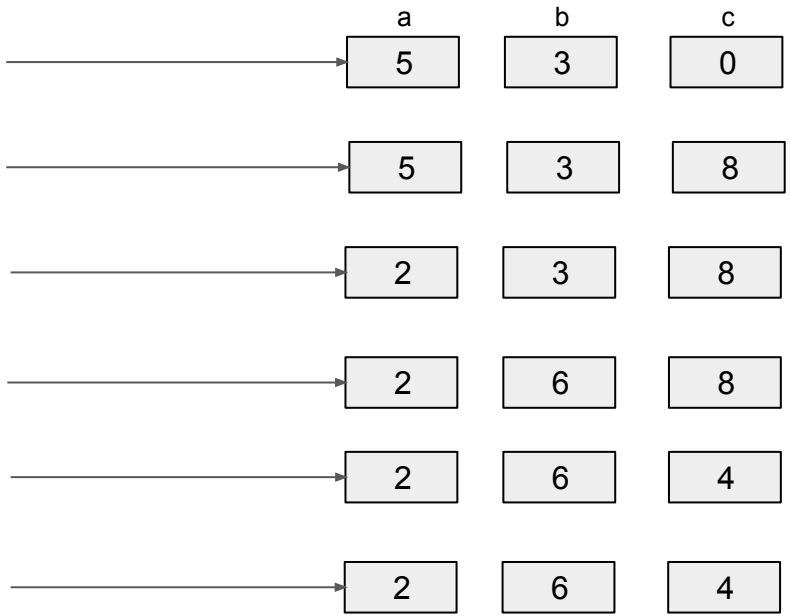
```
a = 13 % 5; // modulo, ovvero il resto della divisione 13 / 5, il risultato è 3
```


NB: la variabile **a** cambia sempre il suo valore poiché il risultato delle operazioni lo stiamo ASSEGNANDO alla variabile **a** sfruttando l'operatore di assegnazione **=**

Operazioni sulle variabili

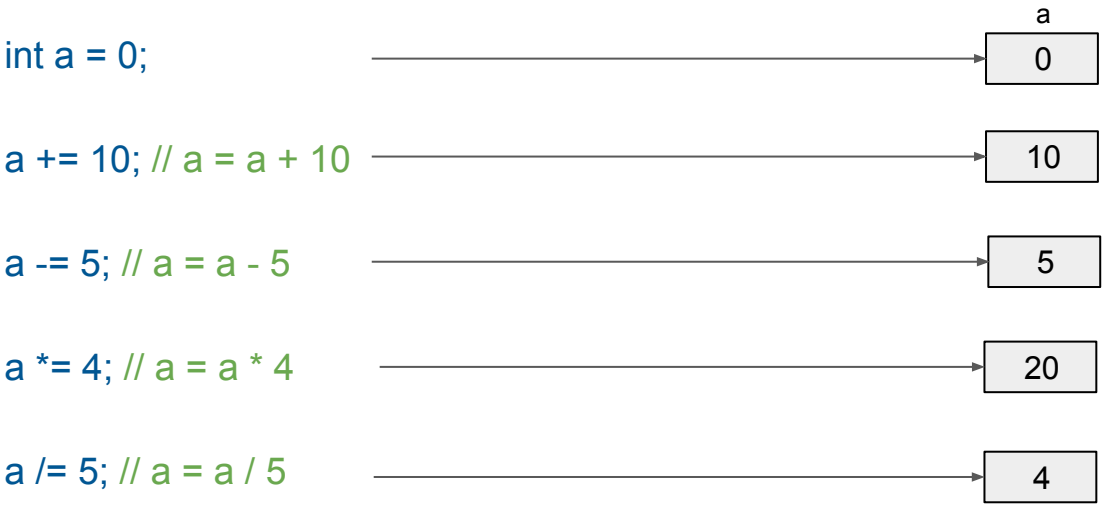
Possiamo anche mettere le variabili nella formula a destra dell'uguale, ad esempio:

```
int a = 5;
int b = 3;
int c = 0;

c = a + b; // c = 5 + 3        ovvero 8

a = c - b - 3; // a = 8 - 3 - 3  ovvero 2

b = a * b; // b = 2 * 3        ovvero 6

c = c / a; // c = 8 / 2        ovvero 4

a = b % c; // a = 6 % 4        ovvero 2
```



