```
import UIKit

// **********************
// VARIABILI E COSTANTI
// **********************

// Variabili
var str = "Hello "
str += "Playground"
print("Interpolazione di stringa: \(str)")
var num = 21
num += 3

// Costanti
// Preferibili quando possibile poiché consentono al compilatore di
eseguire l'ottimizzazione del codice
let num1 = 12
let num2 = 23
let totale = num1 + num2
// Tentare di variare il valore restituisce un errore
// num2 = 24


// **********************
// TIPIZZAZIONE ESPLICITA E IMPLICITA
// **********************

// Tipizzazione esplicita
var str1:String = "La condizione"
// Se non si imposta un valore iniziale o se questo  non fornisce
sufficienti informazioni per l'inferenza del tipo da parte del compilatore
// occorre usare la tipizzazione esplicita
// Double, più preciso di float, è il valore di default (implicito) per i
decimali
let numInt:Int = 12
let num3:Float = 16.50
let num4:Double = 18.50
let condizione: Bool = true

// Tipizzazione implicita
// Preferibile quando possibile poiché il compilatore è in grado di
inferire il tipo e comporta una minor scrittura di codice
var str2 = "Stringa implicita"
let num5 = 33.50
let condizione2 = false

// ***
// IF
// ***

//Nel costrutto if le parentesi tonde attorno alla condizione sono
```

```
opzionali mentre le parentesi graffe sono obbligatorie
if condizione {
    print("\(str1) è vera")
} else {
    print("\(str1) è falsa")
}

if(num1 < num2){
    print("OK")
} else {
    print("No")
}


// **********************
// COLLEZIONI
// **********************

// ARRAY
// Tipo implicito
var mioArray = ["Aldo", "Giovanni", "Giacomo"]
var mioArrayEsp:[String] = ["A","B","C"]
// Tipo esplicito
var mioArray1:Array<Double> = [0.10, 0.15, 0.23]
// [Double] è una contrazione di Array<Double>
var mioArray2:[Double] = [0.15, 0.18, 0.20]
// Uso di inizializzatore per Array senza valori iniziali
var arrayVuoto = [String]()

mioArray.append("Alda")
var mioArray3 = ["Giorgia", "Ely"]
mioArray += (mioArray3)
mioArray[4]
mioArray.insert("Pippo", atIndex:3)
mioArray[4]
mioArray.removeAtIndex(0)
mioArray[4]
mioArray.count
mioArray.removeLast()
mioArray.count
var mioArrayOrdinato = mioArray.sort{$1 > $0}
// Ciclo per array
for i in mioArray {
    print(i)
    print("Mio valore non ordinato: \(mioArray)")
}
for i in mioArrayOrdinato {
    print(i)
    print("Mio valore ordinato: \(mioArrayOrdinato)")
}
```

```swift
// Ciclo con operatore non inclusivo ..<
for i in 0..<4{
    // Cicla da 0 a 4 escluso
    print(i)
}
// Ciclo con operatore inclusivo ...
for i in 0...4{
    // Cicla da 0 a 4 incluso
    print(i)
}
for i in 0..<mioArray.count {
    print("Voce \(i): \(mioArray[i])")
}
// Ciclo con operatore inclusivo ...
for i in 0...mioArray.count-1{
    print("Voce \(i): \(mioArray[i])")
}

// DIZIONARI
var mioDictionary = ["Ciliege": 2.05, "Mele": "Non disponibili", "Fragole":
1]
let value = mioDictionary["Ciliege"]
print(value)
print(mioDictionary.count)

for chiave in mioDictionary.keys {
    print("Chiave: \(chiave)")
}
for valore in mioDictionary.values {
    print("Valore: \(valore)")
}
mioDictionary.updateValue(2.25, forKey: "Ciliege")
print(mioDictionary["Ciliege"])


// **********************
// SWITCH
// **********************

// In Swift, i case: possono essere di qualunque tipo, multipli e contenere
intervalli
let valore = "Giovanni"

switch valore {
    case "Aldo":
     let cognome = "Rossi"
    case "Giovanni", "Giacomo":
     let cognome = "Verdi"
    // Obbligatorio
    default:
     let cognome = "Neri"
```

```
}
var mioVoto=4
switch mioVoto {
    case 1...5:
     print("Insufficiente")
    case 6...8:
     print("Buono")
    default:
     print("Senza voto")
}
// La clausola where aggiunge una condizione extra all'istruzione case che
utilizza la clausola.
