C# Java PHP Python

*# La guida **completa** alla* programmazione ad oggetti

Un libro per imparare velocemente 4 tra i migliori linguaggi per lo sviluppo Web lato server con esercizi e tanti esempi

TONY CHAN

C# Java PHP Python

La guida **completa** alla programmazione ad oggetti.

Un libro per imparare velocemente 4 tra i migliori linguaggi per lo sviluppo Web lato server con esercizi e tanti esempi.

Di ***Tony Chan***

Avviso di esclusione di responsabilità:

Si prega di notare che il contenuto di questo libro è solo a scopo educativo e d’intrattenimento. È stato compiuto ogni sforzo per presentare informazioni accurate, aggiornate, affidabili e complete. Nessuna garanzia di alcun tipo è dichiarata o implicita. I lettori riconoscono che l'autore non s’impegna a fornire consulenza legale, finanziaria, medica o professionale. Il contenuto di questo libro è in parte derivato da varie fonti. Consultare un professionista autorizzato prima di tentare qualsiasi tecnica descritta.

Leggendo questo testo, il lettore accetta che in nessun caso l'autore sarà ritenuto responsabile per eventuali perdite, dirette o indirette, subite a seguito dell'uso delle informazioni contenute in questo documento, incluse omissioni o inesattezze.

L’autore

Tony Chan è nato a South Gate, in California, Stati Uniti, da genitori immigrati cinesi. A 11 anni scrive il suo primo programma software in Basic, a 17 insegna agli amici programmazione e a 20 si laurea in informatica. Recentemente ha partecipato come consulente nella stesura di alcuni testi sull’intelligenza artificiale e sul linguaggio macchina. Ha poi deciso di pubblicare i suoi libri sulla programmazione sia negli Stati Uniti, dove vive e lavora, ma anche in Italia, che ama e che visita spesso. Un giorno, un ristorante di Roma gli ha dedicato la “*Tecno Carbonara*”, dove il guanciale era stato tagliato e modellato a forma di CPU e Microprocessori.
Di Tony si dice che un giorno abbia composto il famoso jingle di Star Wars sfruttando il rumore prodotto da alcuni vecchi dischi rigidi meccanici.
E’, tra l’altro, autore di due testi in Italia che nel 2022 hanno avuto un buon successo: ***Javascript, la guida definitiva*** e ***HTML5 & CSS3, la guida completa.***

 # Indice dei contenuti #

 # Introduzione
 # 1 - Le basi
Si parte!
I dati
Le costanti e le variabili
I metodi
Le variabili in Python
I primi passi con Python
Convertire i dati in Python
Operatori
Le variabili in C# e Java
Provare il codice con Java
Provare il codice con C#
Le variabili in PHP
Tipi di dati e typecasting in PHP
Provare il codice con PHP
Le iterazioni
Quiz & esercizi – riassunto
 # 2 - La programmazione OOP
Classi ed ereditarietà
Le classi in PHP
Polimorfismo, Get e Set in Java
Classi e oggetti in C#
Inizializzare le variabili
Il costruttore al lavoro
Sovraccarico dei metodi
Modificatori di accesso
Polimorfismo, ereditarietà e modificatori di accesso in C#
Classi e oggetti in Python
Ereditarietà e polimorfismo in Python
Gli oggetti in PHP
Ereditarietà e polimorfismo in PHP
Il costruttore e il distruttore in PHP
Il multithreading nella programmazione
Quiz & esercizi – riassunto
 # 3 - Approfondiamo C#

Le impostazioni locali
Typecasting
La gestione del tempo
La gestione dei file
FileInfo
L'imbarazzo della scelta
Gestione della memoria RAM
Multithreading
Sincronizzare i thread
I metodi Yield e Join
Programmazione asincrona
Quiz & esercizi – riassunto
 # 4 - Approfondiamo JAVA
Lo scope
Il blocco try catch e finnaly
La gestione delle eccezioni
La gestione dei file
Il multithreading
Le proprietà di blocco
I metodi dei thread
Misurare il tempo
La classe Locale
Typecasting
L'istruzione Switch
La ricorsione
Quiz & esercizi – riassunto
 # 5 - Approfondiamo Python
Le tuple
La gestione delle eccezioni
La gestione dei file
Operazioni sulle directory
La gestione del calendario
Le espressioni regolari
Metacaratteri
Multithreading?
Multithreading!
Multitasking cooperativo e asyncio

Funzioni anonime o lambda
Metodo Zip e archivio Zip
La libreria NumPy
Quiz & esercizi - riassunto
 # 6 - Approfondiamo PHP
Operazioni sui file
Funzioni
Il costruttore
Sessioni e verifica
Il multithreading
Usiamo Pthread
Usiamo Parallel
Quiz & esercizi – riassunto
 # 7 - Ripasso generale
Usare file XML in PHP
Decodificare jSON in PHP
Il simulatore di chat per Python
La sequenza di Fibonacci in C#
Operazioni sui file e threading con C#
L'accesso e la gestione dei file in Java
Un cronometro per Java
Quiz & esercizi – riassunto

 # Bibliografia

Introduzione

Questo libro vuole essere un'introduzione e una guida ai più importanti linguaggi di programmazione attualmente utilizzati nel mondo. Ho già trattato in un altro bel libro quello che per mia opinione personale è il numero uno, sto parlando di Javascript. In questo testo parleremo degli altri che seguono.

Ottimo inizio, ho appena acquistato un libro inutile, inferiore, non all'altezza.. penserà qualcuno. In realtà bisogna subito specificare per bene un paio di cose: innanzitutto stampiamoci in mente che non siamo noi che dovremo trovare il "mezzo" ottimale per ottenere la realizzazione di un eccellente programma, ma viceversa, sarà in base al programma da realizzare che dovremo imparare a scegliere il miglior linguaggio di programmazione utile allo scopo. Tendenzialmente per integrare un database nella nostra pagina Web prenderemo in considerazione il PHP, per scambiare dati tra sistemi diversi ci serviremo di XML, Python andrà bene quasi per tutto, soprattutto il machine learning, con Java avremo il vantaggio di poter avere un'applicazione multipiattaforma grazie alla compilazione, penseremo al C# in caso il nostro progetto sia quello di creare un programma per la piattaforma Windows, e così via. Sostanzialmente queste scelte ci renderanno la vita, cioè la programmazione molto più semplice risparmiando tempo e fatica.

Inoltre le classifiche sul miglior codice per programmare indubbiamente lasciano un po' il tempo che trovano; però hanno anche una loro dignità, nel senso che statisticamente, se la maggior parte degli addetti ai lavori, mediamente considera Javascript, Java, Python, C#, PHP e pochi altri i migliori sul mercato sarà una buona idea quella di fidarsi di loro.

In questo libro abbiamo deciso di proporvi i quattro linguaggi di programmazione più popolari e versatili del 2022/23. Saranno presentati e discussi in dettaglio diversi concetti relativi al coding. Anche chi parte da zero può iniziare questo percorso con il vantaggio di non avere preconcetti rispetto a chi, ad esempio, viene da Javascript, quindi con un'impostazione sulla programmazione legata alle funzioni, mentre Java e C# sono più per le classi. Lo vedremo più avanti, e inoltre ci renderemo conto che tutte queste nozioni saranno relativamente semplici da apprendere e comprendere, e una volta afferrate, ci condurranno in un percorso speciale, il quale ci porterà direttamente nel cuore del nostro dispositivo, computer, smartphone, smartwatch, tablet e fra qualche anno frigorifero, lavatrice e così via.