Operazioni sulle variabili

Se dobbiamo modificare una variabile eseguendo un'operazione possiamo usare le seguenti abbreviazioni:



Operazioni sulle variabili

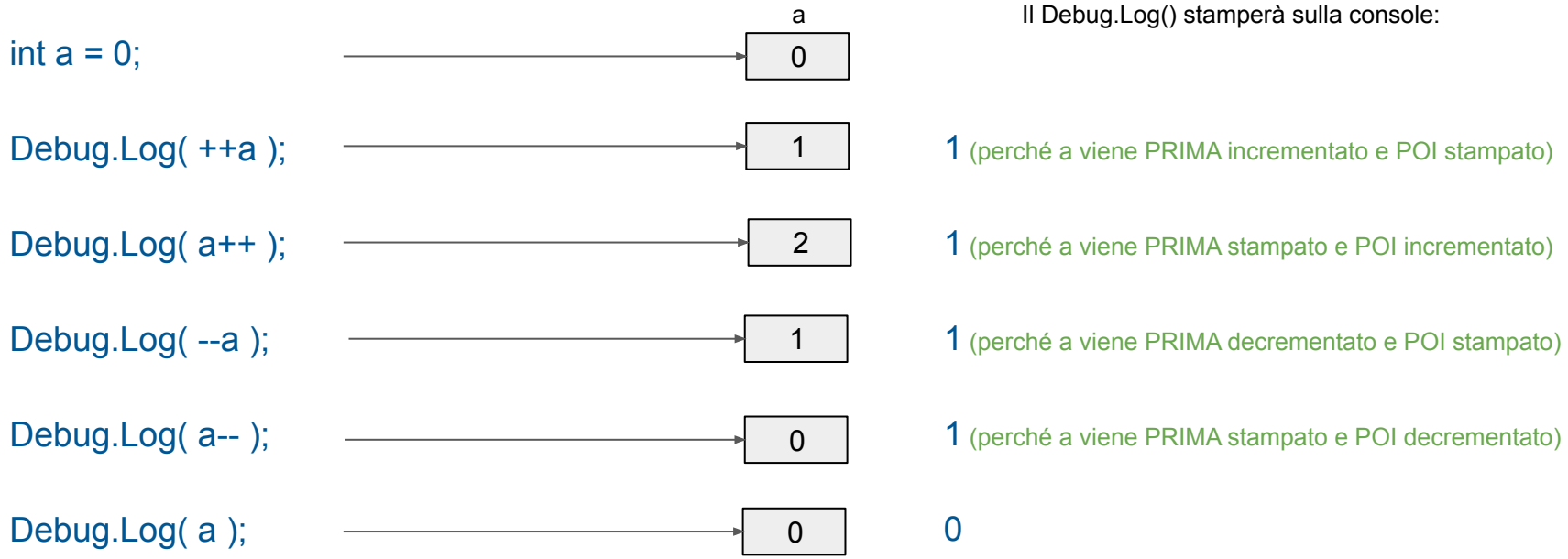
Se dobbiamo incrementare o decrementare una variabile di 1 possiamo usare gli operatori unari (che non necessitano di un secondo numero né dell'operatore di assegnazione)

Operazioni sulle variabili

La differenza tra ++a e ++a (come pure tra --a e a--) è la seguente:

++a e --a risolvono l'incremento/decremento **PRIMA** del resto della formula in cui sono inseriti