// Nell'esempio seguente si sottopone a switch una tupla
let yz = (1,2)
switch yz {
case let (a, b) where a == b:
    print("(\(a), \(b)) y e z sono uguali")
case let (a, b) where a <= b:
    print("(\(a), \(b)) y è minore di z")
case let (a, b) where a >= b:
    print("(\(a), \(b)) y è maggiore di z")
default:
    print("")
}

// Le istruzioni 'case:' non vengono eseguite in cascata di default. Una
precisa scelta progettuale per evitare errori comuni.
// Se un case specifico lo necessita è possibile utilizzare la keyword
fallthrough per indicare questa volontà al compilatore
var mioVoto2=6
switch mioVoto2 {
    case 5:
     print("Insufficiente")
    case 6:
     fallthrough
    case 7:
     print("Discreto")
    case 8:
     print("Buono")
    default:
     print("Senza voto")
}

// Il binding per valore è utilizzato con le keywords "case let" per legare
una costante al case corrispondente
let xy = (0,2)
switch xy {
    case (let x, 0):
     print("X = \(x)")
    case (0, let y):
     print("Y = \(y)")
```

```
    case let (x, y):
     print("X e Y = (\(x), \(y))")
}


// FUNZIONI
//La sintassi per le funzioni è la seguente:
func nomeFunzione(nomeArgomento: tipoArgomento) -> tipoRestituito {
    [...]
    return tipoRestituito;
}

func saluta(nome: String) -> String {
    let saluto = "Ciao \(nome)"
    return saluto
}
saluta("Aldo")

// Funzioni con nomi espliciti
// func nomeFunzione(nomeEsplicito1 nomeLocale1:tipo, nomeEsplicito2
nomeLocale2:tipo) -> tipoRestituito {}
func salutaCon(Nome n:String, Cognome c:String) -> String {
    let saluto = "Ciao \(n) \(c)"
    return saluto
}
salutaCon(Nome:"Aldo", Cognome:"")

//Parametri variadici
func contaString(Stringa:String...)->Int{
    return Stringa.count
}
contaString("A","B")


// **********************
// ENUMERAZIONI
// **********************
// Variabile o costante che contiene un insieme definito di valori
enum Stagioni {
    case Primavera, Estate, Autunno, Inverno
}
let miaStagionePreferita = Stagioni.Primavera

// Le enumerazioni consentono l'uso di funzioni
enum Stagioni2 {
    case Primavera, Estate, Autunno, Inverno

    func traduciEng() -> String {
     switch self {
     case .Primavera:
          return "Spring"
```

```
     case .Estate:
          return "Summer"
     case .Autunno:
          return "Autumn"
     case .Inverno:
          return "Winter"
     }
    }
}
let miaStagionePreferita2 = Stagioni2.Primavera
let myPrefSeason = miaStagionePreferita2.traduciEng()
print("\(myPrefSeason)")


// **********************
// TUPLE
// **********************
// Le tuple sono una via di mezzo tra array e structures
// Dichiarazione semplice di tupla
var persona = ("Aldo", "Rossi", 32)
// Estrazione dei dati in base al numero di indice
print ("\(persona.0)")
print ("\(persona.1)")
print ("\(persona.2)")
// Dichiarazione nominale di tupla
var altraPersona = (nome:"Giovanni", cognome:"Bianchi", eta:37)
// Estrazione dei dati in base alla chiave
print ("\(altraPersona.nome)")
print ("\(altraPersona.cognome)")
print ("\(altraPersona.eta)")
// Estrazione per scomposizione
var (nome, cognome, eta) = altraPersona
print ("\(nome)")
print ("\(cognome)")
print ("\(eta)")

// In Swift, le funzioni possono restituire valori multipli in forma di
tupla
func calcolaMinMax(numeri:[Int]) -> (min:Int, max:Int) {
    var currentMin = numeri[0]
    var currentMax = numeri[0]
    for valore in numeri[1..<numeri.count] {
     if valore < currentMin {
          currentMin = valore
     } else if valore > currentMax {
          currentMax = valore
     }
    }
    return (currentMin, currentMax)
}
let miaSerie = calcolaMinMax([8, -6, 2, 109, 3, 71])
```

```
print("val min is \(miaSerie.min) val max is \(miaSerie.max)")


// Le tuple sono trattate come variabili singole, ovvero possono essere passate come parametri
let miaTupla = (12,3)
func somma(x:Int, y:Int) -> Int {
    return x + y
}
somma(miaTupla)



// CLOSURE
// Le closures sono l'equivalente dei blocchi in Objective-C, possono essere invocati in un secondo tempo e hanno accesso allo scope nel quale sono state definiti in origine.
// Possono essere definite in maniera inline, passate come parametri, o restituite dalle funzioni.