Inizieremo subito presentando questo fantastico poker: C#, PHP, Python e Java.

C# progettato da Anders Hejsberg, inizialmente definito una brutta copia del linguaggio Java (in realtà è molto più simile al C++) è stato rilasciato al pubblico nel Gennaio del 2002. Sviluppato da Microsoft con un team di cui a capo c'era proprio Hejsberg, era basato su .NET Framework 1.0. Il linguaggio può essere utilizzato per creare qualsiasi tipo di applicazione moderna, partendo dal software di produttività fino ai videogiochi. C# condivide alcune delle sue caratteristiche con molti altri popolari linguaggi di programmazione essendo OOP, cioè, come ci ricorda l'acronimo stesso: Object Oriented Programming, ossia orientato agli oggetti. Nello specifico "oggetto software" è una definizione metaforica la quale serve anche a farci capire la differenza sostanziale del modello di programmazione incentrato sui dati piuttosto che sulle procedure, il quale, di fatto già dagli anni '80, ha semplificato la progettazione delle applicazioni.
Solitamente, quando si parla del linguaggio C si fa un po' confusione con il C++, che è un linguaggio precedente e un po' più complesso rispetto a C#. Sebbene C++ sia in grado di produrre un output più efficiente in generale, con C# è più semplice sviluppare applicazioni per dispositivi mobili e il Web. Per quanto riguarda il C, parliamo di un linguaggio che risale al 1972 e che è di natura procedurale, quindi molto più complicato rispetto agli altri due. Necessita sicuramente di un testo a se dedicato.

PHP inizialmente era l'acronimo di Personal Home Pages, poi modificato in Hypertext Preprocessor, cioè preprocessore di ipertesti, essenzialmente un programma che interviene sul codice sorgente di un altro programma, per miglioramenti o aggiunte. Linguaggio creato da Rasmus Lerdorf nel 1995 pensato per l'implementazione di pagine Web, si è poi evoluto come un codice completo negli anni seguenti.
Il PHP è un linguaggio di scripting dal lato server, ha una sintassi derivata dal C e spesso è integrato nel HTML di una pagina Web.
In questo periodo, vista la sua facilità d'uso e flessibilità il PHP è spesso utilizzato per creare blog o forum, nel web, database o anche software come Joomla! e Wordpress.
S'integra alla perfezione con MySQL e con molti altri DBMS, cioè Database

Managment System, tradotto: sistema di gestione dei database, quindi essenzialmente un software per crearli e amministrarli.

Python sviluppato da Guido Van Rossun con l'idea di migliorare il linguaggio ABC. Prende il suo nome da Monty Python, un popolare programma comico televisivo in Inghilterra. E' stato rilasciato nei primi mesi del 1991. Sebbene l'output di Python sia spesso più lento del software realizzato con C#, è comunque diventato piuttosto popolare negli ultimi anni, perché è gratuito, multipiattaforma, facile da usare e imparare, ricchissimo di librerie e molto flessibile. Sostanzialmente può gestire qualsiasi cosa, dalla creazione di semplici applicazioni per il Web a software desktop più pesanti. Python ha persino trovato una nicchia nell'industria dei videogiochi ed è sempre più utilizzato nel machine learning, cioè quel particolare settore relativo all'Intelligenza Artificiale in cui un software è in grado di imparare e migliorare le proprie performance in base ai dati utilizzati. Ciò che è degno di nota di Python è il modo in cui è sensibile a spazi vuoti e ai rientri (i caratteri invisibili creati premendo la barra spaziatrice e/o il tasto Tab). Impareremo quest'approccio unico e altro più avanti nel libro.

Java ideato dalla Sun Microsystems la quale creò un gruppo guidato da James Gosing nel 1992. Inizialmente chiamato Oak, poi fu in seguito rinominato in Java per problemi di copyright. Rilasciato ufficialmente a metà del 1995, è diventato rapidamente un linguaggio di programmazione generico ampiamente adottato soprattutto per l'ambiente online (ad esempio, videogiochi, big data, cloud computing). Oggi, il software scritto con Java alimenta innumerevoli smartphone, data center e servizi online onnipresenti come quelli di Google. In questi ultimi anni si è contraddistinto tra i linguaggi di programmazione più popolari sia per i principianti sia per i programmatori esperti, anche per i suoi costanti aggiornamenti, giunti alla versione otto a Maggio del 2022.
Sebbene il linguaggio di programmazione noto come JavaScript condivide le sue prime quattro lettere con Java, entrambi hanno pochissimo in comune. JavaScript è stato creato dai pionieri di Internet Netscape negli anni novanta per l'utilizzo con la nuova tecnologia dei browser. Il linguaggio è ancora vivo e vegeto su Internet, fornendo funzionalità e usabilità extra per una miriade di siti Web.

Compilazione e interpretazione del codice

Il linguaggio, o codice macchina, è ciò che i computer intendono quando si esegue un programma, proprio come la lingua inglese per gli inglesi. E' composto di lunghe serie di uno e zero. Nella programmazione esiste una distinzione tra linguaggi di alto livello e di basso livello, niente a che vedere con le parolacce ovviamente. Essenzialmente la classificazione si riferisce a quanto più vicino al codice macchina è un linguaggio di programmazione, le lingue di basso livello si riferiscono proprio a questo caso, mentre quelle di livello alto, al contrario, tendono ad avvicinarsi al nostro linguaggio. C# e Python sono considerati linguaggi di alto livello, mentre C++ rappresenta quello che è noto come un linguaggio di livello medio poiché offre alcune funzionalità piuttosto solide per i programmatori più avanzati. Questa distinzione serve a farci capire gli sforzi che sono stati fatti per creare dei linguaggi di alto livello per permettere a tutti di avvicinarsi al mondo della programmazione, ma anche per fare in modo che gli sviluppatori possano concentrarsi più sulla propria idea evitando perdite di tempo per assegnare risorse alla ram o alla cpu, per esempio, perché bisognava fare questo, e molto altro, nei linguaggi di basso livello, come l'Assembly.

Inoltre, è importante ricordare che i computer non possono eseguire immediatamente alcun linguaggio di programmazione. Ad esempio, Java e Python sono entrambi cosiddetti linguaggi interpretati. Ciò significa che ogni riga di un elenco viene, appunto, "interpretata" in codice macchina in tempo reale. Il processo d'interpretazione comporta una velocità di esecuzione più lenta rispetto alla varietà cosiddetta compilata.

In pratica i listati di molti linguaggi di programmazione devono essere sottoposti a un processo chiamato compilazione prima di poterli eseguire. In poche parole, ciò si riferisce alla traduzione completa del codice sorgente di un programma in linguaggio macchina prima della sua esecuzione. Per fortuna il processo di compilazione è automatizzato, quindi non sono necessari dizionari per l'interpretazione del codice da programmatore a macchina. È una funzione di qualsiasi buon ambiente per lo sviluppo del codice (Cioè un IDE, più avanti ne segnaleremo qualcuno).