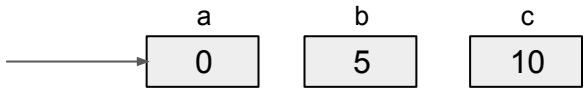
a++ e a-- risolvono l'incremento/decremento **DOPO** il resto della formula



Priorità degli operatori

L'ordine in cui le operazioni vengono eseguite è la stessa della matematica che conosciamo:
prima moltiplicazioni e divisioni, **poi** addizioni e sottrazioni (*da sinistra verso destra*).
Nel caso in cui siano presenti delle parentesi, *le operazioni nelle parentesi hanno la priorità sulle altre*.

```
int a = 0;
int b = 5;
int c = 10;
```




```
a = 1 + 2 * 3 - 15 / 3; // a = 2
```

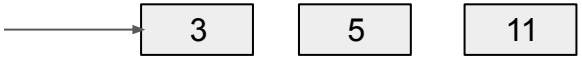

```
a = b * (6 - 4) / a; // a = 5
```

Solo **a** cambia di valore perché è l'unica a cui viene assegnato un nuovo valore con l'operatore **=**

```
a = 2 * c++; // a = 20
```

Anche **c** cambia di valore poiché l'operazione c++ incrementa direttamente il valore di **c** (ma nella formula verrà usato il "*vecchio*" valore di **c** ovvero **10**)

```
a += b - 2 * c; // a +=     -17
```




```
a = --c - b++; // a = 10 - 5 ovvero 5
```

L'operatore --c decrementa **c** e usa il nuovo valore di **c** nella formula, invece b++ incrementa **b** ma nella formula verrà usato il vecchio valore di **b** ovvero **5**

Somma tra stringhe

L'operatore + se usato tra stringhe (o tra una stringa a sinistra e un altro dato a destra) restituisce come risultato una stringa formata dalla concatenazione della stringa di sinistra con quella di destra.

```
string s1 = "Un anello";
string s2 = " per cercarli"; // notare lo spazio all'inizio della stringa prima della p
string s3 = s1 + s2;
// s3 = "Un anello per cercarli";
string s4 = "Un anello" + " per trovarli." + "Un anello" + " per ghermirli"; // Non c'è lo spazio prima della U
// s4 = "Un anello per trovarli.Un anello per ghermirli";
string s5 = "sommando un numero ottengo" + 4;
// s5 = "sommando un numero ottengo4";
string s6 = "attenzione però a fare" + 1 + 1;
// s6 = "attenzione però a fare11";  ← vengono concatenati singolarmente i due 1
string s7 = "se volete fare operazioni sui numeri usate le parentesi" + (1 + 1);
// s7 = "se volete fare operazioni sui numeri usate le parentesi2";  ← vengono prima sommati i due 1
```

Gli operatori di confronto

Gli operatori di confronto si occupano di rispondere a delle domande semplici su uguaglianze e disuguaglianze tra due valori. Il risultato di un operatore logico è sempre un valore booleano, ovvero un valore che può essere soltanto o true o false.
In pratica rispondono ad un domanda cui si può rispondere solo con Sì o No.

```
a == b // restituisce true se a e b hanno il medesimo valore, altrimenti restituisce false
a != b // restituisce true se a e b NON hanno il medesimo valore, altrimenti restituisce false
a > b // restituisce true se a è maggiore di b, altrimenti restituisce false
a >= b // restituisce true se a è maggiore o uguale a b, altrimenti restituisce false
a < b // restituisce true se a è minore di b, altrimenti restituisce false
a <= b // restituisce true se a è minore o uguale a b, altrimenti restituisce false
```

```
bool risultato = 5 > 3;
Debug.Log(risultato);  // stamperà true
Debug.Log( 5 < 3 ); // stamperà false
```

Gli operatori logici

Gli operatori logici si occupano di dare una risposta in base a due valori booleani, la risposta sarà a sua volta un valore booleano, quindi ancora una volta true o false, **Sì o No**.