// Una funzione Swift può accettare come parametro anche una o più funzioni
// Nell'esempio la funzione accetta il parametro "parNome" di tipo String e il parametro risultante dalla funzione
// "parFunzione" la quale, a sua volta, accetta un parametro di tipo String e restituisce un valore di tipo String
// Il corpo della funzione "principale" esegue la funzione definita come secondo parametro passandole come argomento il primo parametro
func miaFunzione(parNome:String, parFunzione:(String)->String) {
    print(parFunzione(parNome))
}
// Senza closure è necessario
// 1. definire la funzione che sarà passata come secondo parametro
func funzionePar(parStr:String)->String{
    return parStr
}
// 2. Passarla alla funzione "principale"
miaFunzione("Ciro", parFunzione:funzionePar)

// Con la closure è possibile abbreviare, indicando il parametro che verrebbe passato alla funzione "parametro" prima della keyword
miaFunzione("Ciro"){ (parNome) -> String in
    return(parNome)
}

// Esempio di closure che calcola i valori pari
let valPari = { (num: Int) -> Bool in
    let modulo = num % 2;
    return (modulo == 0);
}
// Per passare una closure come parametro di una funzione, usiamo il tipo della closure nella definizione della funzione
```

```
func verificaPari(num: Int, verifica: (Int -> Bool)) -> Bool {
    return verifica(num);
}
verificaPari(12, verifica:valPari)
// restituisce true
verificaPari(13, verifica: valPari)
// restituisce false



// **********************
// CLASSI
// **********************
// Le classi Swift sono gestite per riferimento
// Dichiarazione di classe
class MiaCalcolatrice {
    // Le proprietà devono essere inizializzate al momento della
dichiarazione (a meno che siano optionals.
    // Diversamente, occorre definire un inizializzatore.
    let totale: Double
    let iva: Double
    // Inizializzatore
    init(totale:Double, iva:Double) {
     self.totale = totale
     self.iva = iva
    }
    // Metodo
    func calcolaSubtotale() -> Double {
     return totale + (totale / iva)
    }
}
// Si usa la dot notation per accedere alle proprietà di un oggetto, per
invocare metodi di classe e di istanza
let miaCalc1 = MiaCalcolatrice(totale: 50.00, iva: 22)
print(miaCalc1.calcolaSubtotale())

// L'ereditarietà (singola) segue la classica struttura con il carattere di
due punti
class MiaSuperClasse {
    let proprietaSup:String
    init(proprietaSup:String){
     self.proprietaSup = proprietaSup
    }
    func metodoSup()->String {
     return "Ciao \(proprietaSup)"
    }
}
class MiaSubClasse:MiaSuperClasse {
    let proprietaSub:String
    init(proprietaSub:String, proprietaSup:String){
     self.proprietaSub = proprietaSub
```

```
     super.init(proprietaSup:proprietaSup)
    }
}
let MiaIstanzaSup = MiaSuperClasse(proprietaSup: "ProSup")
let MiaIstanzaSub = MiaSubClasse(proprietaSub: "ProSub",proprietaSup
    :MiaIstanzaSup.proprietaSup)
print(MiaIstanzaSub.metodoSup())



// **********************
// ESTENSIONI E PROTOCOLLI
// **********************
// ESTENSIONI
// Consentono di estendere le funzionalità di un tipo esistente di classe,
struttura o enumerazione
// Presso GitHub è disponibile una raccolte di extension per tipi e classi
Swift
var miaStr:String = "Ciao "
miaStr.lowercaseString

extension String {
    var addCopyright: String{
       return self+"2015 Copyright"
    }
}
miaStr.addCopyright

// PROTOCOLLI
// Lista di metodi e proprietà che definisce un “contratto” o “interfaccia”
// che verrà implementato dalle classi che adottano il protocollo
protocol mioProtocollo {
    // Le proprietà del protocollo devono dichiarare getter e setter
espliciti
    var proprietaProt:Int {
        get
        set
    }
    func metodoProt(x:Int,y:Int)-> Int
}
//Ogni classe conforme al protocollo deve implementare i metodi del
protocollo stesso, pena un errore di compilazione
class miaClasse: mioProtocollo {
    var proprietaProt: Int = 3
    func metodoProt(x:Int,y:Int) -> Int{
     return x+y
    }
}
var istanzaClasse = miaClasse()
istanzaClasse.metodoProt(3,y:6)
```

```
// **********************
// STRUTTURE
// **********************
// Le strutture sono definite con la parola chiave struct.
// Tutto ciò che è possibile fare con una classe, si può fare con una
struttura, con alcune differenze importanti, tra cui:
// 1. le strutture non supportano l'ereditarietà
// 2. Swift fornisce automaticamente un costruttore alla struct
struct mioProdotto {
 var nome : String
 var prezzo : Float
}
var p1 = mioProdotto(nome: "Caffe", prezzo: 12.20)
print ("\(p1.nome) al prezzo di \(p1.prezzo) €")
p1.prezzo = 13.00
print ("\(p1.nome) al prezzo di \(p1.prezzo) €")

struct punto{
    var x:Float
    var y:Float
}


// GENERICS
// Le funzioni generiche utilizzano segnaposto invece di un tipo effettivo.