Una volta che il nostro programma sarà compilato, in teoria potrà essere eseguito senza difficoltà solamente nell'ambiente in cui è stato elaborato. Essenzialmente ogni applicazione nata, ad esempio, in Windows, non potrà essere utilizzata con Linux o MacOS e viceversa. Inoltre la compilazione

non rende il nostro programma del tutto indipendente. Per esempio, un assembly in C# si riferisce all'output del progetto, come in un file eseguibile dall'utente di solito con l'estensione di file .exe nell'ambiente Windows (es. lamiaapp.exe). Gli assembly di solito contengono altre risorse utilizzate da un programma, inclusi i dati d'immagine. Ospitano anche i metadati del nostro progetto, come le informazioni sulla versione e potenzialmente un elenco di quali altri assembly sono necessari per eseguire il programma; progetti più grandi possono essere costituiti da numerosi assembly.
Infine, poiché ne parleremo più avanti, apriamo una piccola parentesi sulla differenza tra 32 o 64 bit, la quale essenzialmente sta nell'utilizzo della ram del computer, nei vecchi sistemi a 32 bit potremo usufruire solo di un massimo di 2 GB per programma, mentre con i più recenti software a 64 bit ne avremo a disposizione da 4 GB in su.

Strumenti (Tools) di lavoro

Sebbene, in linea teorica, per scrivere del codice con Java, PHP, Python, o C# sia sufficiente disporre di un editor di testi (come il blocco note), quando si comincia a scrivere codice in modo un po' più "serio", gli strumenti giusti possono fare davvero la differenza.
Molto importante è la distinzione tra un IDE (*ambiente di sviluppo integrato*) e un editor di testo, infatti, il primo ci permetterà, oltre alla semplice stesura del listato, la creazione vera e propria di un programma a se stante, per mezzo della cosiddetta "compilazione", che renderà la nostra applicazione indipendente e funzionante, senza bisogno di un software di supporto. Mentre l'editor di testo è molto più limitato, serve solo a scrivere del codice ed eventualmente testarlo.
Senza la pretesa di essere esaustivi, nelle pagine che seguono, proporremo una panoramica degli strumenti (gratuiti) più diffusi con cui potremo scrivere listati e compilarli, in tutti e quattro linguaggi che studieremo. Andiamo a vederli nel prossimo paragrafo.

Editor di testo e Linter

Una prima alternativa al Blocco Note è costituita degli editor di testo "più evoluti". Ne ricordiamo tre: Notepad ++, Atom e Visual Studio.
Il primo di questi, **Notepad ++** è un editor gratuito che può essere scaricato dal sito https://notepad-plus-plus.org/.
La sua interfaccia è semplice e di facile utilizzo. Dispone di una serie di

caratteristiche davvero utili: evidenziazione della sintassi, raggruppamento di porzioni omogenee di codice (*Syntax Folding*) in modo da poter nascondere o visualizzare porzioni di un documento lungo, evidenziazione della sintassi e *Syntax Folding* personalizzato dall'utente, evidenziazione delle parentesi, ricerca/sostituisci mediante espressioni regolari (Perl Compatible Regular Expression), autocompletamento della sintassi, segnalibri, visualizzazione a schede, esposizione di documenti affiancati per il confronto.

Atom è un editor gratuito scaricabile dal sito https://atom.io/ disponibile per più piattaforme (OS X, Windows e Linux). Può essere completato con diversi pacchetti open source e dispone del supporto al sistema di controllo della versione Git.

Fra i punti di forza di Atom ci sono: autocompletamento, evidenziazione della sintassi, funzionalità di ricerca e sostituzione fra più file, possibilità di aprirne diversi in pannelli affiancati per poterli confrontare.

Visual Studio Code è l'editor che abbiamo usato per scrivere gli esempi di questo libro. È un editor sviluppato da Microsoft per più piattaforme (OS X, Windows e Linux). Si tratta di uno strumento gratuito scaricabile dalla pagina https://code.visualstudio.com/.

Dispone già di Git ed è integrabile con ulteriori pacchetti. Fra i suoi punti di forza ci sono: autocompletamento, evidenziazione della sintassi, funzionalità di ricerca e sostituzione, possibilità di impostare breakpoint, lavorare direttamente con file e cartelle senza la necessità di creare progetti.

Il **Linter** è un programma che in genere s'integra con un editor di codice e permette di evidenziare gli errori di sintassi o in generale di scrittura del codice. Ogni linguaggio ne ha diversi a disposizione. Per Python segnaliamo Pylint (https://pypi.org/project/pylint/), per Java, tra i tanti suggerisco Checkstyle (https://checkstyle.sourceforge.io/), per C# abbiamo delle ottime impressioni con SonarLint (https://www.sonarlint.org/), infine il PHP ha a disposizione PHPlint (https://www.npmjs.com/package/phplint).

AppStore: oltre ai software elencati, in tutti i dispositivi aggiornati con Windows, MacOS, iOS e Android, negli appstore dedicati esistono numerosi editor gratuiti anche in prova, più o meno validi, comodi e veloci, da utilizzare per esercitarsi e studiare. Vale la pena darci un'occhiata per trovare quello che fa per noi.

IDE

Come abbiamo già scritto, per programmare seriamente non potremo in alcun modo fare a meno di un IDE, ossia di un ambiente di sviluppo integrato (*Integrated Development Environment*). Praticamente stiamo parlando di un software che ci servirà per scrivere un listato, per testarlo ed eventualmente compilarlo, per renderlo eseguibile e indipendente. Di seguito ne elencheremo alcuni utili per i nostri esperimenti e per la realizzazione delle applicazioni.

Per quanto riguarda C#, potremo usufruire di Visual Studio per sviluppatori .NET (https://visualstudio.microsoft.com/it/), degno di menzione è anche Xamarin, un framework per VisualStudio sempre della Microsoft (https://dotnet.microsoft.com/en-us/apps/xamarin), eccellente poiché multipiattaforma.

Un buon ambiente per il PHP è Eclipse (https://www.eclipse.org/) ma anche XAMPP, che include MySQL (https://www.apachefriends.org/it/index.html).

Con Python avremo l'imbarazzo della scelta. Sono ottime opzioni: PyCharm (https://www.jetbrains.com/pycharm/), PyDev (https://www.pydev.org/).

Anche per Java andrà bene Eclipse (https://www.eclipse.org/) e soprattutto l'IDE ufficiale dell'Oracle, NetBeans: https://netbeans.apache.org/.

Prima di iniziare dovremo anche ricordarci di scaricare il codice sorgente da: https://java.com/it/, https://www.python.org/, https://www.php.net/ e https://dotnet.microsoft.com/en-us/download per C#. Inoltre sui numerosi canali social e le infinite community avremo praticamente tutte le informazioni necessarie per iniziare a programmare, anche se l'ambiente utilizzato non è quello standard, cioè Windows o MacOS, ma Linux o altri.

In conclusione vorrei segnalare che, anche se di questi software per l'ambiente di sviluppo ne esistono tantissimi altri, più o meno validi, inizialmente non converrà perdere troppo tempo sulla loro scelta, soprattutto per chi è alle prime armi. In seguito, con l'esperienza o il consiglio di qualche collega, potremo sbizzarirci per sperimentarne altri.