a && b // operatore AND restituisce : true solo se a e b sono **entrambi true**
a || b // operatore OR restituisce : true solo se **almeno uno** tra a e b è **true**
a ^ b // operatore XOR (o OR esclusivo) : restituisce true solo se **soltanto uno** tra a e b è **true**
!a // operatore di negazione logica: restituisce **true se a è false** e **viceversa** (restituisce *l'inverso* di a)

Gli operatori logici vengono spesso utilizzati in formule composte con operatori di confronto ad esempio:
x > 10 && x <= 20 // restituisce true solo se x > 10 e x <= 20 sono entrambi true
// ovvero se x è compreso tra 10 (escluso) e 20 (incluso)

NB: l'operatore XOR usato tra numeri e non tra booleani NON calcola la potenza ma esegue un'operazione sui bit (che non sarà oggetto di questo corso). Ad esempio 10 ^ 2 = 8

Istruzioni Condizionali (if)

Finora i nostri codici erano sempre e solo una sequenza di istruzioni da eseguire, una dopo l'altra senza scelte o variazioni sull'esecuzione.

Le istruzioni condizionali ci permettono di inserire proprio questo: **una scelta**.

In un diagramma le istruzioni condizionali rappresentano/sono rappresentate da un bivio.
Il programma seguirà una sola delle due strade, in base ad una condizione.

Istruzioni Condizionali (if)

La condizione è semplicemente una domanda a cui si può rispondere **Sì o No**.
Ovvero un valore *booleano*.
E abbiamo già visto che domande di questo tipo si pongono usando
operatori di confronto e operatori logici.

Istruzioni Condizionali (if)

La sintassi di un'istruzione condizionale è la seguente:

```
if ( CONDIZIONE )
{
      // codice da eseguire se la condizione è true
}
// Il resto del programma
```

Il codice all'interno delle parentesi graffe verrà eseguito SOLO se la condizione è true

Istruzioni Condizionali (if)

Vediamo un esempio

```
if ( x > 5 )
{
	Debug.Log(“x e’ maggiore di 5”);
}
```

La frase verrà stampata solo se l’operazione x > 5 restituirà un valore true

Istruzioni Condizionali (if)

Vediamo un esempio

```
if ( true )
{
    Debug.Log(“Questa frase verra’ stampata sempre!”);
}
```

Poiché il valore nella condizione è direttamente true (un if inutile)

Istruzioni Condizionali (if)

Se vogliamo far eseguire del codice solo nel caso in cui la condizione non è rispettata possiamo utilizzare la parola chiave else e altre parentesi graffe:

```
if ( x > 10 )
{
        Debug.Log(“Questa frase verra’ stampata solo se x > 10”);
}
else
{
        Debug.Log(“Questa frase verra’ stampata solo se x <= 9”);
}

Debug.Log(“Questa frase verra’ stampata sempre!”);
```

Istruzioni Condizionali (if)

Possiamo concatenare molteplici condizioni if utilizzando il costrutto else if ()

```
if ( x > 10 )
{
        Debug.Log(“Questa frase verra’ stampata solo se x > 10”);
}
else if ( x < 0 )
{
        Debug.Log(“Questa frase verra’ stampata solo se x < 0”);
}
else
{
        Debug.Log(“Questa frase verra’ stampata solo se x e’ compreso tra 0 e 10”);
}
Debug.Log(“Questa frase verra’ stampata sempre!”);
```

Cicli

Se abbiamo bisogno di ripetere più volte uno stesso codice, ad esempio se decidessimo di stampare dieci volte un messaggio oppure eseguire un'operazione più volte su un insieme di numeri, dovremo riscrivere molte volte le stesse righe di codice, mettendo al massimo qualche piccola variazione.

In casi come questo ci vengono in soccorso i cicli (loops in inglese).

Ogni singola esecuzione del codice all'interno di un ciclo viene chiamata iterazione.