// Nella funzione, il segnaposto è T. È possibile utilizzare qualsiasi
parola come segnaposto ma è convenzione usare T.
// T sarà sostituito con un tipo effettivo determinato quando verrà
chiamata la funzione.
func stampaItemInArray<T>(a: [T]) {
    for item in a {
        print(item)
    }
}
var arrayStr = ["A","B","C","D"]
stampaItemInArray(arrayStr)
var arrayInt = [0,1,2,3]
stampaItemInArray(arrayInt)


// **********************
// OPTIONAL
// **********************
/* Optional consiste nella possibilità che una variabile possa contenere o
un metodo/funzione possa restituire un valore del tipo specificato o possa,
opzionalmente, non restituire alcun valore o meglio il valore nil. Quindi
una variabile Optional aggiunge all'insieme dei suoi possibili valori il
valore nil (non assegnato). In Swift nil non è il puntatore ad un oggetto
non esistente come in altri linguaggi, ma è un valore che indica l'assenza
```

```
di dato
 // In pratica l'optional è un'enumerazione così strutturata:
 enum optional{
    case nil
    case tipo
 }
 */
// Per dichiarare il tipo optional si utilizza il simbolo ? dopo il tipo
obbligatorio
var mioOptional:String?
print("Il valore di mioOptional è: \(mioOptional)")
// - Unwrap implicito: si è certi che il valore non sia nil
var mioOptional2:String!
//mioOptional2="Ciao"
print("Il valore di mioOptional2 è: \(mioOptional2)")

// Se proviamo ad utilizzare un valore che "potrebbe" non essere corretto,
ad esempio tentando di convertire una String in Int
var miaVar1 = "A"
var miaVar2 : Int = Int(miaVar1)
// il compilatore ci informa che occorre scegliere tra le seguenti opzioni:
// - "forzare l'unwrap" della variabile per vedere cosa contiene
effettivamente usando !
// - rimandare l'unwrap definendo Optional la nuova variabile usando ?
var miaVar2 : Int? = Int(miaVar1)
// Provando ad utilizzare il valore occorre verificarne la corretezza
if (miaVar2 != nil) {
    var miaVar3 = miaVar2! + 22
    print("Mia var 3 è utilizzata: \(miaVar3)")
}else{
    print("Mia var 3 NON è utilizzata")
}
// L’optional binding è la sintassi proposta per eseguire l’unwrap
indicando:
// 1. il blocco da eseguire se la variabile non è nil
// 2. il nome di una nuova variabile, risultato dell’unwrap
if let _miaVar2 = miaVar2 {
    let _miaVar2 = miaVar2! + 22
    print(_miaVar2)
}else{
    // Fai qualcosa
}

// - Optional chaining
// Consente di chiamare metodi su oggetti potenzialmente nil
// In presenza di una variabile optional è possibile richiamare metodi o
proprietà della variabile stessa senza eseguire l’unwrap
// Sarà sufficiente usare il carattere punto interrogativo (?) dopo il nome
della variabile
class Casa {
    var garage: Garage?
```

```
}
class Garage {
    var numAuto: Int?
}
// Normalmente si potrebbe verificare il numero di garage per una casa
così:
let casa = Casa()
let garage = casa.garage
if let garage = garage {
    // La casa dispone di garage
    if let numAuto1 = garage.numAuto {
     // Una o più auto nel garage
    } else {
     // Nessuna auto in garage
    }
} else {
    // La casa non ha garage
}
// Usando l'optional chaining è possibile velocizzare le operazioni:
let casa2 = Casa()
let numAuto2 = casa2.garage?.numAuto
if let numAuto2 = numAuto2 {
    // Una o più auto nel garage
} else {
    // La casa non ha garage
}

/*
// **********************
// DELEGATI
// **********************
// Un delegato è una classe che implementa un protocollo che si vuole
informare su eventi o a cui si chiede di svolgere alcuni compiti
class Delegatore {
    var del:Sveglia?
    func uso(){
       del?.Svegliati()
    }
}
var id=Delegatore()
id.uso()
*/


// **********************
// ASSERTIONS
// **********************
// Un’asserzione è un costrutto che al verificarsi di una condizione
segnala l’anomalia e quindi termina l’esecuzione del programma con un
errore.
// le assertions sono molto utili durante la fase di debug per verificare
```

che i parametri di una funzione siano sempre valorizzati correttamente.

```
let eta2 = -1
assert(eta2>=0, "L'età non può essere minore di zero")
```