Chi avesse dei problemi con la lingua inglese potrà sempre abusare dell'utilizzo di Google Translate o di qualsiasi altra applicazione simile, soprattutto quelle integrate nel browser, le quali ci tradurranno istantaneamente qualunque pagina Web.

L'ambiente di lavoro

L'ambiente standard che abbiamo utilizzato per questo libro è Windows,

anche tutti i software suggeriti hanno come base l'OS Microsoft. Ovviamente sappiamo tutti che MacOS e Linux sono altrettanto validi come sistemi operativi. Se alcuni programmi non dovessero essere disponibili per il vostro ambiente di lavoro, andrà benissimo cercarsi un'alternativa con Google, o nelle varie piattaforme dedicate alla programmazione.
A parte ciò va detto che anche i linguaggi di programmazione hanno il loro ambiente, ad esempio le applicazioni Java e PHP utilizzeranno maggiormente il browser. C# è più legata a Windows e alla piattaforma .NET, mentre Python non ha un ambiente preferito, è più orientato a essere multipiattaforma. Tuttavia, una volta compilati, lo stesso varrà anche per C# e Java. Anche se, un programma di quest'ultimo, nonostante la compilazione, non sarà subito eseguibile, necessiterà della *Java Virtual Machine* (JVM), un pacchetto software generalmente presente su tutti i dispositivi. Quest'approccio nell'esecuzione di Java su una macchina virtuale dedicata crea quindi un grande grado d'indipendenza dalla piattaforma. Java è un linguaggio interpretato. Quando si esegue un programma, il compilatore genera bytecode. Questo è un formato intermedio che richiederà la *Java Virtual Machine* (JVM) per essere eseguito; non è possibile avviare un programma bytecode senza di essa. Infine, abbiamo *Java Runtime Environment* (JRE). Questo componente è utilizzato per combinare l'output del codice eseguito all'interno del JDK con alcune librerie software aggiuntive richieste, consentendo l'esecuzione effettiva dei programmi Java. Ovviamente, anche per tutti gli altri, come già detto, servirà installare le librerie di base dai rispettivi siti ufficiali.

Server virtuale

Per i nostri esperimenti, qualche volta, avremo bisogno di simulare un server. L'alternativa è quella di pagare un software professionale. Comunque, come abbiamo già segnalato precedentemente, per iniziare andrà benissimo utilizzare un servizio gratuito e open source come quello offerto da XAMPP (https://www.apachefriends.org/it/index.html).
Sarà sicuramente un'ottima opzione rispetto all'installazione di Apache e all'eventuale aggiunta di MySQL e PHP.

UML

Con progetti software più grandi, ma non solo, è spesso una buona idea servirsi di carta e penna, per così dire, prima di scrivere una riga di codice.

Uno degli strumenti migliori e più popolari per visualizzare i software orientati agli oggetti che tratteremo nel libro consiste nell'utilizzare *l'Universal Modeling Language* (UML). Creato da Rational Software a metà degli anni '90, UML da allora è diventato uno strumento onnipresente nell'ingegneria del software. Essenzialmente serve a creare diagrammi di attività, o processi, come i diagrammi di flusso. Sul Web ne esistono tantissime versioni, alcune gratuite, come ad esempio StarUML (https://staruml.io/download) o UMLet (https://www.umlet.com/). Non è un software obbligatorio per studiare, tuttavia potrebbe rivelarsi molto utile. Dedicategli qualche minuto, potreste non farne più a meno.

Debug e Boilerplate code

La correzione degli errori nei nostri listati sarà quella parte di tempo che non vorremmo mai sprecare, ma con cui dovremo spesso avere a che fare. Tecnicamente si chiama debug del codice, e ora andremo a dare uno sguardo ad alcuni dei metodi più comuni per eseguirlo al meglio:

- *Debug con tracciamento*: Questo metodo significa semplicemente prestare molta attenzione ai risultati di ogni riga di codice eseguita visualizzandoli sullo schermo, uno per uno. È possibile prestare molta attenzione al contenuto delle variabili e di altre strutture di dati poiché sono alterate durante l'esecuzione di un programma. Il tracciamento è qualcosa che funziona alla grande per progetti più piccoli, come i tutorial all'interno di questo libro.

-*Debug riproduci e registra*: Con questo approccio, parti dell'esecuzione di un programma vengono registrate e riprodotte per esaminarne i potenziali difetti. Questo non si riferisce alla riproduzione esterna o visiva del software; piuttosto si concentra sui procedimenti a livello statale all'interno di un programma.

- *Debug post-mortem*: questo metodo consiste nell'analisi dei dati del registro post-arresto anomalo su un programma. Molti tipi di software scrivono file di registro su disco dopo gravi malfunzionamenti, inclusa la maggior parte dei sistemi operativi. Questi file possono quindi essere esaminati alla ricerca di indizi su quali bug hanno causato un arresto anomalo.

- *Debug da remoto*: Il debug non deve essere eseguito sul dispositivo che esegue il programma di messa a fuoco. Utilizzando i metodi di rete più diffusi, come il Wi-Fi o il cablaggio USB, è possibile collegare dispositivi

con ruoli e fattori di forma diversi per lavorare insieme. Farlo in remoto è l'approccio più comune durante la scrittura e il debug di software per Android e iOS, poiché la macchina di sviluppo è quasi sempre un computer separato di dimensioni standard.
- *Debug con raggruppamento*: Questo è un approccio utile ogni volta che viene trovata una quantità particolarmente grande di bug. Il programmatore prima identifica tutte le caratteristiche comuni nei bug. I problemi sono quindi classificati in gruppi specifici che ne condividono gli attributi, con la logica che, anche se sono risolti alcuni bug all'interno di un cluster, il resto dovrebbe seguire.
- *Debug tramite semplificazione del codice*: A volte la migliore strategia per eliminare i bug sarà quella di rimuovere parti di codice funzionante dalle aree interessate dagli errori. Questo ovviamente funziona meglio per i bug di natura più nascosta. Quando non siamo ancora sicuri di cosa non funziona (ad esempio, cosa sta causando un arresto anomalo), rimuovi le parti che ovviamente funzionano, una per una, e attira il bug.
Oltre agli errori tecnici inerenti il codice dovremo anche prestare attenzione alla parte estetica, o meglio, per usare un termine che ha più a che fare con la moda, evitare un design ridondante e antiestetico. In inglese è meglio definito con: *boilerplate code*, cioè parti di codice ripetute più volte apparentemente senza cambiamenti. In alcuni casi potremmo risolvere il problema con un'aggiustatina al listato, ma, a volte non sarà possibile eseguire il cosiddetto *refactoring*, cioè la modifica di parti del codice senza cambiarne la struttura principale, quindi fin dalle prime fasi sarà nostra premura fare molta attenzione a non incorrere nel *boilerplate code*, per evitare di dover riscrivere il programma da capo.

Commentare il codice
Nel corso del nostro studio, in qualunque linguaggio di programmazione ci dovessimo imbattere, avremo dei listati da interpretare o semplicemente da copiare. In molti di questi saranno presenti dei commenti per aiutare a comprendere meglio il metodo, la sintassi, la classe o la porzione di codice in esame. Ogni linguaggio, per permettere di integrare del testo avulso dal codice, ha una sua grammatica specifica: in Python dovremo utilizzare il simbolo del cancelletto # prima del commento, in C#, Java e PHP avremo il doppio slash //. Il PHP accetta anche il cancelletto. I commenti lunghi, cioè

quelli che superano la lunghezza della riga, saranno compresi tra /* e */ , come anche in C#.