Cicli while

Il primo ciclo che vedremo è il ciclo while.
La sintassi è simile a quella di una condizione if ed è la seguente:

```
while ( CONDIZIONE )
{
	// Istruzioni da ripetere
}
```

Quando il programma arriverà al ciclo valuterà se la *condizione è true*, in caso affermativo eseguirà tutto il codice compreso tra le parentesi graffe.
Una volta terminato il codice controllerà nuovamente se la condizione è true, in caso affermativo eseguirà tutto il codice compreso tra le parentesi **DI NUOVO**. Continuerà così *fintanto che la condizione è true*.
Uscirà da questo ciclo solo quando la condizione sarà false.

Cicli while

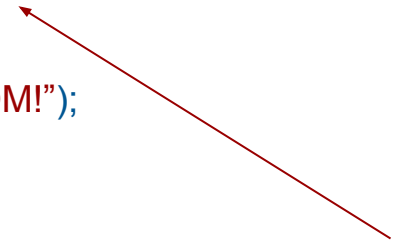
E’ di vitale importanza che la condizione diventi eventualmente false, altrimenti ci ritroveremmo in un loop infinito. Ed un loop infinito fa impallare i programmi.

Il modo più classico di usare un ciclo while per ripetere qualcosa un determinato numero di volte è quella di
usare una variabile (dichiarata PRIMA del ciclo) come contatore,
incrementare il contatore all’interno del ciclo
e valutare il valore del contatore nella condizione.

Cicli while

Esempio: stampiamo un conto alla rovescia.

```
int contatore = 10;
while ( contatore >= 0 )
{
        Debug.Log(“Mancano ancora “ + contatore + “ all’esplosione”);
        contatore--;
}
Debug.Log(“BOOM!”);
```



Se ci si scorda di questa riga il loop diventa infinito!

Risultato sulla console:

“Mancano ancora 10 all’esplosione”
“Mancano ancora 9 all’esplosione”
“Mancano ancora 8 all’esplosione”
“Mancano ancora 7 all’esplosione”
“Mancano ancora 6 all’esplosione”
“Mancano ancora 5 all’esplosione”
“Mancano ancora 4 all’esplosione”
“Mancano ancora 3 all’esplosione”
“Mancano ancora 2 all’esplosione”
“Mancano ancora 1 all’esplosione”
“Mancano ancora 0 all’esplosione”
“BOOM!”

Cicli while

Esempio: stampiamo i numeri pari tra 5 e 25.

```
int contatore = 5;
while ( contatore <= 25 )
{
        if ( contatore % 2 == 0 )
        {
                Debug.Log ( “Il num “ + contatore + “ e’ pari!”);
        }
        contatore++;
}
```

Risultato sulla console:

“Il num 6 e’ pari!”
“Il num 8 e’ pari!”
“Il num 10 e’ pari!”
“Il num 12 e’ pari!”
“Il num 14 e’ pari!”
“Il num 16 e’ pari!”
“Il num 18 e’ pari!”
“Il num 20 e’ pari!”
“Il num 22 e’ pari!”
“Il num 24 e’ pari!”

Cicli while

Esempio: un ciclo in cui non entrerà mai poiché la condizione è falsa.

```
int contatore = 10;
while ( contatore < 3 )
{
        Debug.Log ( “Il valore del contatore e’“ + contatore);
        contatore++;
}
Debug.Log ( “Fine del ciclo“);
```

Risultato sulla console:

“Fine del ciclo”

Cicli do while

I cicli do while sono del tutto simili ai cicli while, ma garantiscono l'esecuzione del codice tra le parentesi graffe ALMENO una volta.
La sintassi è la seguente:

```
do
{
        // Istruzioni da eseguire
} while ( CONDIZIONE );
```

NB: fare attenzione al punto e virgola dopo la condizione!

Cicli do while

Esempio: stampiamo un conto alla rovescia.

```
int contatore = 10;
do
{
        Debug.Log(“Mancano ancora “ + contatore + “ all’esplosione”);
        contatore--;
} while ( contatore >= 0 );
Debug.Log(“BOOM!”);
```

Risultato sulla console:

“Mancano ancora 10 all’esplosione”
“Mancano ancora 9 all’esplosione”
“Mancano ancora 8 all’esplosione”
“Mancano ancora 7 all’esplosione”
“Mancano ancora 6 all’esplosione”
“Mancano ancora 5 all’esplosione”
“Mancano ancora 4 all’esplosione”
“Mancano ancora 3 all’esplosione”
“Mancano ancora 2 all’esplosione”
“Mancano ancora 1 all’esplosione”
“Mancano ancora 0 all’esplosione”
“BOOM!”

Cicli do while

Esempio: stampiamo i numeri pari tra 5 e 25.

```
int contatore = 5;
do
{
        if ( contatore % 2 == 0 )
        {
                Debug.Log ( “Il num “ + contatore + “ e’ pari!”);
        }
        contatore++;
} while ( contatore <= 25 );
```

Risultato sulla console:

“Il num 6 e’ pari!”
“Il num 8 e’ pari!”
“Il num 10 e’ pari!”
“Il num 12 e’ pari!”
“Il num 14 e’ pari!”
“Il num 16 e’ pari!”
“Il num 18 e’ pari!”
“Il num 20 e’ pari!”
“Il num 22 e’ pari!”
“Il num 24 e’ pari!”

Cicli do while

Esempio: un ciclo in cui non entrerà mai poiché la condizione è falsa.

```
int contatore = 10;
do
{
        Debug.Log ( “Il valore del contatore e’“ + contatore);
        contatore++;
} while ( contatore < 3 );
Debug.Log ( “Fine del ciclo“);
```

Risultato sulla console:

“Il valore del contatore e’ 10”
“Fine del ciclo”

Cicli for

I cicli for sono i più utilizzati quando si devono eseguire istruzioni su un intervallo numero o per un determinato numero di volte. Questo perché condensano all'interno della loro prima riga la definizione del contatore, la condizione e l'incremento del contatore. La loro sintassi è la seguente

```
for ( DEFINIZIONE CONTATORE ; CONDIZIONE ; INCREMENTO CONTATORE )
{
    // Istruzioni da eseguire
}
```

NB: notare i due punti e virgola che separano la condizione dal resto

NB: ciò che scriveremo al posto di INCREMENTO CONTATORE verrà eseguito DOPO OGNI ITERAZIONE del ciclo

ciò che scriveremo al posto di DEFINIZIONE CONTATORE verrà eseguito PRIMA del controllo della CONDIZIONE

Cicli for

Esempio: stampiamo un conto alla rovescia.

```
for (int contatore = 10; contatore >= 0; contatore--)
{
        Debug.Log(“Mancano ancora “ + contatore + “ all’esplosione”);
        // contatore--;
}
Debug.Log(“BOOM!”);
```

Non è più necessaria perché il **contatore** viene incrementato automaticamente dopo ogni iterazione (poiché abbiamo scritto **contatore--** dopo la condizione)

Risultato sulla console:

“Mancano ancora 10 all’esplosione”
“Mancano ancora 9 all’esplosione”
“Mancano ancora 8 all’esplosione”
“Mancano ancora 7 all’esplosione”
“Mancano ancora 6 all’esplosione”
“Mancano ancora 5 all’esplosione”
“Mancano ancora 4 all’esplosione”
“Mancano ancora 3 all’esplosione”
“Mancano ancora 2 all’esplosione”
“Mancano ancora 1 all’esplosione”
“Mancano ancora 0 all’esplosione”
“BOOM!”