 # 1 - Le basi

Le basi della programmazione
Le variabili, le costanti e i metodi
Le iterazioni

Inizieremo il nostro viaggio nel mondo della programmazione dei principali linguaggi esistenti introducendo passo dopo passo i concetti basilari per poi procedere verso argomenti più complessi. In questo paragrafo partiremo con le nozioni che sono le fondamenta di tutti e quattro i nostri linguaggi, ma anche di molti altri.

Si parte!

Per cominciare introduciamo le regole di base che sicuramente molti già conosceranno, ma è sempre bene ripassare.
Ognuno dei linguaggi che andremo a studiare ha delle sue regole sintattiche che dovremo imparare a conoscere bene e soprattutto a non confonderle tra di loro.
Python è stato pensato per avere una sintassi relativamente semplice e leggibile. Ogni istruzione sarà divisa semplicemente da una nuova riga, senza il classico punto e virgola che fa da separatore, come in Java o Javascript, inoltre, per definire un ambito, o delimitare blocchi di codice come quello di una funzione o di un loop, è utilizzato lo "*spazio bianco*", tradotto da "*whitespace*", pratica tecnicamente meglio conosciuta come **Indentazione**. Sostanzialmente lo spazio sostituisce le parentesi graffe sempre usate ad esempio con Java o Javascript.

Questo rende Python diverso da tutti gli altri ma anche più leggibile e subito comprensibile, vedremo più avanti come e perché. In Java invece la sintassi è stata ereditata dal C++, anche se sono molto diversi, ad esempio in quest'ultimo non sono presenti variabili locali e funzioni. Sempre in Java il codice è suddiviso in classi, dove i valori essenzialmente sono anche oggetti software, a parte i **tipi di dati primitivi** come ad esempio i valori booleani, cioè vero o falso, *true* o *false*, oppure i numeri interi (parleremo dell'argomento più avanti).

Per quanto riguarda il C#, le sue regole sono leggermente più complesse, innanzitutto vi è distinzione tra maiuscole e minuscole tra le istruzioni (ad esempio: *Console.WriteLine*) e anche le virgolette, che solitamente racchiudono una stringa, non potranno essere singole come in Javascript, quindi 'prova' non sarà ritenuto valido, al contrario di "prova".
Con PHP dovremo impostare un tag (*<?php*) all'inizio e alla fine (*?>*) di ogni listato. All'interno del codice ogni riga terminerà con il classico punto e virgola. Anche in PHP i nomi sono sensibili alle lettere maiuscole o minuscole.
Ci sono ancora tantissime altre sfumature e differenze sostanziali tra i vari linguaggi, ma adesso è bene non soffermarci troppo su questo punto, ci torneremo di volta in volta mentre proseguiremo nel percorso di studio. Useremo questa metodologia anche per evitare di rendere noiosa o pesante la lettura e per cercare di far acquisire i numerosi concetti gradualmente. Adesso entriamo in campo partendo dai dati, dalle variabili e dalle costanti.

I dati

Essenzialmente ogni programma è un'elaborazione di diversi dati. Tecnicamente all'interno della nostra memoria del computer i dati sono byte, in gruppi di otto bit, con un valore che va da 0 a 255. Possono essere anche usati insieme per rappresentare valori più consistenti. Sappiamo che il valore della grandezza aumenta fino a raggiungere i più odierni e conosciuti megabyte (un milione di byte) e gigabyte (un miliardo di byte).
Ad ogni byte può essere assegnato qualsiasi valore, come ad esempio una lettera o un colore. Da qui derivano gli standard, l'ASCII per il testo, JPG per le immagini, e così via. Ogni linguaggio di programmazione avrà il suo tipo di dato, che di solito, come già accennato precedentemente, consiste in: valori booleani (vero e falso), numeri interi o in virgola mobile e le stringhe. Approfondiremo tutte queste cose poco più avanti. In conclusione sono dati anche gli oggetti (software, come ad esempio una porzione di codice), o gli insiemi (raccolte di dati) che esploreremo nei prossimi listati. Adesso parliamo delle variabili e dei metodi.

Le costanti e le variabili

Le costanti che troveremo nei nostri listati sono dei dati non modificabili, i quali resteranno tali per tutta l'esecuzione del programma, al contrario delle variabili ovviamente. Entrambe sono una forma di archiviazione temporanea, disponibile per l'utente durante il tempo che trascorre con un

programma. Utilizzano in genere la memoria ad accesso casuale (RAM) nel dispositivo. Perciò, quando spengeremo il dispositivo, i dati scompariranno, salvo che non siano prima salvati su un dispositivo di archiviazione.
Una variabile potrebbe essere utilizzata per memorizzare il nome del giocatore in un videogame, per nominare un oggetto, per eseguire complicati calcoli o per visualizzare un grafico, quindi avrà svariate operazioni in cui sarà coinvolta. E anche il programmatore le utilizzerà più e più volte in un programma.
Ci sono numerosi tipi di variabili e costanti in ogni linguaggio di programmazione. In effetti, ne esistono separate con: stringhe di testo, singoli caratteri alfanumerici e diversi intervalli di valori numerici. Perché fare queste distinzioni? Ebbene, talvolta capiterà di dover memorizzare una semplice stringa, come ad esempio la parola "Tony", e in effetti, ragionandoci su, per un computer sarebbe un notevole spreco di memoria quello di riservare uno spazio destinato a valori numerici anche di 20 cifre con una semplice stringa di quattro lettere. Anche per questo motivo vi sono delle distinzioni tra le variabili e le costanti nella maggior parte dei linguaggi di programmazione. Molti di essi, ad esempio, non dispongono di variabili universali in grado di memorizzare qualsiasi tipo d'informazione. Nella maggior parte dei casi, il programmatore dovrà definire le dichiarazioni delle variabili (e anche delle costanti) prima dell'uso, vedremo come tra breve.
Parliamo adesso dei metodi, cioè di alcune operazioni legate solitamente agli oggetti software, quindi ad esempio alle classi, le quali essenzialmente ci serviranno soprattutto per creare altri oggetti, tutti con delle istruzioni e dei dati utili a svolgere un determinato compito, cioè lo scopo finale del programma.

I metodi

I metodi ci serviranno per eseguire operazioni con gli oggetti o le istanze delle classi. Essenzialmente sono istruzioni specifiche che permettono alle nostre applicazioni di essere avviate. Hanno sintassi e modalità differenti tra i vari linguaggi e si dividono in:

- *metodi astratti*, come per le classi astratte sono utili per l'ereditarietà.
- *metodi di estensione*, li conosciamo soprattutto in C#, utili anche per la migliore leggibilità del codice.
- *metodi di istanza*, sono tutti quei metodi invocati con riferimento ad un oggetto.

- *metodi statici*, in questo caso non si riferiscono ad un solo oggetto ma a tutta la classe.
- *costruttori*, usati per evocare un nuovo oggetto.
Li affronteremo tutti in maniera molto più dettagliata nel prosieguo del libro.

Le variabili in Python

Tecnicamente in Python la variabile è una posizione all'interno della memoria di un dispositivo (smartphone o computer) in cui archivieremo dei dati. E' composta dal nome, dal valore e ha sei tipologie differenti: set, dizionari, numeri, stringhe, liste e tuple. A differenza di altri due linguaggi che studieremo, cioè Java e C#, in Python (e anche PHP) spesso le variabili non richiederanno la specifica della tipologia del dato. Sarà automaticamente definito nel momento della dichiarazione.
Andiamo a vedere nel dettaglio le differenze tra le varie tipologie:

- **Set** è una raccolta di valori non ordinati, inizializzati utilizzando parentesi graffe. I valori duplicati non saranno considerati: num = {7, 3, 7, 2, 7}
- **Dizionario** composto da coppie *chiave-valore* non ordinate, anch'essi inseriti tra le parentesi graffe: Auto = {'mod': 'Z3', 'velocità': 230}
- **Numeri** includono numeri negativi, positivi, complessi e in virgola mobile. I calcoli potranno essere eseguiti al momento della creazione delle definizioni, ad esempio: età_Mario = 45 o: Risultato = 3,66 – 5,12
- **Stringhe** sono insiemi contigui di caratteri alfanumerici. Sia le virgolette singole sia le doppie sono accettate: nome = "Mario"
- **Liste** sono costituite da valori o variabili di qualsiasi tipo. Racchiusi tra parentesi quadre, le liste per le stringhe avranno le virgolette singole: Lista_prova = [55, 'Gatti', 22,'Cani']
- **Tuple** sono delle raccolte come le liste, ma sono di sola lettura e quindi non potranno essere aggiornate o modificate. Sono racchiuse tra parentesi:

```
La_mia_prima_Tupla = (55, "Cavalli")
```

Infine non dimentichiamoci delle **costanti** che seguono una regola piuttosto semplice, dovranno essere dichiarate in lettere tutte maiuscole: GATTI = 3

I primi passi con Python

Dopo tutta questa teoria sicuramente utile, ma anche un po' noiosa, è finalmente arrivato il momento di fare sul serio! Iniziamo eseguendo un listato molto semplice per scrivere su schermo una stringa di testo:

```
   Sport = 'pugilato'
print("Quest'anno praticherò il:", Sport)
```

Notiamo come la virgola separi la variabile dalla stringa. Proviamo questo:

```
   Sport = {'pugilato', 'calcio', 'nuoto'}
print("Quest'anno praticherò questi sport:", Sport)
```

Per chi proviene da altri linguaggi, noterà come le liste siano trattate diversamente dai vettori (array) in Python, infatti, la variabile *Sport* sarà mostrata nella sua interezza, quindi comprese parentesi e virgolette.
Proseguiamo e vediamo come eseguire le classiche operazioni di base, cioè addizioni, sottrazioni, moltiplicazioni, divisioni ed elevamento a potenza:

```
   a = 5+3-2*6/2
print(a) # risultato 2.0
```

Osserviamo come far eseguire prima le addizioni e le sottrazioni:

```
   a = (5+3-2)*6/2
print(a) # risultato 18.0
```

Per elevare a potenza:

```
   print(6**3) # sei elevato alla terza: 216
```

Le operazioni di base si possono anche fare in questo modo:

```
   a = 5
b = a + 2
print(b) # risultato 7
```

Oppure in quest'altro:

```
    a = 5
a += 2
print(a) # risultato 7
```

Adesso vedremo che questi dati potremo anche convertirli, nel caso specifico osserveremo come e perché cambiare la tipologia di una variabile in Python.

Convertire i dati in Python

Quando inizieremo a scrivere dei listati più complessi, molto probabilmente, in alcuni casi, avremo bisogno che determinate variabili debbano essere di una tipologia specifica. Ad esempio sicuramente ci potrà capitare di dover convertire un numero in virgola mobile in intero, per esigenze di calcolo o anche solo per comodità. Questo processo per la conversione da un tipo di dati a un altro si chiama *typecasting* o conversione del tipo di dato. Python ci permette di farlo usando alcuni metodi piuttosto semplici.
Esistono due tipi di conversione: implicita ed esplicita. Della prima ne abbiamo già parlato nel paragrafo precedente, cioè quando Python deduce il tipo di variabile dall'input che le assegniamo per la prima volta, ad esempio una lista, un set, una stringa o un numero. Adesso esploreremo il typecasting esplicito, in cui un programmatore assegna non solo il valore a una variabile, ma converte anche il suo tipo di dati in un altro, vediamo la lista delle variabili e le possibili conversioni:

- **int()** con questa istruzione le variabili potranno essere cambiate in un numero intero:

```
    peso = "60"
nuova_var = int(peso)
print(nuova_var)
```

- **float()** varierà in un numero in virgola mobile:

```
    peso = 60
nuova_var = float(peso)
print(nuova_var)
```

- **str()** invece farà di un numero intero o in virgola mobile una stringa:

```
    peso = float(60.5)
nuova_var = str(peso)
print(nuova_var)
```

- **list()** con questa istruzione una stringa diventerà una lista:

```
    peso ="sessanta"
nuova_var = list(peso)
print(nuova_var)
```

- **set()** infine ecco come una stringa diventerà un set:

```
    peso ="sessanta"
nuova_var = set(peso)
print(nuova_var)
```

Operatori

Gli operatori sono quei simboli matematici che sono assolutamente imprescindibili nel mondo dell'informatica. Per questo motivo andremo a elencare i sei principali, ricordandoci bene che li useremo in egual misura in tutti i linguaggi di programmazione di questo libro, e in sostanza anche in tutti gli altri che esistono, a parte rarissime eccezioni come il linguaggio *Whitespace*.

Operatore	Utilizzo	Esempio
=	*Assegna un valore a una variabile*	a = 1
==	*Uguale a..*	if a == 5..
!=	*Diverso da..*	if a != 5..
< >	*Minore o maggiore di..*	if a < 5..
>= <=	*Maggiore o uguale, minore o uguale a..*	if a >= 5..

Proviamo sul campo ad utilizzare gli operatori appena visti con Python:

```
print('Inserisci la tua età:')
  Eta = input()
  Eta = int(Eta)
```

```
if Eta >= 18: print('Sei maggiorenne')
if Eta < 18: print('Sei minorenne')
```

Questo listato ci farà compagnia anche con gli altri linguaggi del libro, così potremo osservarne e comprenderne alcune differenze ma anche le molte similitudini.

Le variabili in C# e Java

Questo tipo di dato fondamentale nella programmazione ovviamente sussiste anche nel contesto di C# e di Java. A differenza di Python, questi linguaggi di programmazione richiedono la definizione manuale del tipo di dati di una variabile, e, come potremo vedere, per la maggior parte, le dichiarazioni di tipo variabile sono identiche per entrambi questi linguaggi. Per una panoramica dettagliata su alcuni dei principali tipi di variabili in Java e C# diamo uno sguardo all'elenco seguente:

- **bool** valore booleano, cioè vero o falso (*true* o *false*).
- **char** è il valore che riporta ad un singolo carattere alfanumerico.
- **double** numeri in virgola mobile, utilizzo 8 byte (64 bit).
- **float** numeri in virgola mobile, utilizzo 4 byte (32 bit).
- **int** identifica tutti i numeri interi.
- **String** / **string** essenzialmente un insieme di caratteri alfanumerici.