Cicli for

Esempio: stampiamo i numeri pari tra 5 e 25.

```
for (int contatore = 5; contatore <= 25; contatore++)
{
        if ( contatore % 2 == 0 )
        {
                Debug.Log ( “Il num “ + contatore + “ e’ pari!”);
        }

}
```

Risultato sulla console:

“Il num 6 e’ pari!”
“Il num 8 e’ pari!”
“Il num 10 e’ pari!”
“Il num 12 e’ pari!”
“Il num 14 e’ pari!”
“Il num 16 e’ pari!”
“Il num 18 e’ pari!”
“Il num 20 e’ pari!”
“Il num 22 e’ pari!”
“Il num 24 e’ pari!”

Cicli for

Esempio: un ciclo in cui non entrerà mai poiché la condizione è falsa.

```
for (int contatore = 10; contatore < 3; contatore++)
{
        Debug.Log ( “Il valore del contatore e’“ + contatore);

}
Debug.Log ( “Fine del ciclo“);
```

Risultato sulla console:

“Fine del ciclo”

Break

All’interno di un ciclo è possibile forzarne l’uscita (in un qualsiasi punto) indipendentemente dalla condizione usando la parola chiave break.

Esempio: un ciclo infinito che stampa i multipli di 2 e 5, ma se trova un multiplo di entrambi interrompe il ciclo immediatamente.

```
int n = 1;
while ( true ) // un ciclo infinito, la condizione e’ sempre true!
{
        if ( n % 2 == 0 && n % 5 == 0 )
        {
                break;
        }
        else if ( n % 2 == 0)
        {
                Debug.Log ( “Il numero “ + n + “ e’ multiplo di 2“);
        }
        else if ( n % 5 == 0 )
        {
                Debug.Log ( “Il numero “ + n + “ e’ multiplo di 5“);
        }
        n++;
}
Debug.Log ( “Fine del ciclo“);
```

Risultato sulla console:

“Il numero 2 e’ multiplo di 2”
“Il numero 4 e’ multiplo di 2”
“Il numero 5 e’ multiplo di 5”
“Il numero 6 e’ multiplo di 2”
“Il numero 8 e’ multiplo di 2”
“Fine del ciclo”

Continue

All’interno di un ciclo è possibile forzare il passaggio all’iterazione successiva usando la parola chiave continue (nei cicli for l’incremento/decremento del contatore viene eseguito comunque).

Esempio: un ciclo che va da 1 a 10 e che stampa il valore del contatore ad ogni iterazione e stampa la frase “che e’ multiplo di 3” quando il valore del contatore i è un multiplo di 3

```
for (int i = 1; i < 10; i++)
{
        Debug.Log ( “Iterazione con i = “ + i);

        if ( i % 3 != 0 )
        {
                continue;
        }

        Debug.Log ( “Che e’ un multiplo di 3“);
}
Debug.Log ( “Fine del ciclo“);
```

Risultato sulla console:

“Iterazione con i = 1”
“Iterazione con i = 2”
“Iterazione con i = 3”
“Che e’ un multiplo di 3”
“Iterazione con i = 4”
“Iterazione con i = 5”
“Iterazione con i = 6”
“Che e’ un multiplo di 3”
“Iterazione con i = 7”
“Iterazione con i = 8”
“Iterazione con i = 9”
“Che e’ un multiplo di 3”
“Fine del ciclo”

Continue

ATTENZIONE: nei cicli while l'utilizzo del continue potrebbe inibire l'incremento del contatore!

Esempio: un ciclo che va da 3 a 10 e che stampa il valore del contatore ad ogni iterazione e stampa la frase "che e' multiplo di 3" quando il valore del contatore i è un multiplo di 3

```
int i = 3; // ← parto da 3 cosi' viene evitato il continue una volta
while (i < 10)
{
        Debug.Log ( "Iterazione con i = " + i);

        if ( i % 3 != 0 )
        {
                continue;
        }
        Debug.Log ( "Che e' un multiplo di 3");
        i++;
}
Debug.Log ( "Fine del ciclo");
```

Risultato sulla console:

"Iterazione con i = 3"
"Che e' un multiplo di 3"
"Iterazione con i = 4"
"Iterazione con i = 4"
"Iterazione con i = 4"
"Iterazione con i = 4"
"Iterazione con i = 4"
"Iterazione con i = 4"
…
UN CICLO INFINITO!!!!