In conclusione, per quanto riguarda le **costanti** in Java ci serviremo del modificatore *final* utilizzando la sintassi:

```
final Qualcosa = 5
```

In C# invece avremo il modificatore *const*:

```
const int Qualcosa = 5
```

Provare il codice con Java

Abbiamo visto precedentemente con Python un listato in cui c'era la richiesta della nostra età. Il programma ha quindi mostrato un commento sullo schermo basato su questo input. Adesso ci renderemo conto che qui, con Java, le cose si fanno leggermente più complesse:

```
   import java.util.Scanner;
public class CalcoloEta {
  public static void main(String args[]) {
   Scanner richiesta_input = new Scanner(System.in);
    System.out.print("Inserisci la tua età: ");
    int Eta = richiesta_input.nextInt();

  if (Eta >= 18) System.out.println("Sei Maggiorenne");
  if (Eta < 18) System.out.println("Sei minorenne");
  }
}
```

Come abbiamo appena visto e anticipato Java richiede più impostazioni e, in alcuni casi come questo, più righe di codice rispetto a Python.
In questo listato abbiamo anche osservato che i programmi scritti in Java possono essere implementati utilizzando i cosiddetti *Java Packages*; i quali sono fondamentalmente contenitori di dati che aggiungeranno nuove funzionalità ai nostri progetti. Ad esempio, la prima riga del listato aggiunge funzionalità interattive a qualsiasi programma Java e dovrà essere presente quando sarà necessario l'input dell'utente. Nel nostro caso, del pacchetto chiamato *java.util* utilizzeremo la funzione *scanner*, che servirà per leggere l'input da tastiera. Ne vedremo altri man mano che procederemo nello studio.
Ora analizziamo i meccanismi concernenti l'input utente del listato precedente:

```
   Scanner richiesta_input = new Scanner(System.in);
```

Quello che succede qui è che creiamo un oggetto *Scanner* chiamato *richiesta_input.* Andando avanti, incontriamo le seguenti righe di codice:

```
   System.out.print("Inserisci la tua età: ");
int Eta = richiesta_input.nextInt();
```

Possiamo certamente osservare quanto sia diverso da Python la sintassi per scrivere su schermo, dopo sarà inizializzata una variabile *int* (cioè un numero intero) chiamata *Eta*, la quale è quindi inviata a una funzione chiamata *nextInt(),* che attende l'input dell'utente, aspettandosi un numero intero. La suddetta funzione fa parte del pacchetto (*packages*)

java.util.Scanner da noi importato nella prima riga del listato.
Oltre a conoscere i *packages* è molto importante ricordarsi che al nome della classe utilizzata inizialmente dovrà corrispondere una cartella dedicata con la stessa denominazione, quindi in questo caso la cartella sarà: *CalcoloEta*. Quest'operazione diventerà molto comune e sarà una delle prime cose che faremo quando testeremo il nostro listato Java.
Infine avremo forse notato alcune istruzioni in maiuscolo e l'ampio uso del punto e virgola (;), Java ne richiede uno dopo ogni istruzione. Inoltre, la sua sintassi pretende le parentesi attorno ai confini delle variabili e con la maggior parte delle funzioni. Tutte queste convenzioni si applicheranno anche in C#.

Provare il codice con C#

Adesso proveremo a vedere come sarà strutturato il listato sull'età in C#, probabilmente lo troveremo tutto sommato abbastanza simile a quello in Java, con alcune differenze chiave che esamineremo nel dettaglio nelle prossime righe, intanto vediamone la sintassi:

```
   using System
public class calcoloEta
{
  public static void Main()
  {
   Console.WriteLine("Inserisci la tua età: ");
    int Eta = Convert.ToInt16(Console.ReadLine());
     if (Eta >= 18) Console.WriteLine("Sei maggiorenne");
     if (Eta < 18) Console.WriteLine("Sei minorenne");
   }
 }
```

La prima riga di codice *using System* attiva uno specifico spazio dei nomi, più precisamente detto: *namespace*. Gli spazi dei nomi in C# sono *elementi contenitore* che ci aiuteranno a organizzare il nostro codice, ma soprattutto ci serviranno a risparmiare tempo. Senza lo spazio dei nomi con *System*, invece di *Console.WriteLine*, digiteremmo *System.Console.WriteLine* ogni volta che avremo bisogno di scrivere qualcosa su schermo. Si possono anche dichiarare i propri spazi dei nomi personalizzati, cosa che faremo più avanti nel libro. Per ora, è sufficiente

saperlo.
Rispetto a Java, C# usa un lessico diverso per molte delle sue funzioni. Per stampare il testo sullo schermo, abbiamo *Console.Writeline*. Per l'input dell'utente, abbiamo *Console.ReadLine* come mostrato dalla riga che abbiamo visto nel listato precedente:

```
int Eta = Convert.ToInt16(Console.ReadLine());
```

Quello che succede qui è che inizializziamo una variabile intera, *Eta*, e la passiamo a una funzione di conversione *Convert*. Gli diciamo di attendere l'input dell'utente e di aspettarsi un valore intero a 16 bit con segno. Questi sono numeri interi con un intervallo da -32.768 a 32.768. Questo sarà uno spazio sufficiente per ciò che il nostro utente andrà a inserire.
Interi a 16 bit senza segno portano valori compresi tra 0 e 65.536, il che significa che non si possono memorizzare valori negativi. Se abbiamo bisogno di memorizzare numeri davvero grandi, potremmo optare per l'utilizzo numeri interi a 32 bit, che vanno da -2.147.483.647 a 2.147.483.647, e senza segno da 0 a 4.294.967.295. Tuttavia in questo libro sarà sufficiente attenersi ai numeri più piccoli.

Le variabili in PHP

Rispetto agli altri il PHP è un linguaggio con *tipizzazione debole*, come Javascript. Questo vuol dire che le regole sulle dichiarazioni del tipo di variabili sono meno stringenti, a differenza di C# o Java dove bisogna sempre esplicitarne la tipologia, ad esempio: *int x = 1*. Vediamo la sintassi del PHP in questo caso:

```
<?php
$x = 1;
$Y = "Mario";
echo $Y;
```

Perciò, prima della nostra variabile avremo il simbolo *$*. Ricordiamoci che i nomi assegnati sono *case sensitive*, cioè sensibili alle lettere maiuscole o minuscole. Dobbiamo anche considerare che le variabili generalmente hanno uno *scope* (ambito) globale, mentre quelle all'interno delle funzioni hanno una validità locale, quindi relativa solo all'interno della funzione.
Le costanti invece verranno esplicitate tramite la funzione *define()* dove

all’interno avremo due argomenti: il nome e il valore della costante. Vediamo la sintassi: define("Qualcosa", 5)

Tipi di dati e typecasting in PHP

Anche in PHP la definizione dei dati avviene in maniera automatica, come in Python. Lo stesso vale per l'attribuzione forzata di un tipo di dato, cioè il *typecasting,* anche in PHP potremo farlo.
Guardiamo adesso le tipologie differenti di dati:

- **Binario** numero in base 2.
- **Booleano** un valore che può essere vero o falso (*true, false*).
- **Float** numero in virgola mobile.
- **Integer** numero intero privo di virgola, anche negativo.
- **Object** cardine dalla programmazione ad oggetti, dati con proprietà e metodi.
- **String** sequenze di caratteri.
- **Vettore (array)** insiemi di variabili associate a più valori.