Quando usare break e continue

Ogni ciclo che utilizza break e/o continue può essere riscritto senza l'utilizzo di queste istruzioni.

Nella maggior parte dei casi è consigliabile evitare di utilizzarli,
soprattutto finché non si acquisisce una buona esperienza coi cicli.

Possono però tornare utili quando riscrivere un ciclo senza di essi ne peggiora in maniera critica la leggibilità (ad esempio introducendo un numero eccessivo di condizioni o di if uno dentro l'altro).

Enumeratori

Un tipo di variabile che possiamo definire noi. Ci permette di elencare una serie di valori, definendo noi stessi i nomi di questi valori.
Ad esempio potremmo fare l'elenco degli elementi delle magie di un gioco:
FUOCO, TERRA, ARIA, ACQUA, GHIACCIO, FULMINE.

Se usassimo una variabile di tipo int (associando per esempio FUOCO a 0 e TERRA a 1) dovremmo sempre ricordarci quale numero indica quale elemento.
Se usassimo una variabile di tipo string (ad esempio "Fuoco" e "Terra") potrebbe accadere di sbagliare a scrivere anche un singolo carattere, una volta, e ritrovarci con una variabile che non fa quello che ci aspettiamo.

Enumeratori

Con gli enumeratori possiamo definire noi l'elenco dei valori possibili.
Ad esempio scriviamo l'elenco degli elementi usati nel gioco così:

```
public enum ELEMENTI
{
    FUOCO,
    TERRA,
    ACQUA,
    ARIA,
    GHIACCIO,
    FULMINE
}
```

Enumeratori

Possiamo poi dichiarare e utilizzare variabili di tipo ELEMENTI in questo modo:

```
ELEMENTI mioElemento = ELEMENTI.FUOCO;
```

ELEMENTI è il tipo della variabile.
mioElemento è il nome della variabile.
ELEMENTI.FUOCO è il valore (metto il “nome dell’enum” . “nome del valore”)

Usando questa tecnica non potremo mai sbagliare, non senza accorgercene, perché se ad esempio scrivessi ELEMENTI.FUCO il compilatore ci darebbe errore.

Switch

Un altro elemento per il controllo del flusso di un programma.

Simile ad una sequenza di if / else if

```
int mese = 2;
switch ( mese )
{
    case 10:
        Debug.Log(“Ottobre”);
        break;
    case 2:
        Debug.Log(“Febbraio”);
        break;
    (....)
}
```

Valore in base a cui fare una scelta

Elenco dei possibili valori che “mese” (o la variabile messa in cima tra parentesi tonde) può assumere.

Switch

Molto utile quando un valore può assumere un piccolo set di valori.

```
int mese = 2;
switch ( mese )
{
      case 1:
            Debug.Log(“Gennaio”);
            break;
      case 2:
            Debug.Log(“Febbraio”);
            break;
      (....)
}
```

Questi sono due punti, non punto e virgola!

Una o più istruzioni da eseguire se il valore di “mese” coincide proprio col valore scritto dopo la parola chiave “case”

Necessario tra un “case” e l’altro. Avverte il codice di uscire dallo switch (e proseguire col resto del codice)

Switch

```
int mese = 2;
switch ( mese )
{
      case 1:
      case 2:
      case 3:
            Debug.Log(“Primo Trimestre”);
            break;
      default:
            Debug.Log(“Mese sconosciuto”);
      (....)
}
```

Mettendo più case uno dopo l’altro, l’elenco di istruzioni seguenti saranno eseguite se il valore di “mese” sarà uguale a uno qualsiasi di questi tre valori.

Posso aggiungere un caso di default nel caso in cui il valore che passo allo switch non sia presente tra i valori dei vari “case”
(ad esempio se passassi 13 come mese)