Molto simili agli altri, con piccole differenze. Ora proveremo a fare il *typecasting* con PHP:

```
    <?php
$string = "Mario";
    $ogg = (object)$string;
echo $ogg->prova;
    ?>
```

In questo caso abbiamo trasformato una stringa (*$string*) in un oggetto (*$ogg*) e attraverso la freccia abbiamo visualizzato su schermo il risultato.

Provare il codice con PHP

Ed eccoci al test del codice dove proveremo una versione evoluta del nostro listato che controlla se siamo maggiorenni:

```
    <html>
    <body>
    <form>
    <h1 align="center">Verifica se sei maggiorenne</h1>
    <form method="post" action="">
  <p><input  type="text"  name="eta"  placeholder="Inserisci  la  tua
età: " required /></p>
  <p><input type="submit" name="contr_eta" value="controlla eta" />
</p>
    </form>

    <?php
    if(isset($_POST['contr_eta'])){
  $eta=$_POST['eta'];
  if($eta>18){
     echo '<p>Sei maggiorenne!</p>';
 }
 else{
    echo '<p>Sei minorenne</p>';
 }
}
?>

    </body>
</html>
```

Oltre alla lunghezza, la differenza con gli altri due listati simili che abbiamo visto è che questa volta abbiamo integrato il codice nel HTML. Ed è questo l'ambiente in cui spesso sguazzeremo quando lavoreremo con il PHP, e, molto spesso, è qui che questo linguaggio riesce a dare il meglio di sé.

Le iterazioni

Indubbiamente, i tipi di dati che si possono maneggiare nei linguaggi di

programmazione, come ad esempio le variabili che abbiamo studiato finora, sono fondamentali. Tuttavia dati a parte, abbiamo un sacco di strutture logiche a nostra disposizione per portare a termine tutti i compiti con il nostro listato. Queste strutture costituiscono ciò che è noto come controllo del flusso. Quando è digitato un elenco di programmi, dal computer è letto dall'alto verso il basso. Abbastanza spesso, l'elaborazione all'interno di questo programma deve essere ripetuta numerose volte. Ha quindi senso disporre delle capacità per il ciclo continuo e il flusso del programma condizionale.
I cicli di programmazione ci portano al concetto d'iterazione. L'iterazione è il processo di ripetizione di passaggi specifici per un numero di volte solitamente predeterminato al fine di ottenere un risultato utile. Una sequenza ripetuta di azioni nel contesto della programmazione è definita loop, ciclo o iterazione. Esistono molti modi per creare questi loop e dipende anche dalla lingua utilizzata. Vediamone un piccolo riepilogo:

Ciclo *While* in Java:

```
   int i=3
while (i>0) {
System.out.println("Tre ciao");
--i;
}
```

Il primo metodo d'iterazione è il ciclo *while* come dimostrato in Java. Quest'approccio esegue azioni finché non è soddisfatta la condizione nella funzione *while*. Nel nostro esempio, il ciclo è eseguito mentre la variabile *i* è maggiore di zero. A parte le differenze nella sintassi dei comandi quando si tratta di stampare il testo sullo schermo, il ciclo *while* è identico sia in Java che in C#.

Ciclo *While* in PHP:

```
   <?php
$x = 5;
while ($x > 0) {
   echo "Valore di \$x = $x\n";
```

```
    $x--;
}
```

Molto simile agli altri due, cambia solo leggermente la sintassi.

Ciclo *for* in C#:

```
    for(int i=0; i<3; i++)
{
  Console.WriteLine("Ciao!");
}
```

L'esempio in C# può sembrare leggermente più complesso. La struttura in questione, è un'antica tecnica composta di più elementi: La testa (cioè la parte che inizia con for) contiene istruzioni su quante volte il corpo (cioè la parte racchiusa tra parentesi graffe) deve essere eseguito. Nel nostro esempio, la parte iniziale recita come segue: definire una variabile numerica ausiliaria *i* e dargli il valore di zero (int i=0;), dopodiché esegui la parte del corpo del ciclo fintanto che la variabile *i* è minore di tre (i<3;), riprendi il programma. Infine aggiungi uno (1) al valore di i (i++).
Ancora una volta, a parte le differenze nella sintassi dei comandi, cioè *System.out.println* e *Console.WriteLine*, l'esempio è identico sia in Java sia in C#. Inoltre, in entrambi i linguaggi abbiamo anche il ciclo *do-while*, nel quale, rispetto a *while*, la condizione sarà verificata alla fine, mentre solo C# include l'istruzione *foreach*, utile per iterazioni sugli insiemi.

Ciclo *for* e *while* in Python:

```
    for i in range(10): print("Ciao numero",i)
```

Si noterà subito la mancanza del punto e virgola e delle parentesi graffe nel ciclo in Python. La riga contenente il comando *print* è infatti inclusa nel blocco di codice del loop usando solo l'indentazione. Inoltre, invece della struttura un po' più complicata di Java e C#, in Python possiamo semplicemente utilizzare una funzione elegante chiamata *range* per impostare il numero d'iterazioni.

```
    x=0
while x<5:
```

```
x=int(input('scrivi un numero maggiore di 5 '))
```

In quest'ultimo esempio Python si cimenta con *while*, finché l'utente non inserirà un numero maggiore o pari a cinque, il loop non terminerà.

Ciclo *for* in PHP:

```
   <?php
$colori = ['bianco', 'giallo', 'rosso'];
  for ($x = 0; $x < count($colori); $x++) {
    echo "$colori[$x]\n";
}
```

In questo esempio utilizziamo un conteggio, appena raggiunge il numero dei colori, il ciclo s'interrompe e mostra la stringa.
Anche in PHP, oltre ai cicli *for* e *while* abbiamo *do-while*, nel quale, rispetto a *while*, la condizione verrà verificata alla fine, e *foreach*, utile per iterazioni sugli insiemi.
Senza dubbio vi sono ancora un'infinita quantità di possibilità da esplorare con i loop. Sarà compito vostro studiarne tutti i segreti, noi, per problemi di spazio, abbiamo solo lanciato il sasso.

Quiz & esercizi

1) Creare un listato Python che riporti il valore 10, ma come stringa.
2) Convertire la stringa 10 di Python in un numero in virgola mobile.
3) Completa e correggi il seguente listato in Python:

```
print('Inserisci il valore del tuo portafoglio:')
  ???? = input()
  ???? = int()
       if Soldi >= 1000: print('Sei povero')
       if Soldi ....
```

4) Scrivere il listato completo appena visto negli altri tre linguaggi.
5) Scrivere quattro listati nei quattro linguaggi in esame utilizzando un loop che riporti come risultato: 1 2 3
6) Scrivere un listato che sfrutti i cicli *do-while* e *foreach* in PHP e C#
7) Correggere il seguente listato Python per farlo funzionare correttamente:

```
   x=0
while z>50:
z=int(input('scrivi un numero minore di 50 '))
```

8) Correggere il seguente listato Java per farlo funzionare correttamente:

```
   int x=5
while (x!=4) {
System.out.println("Cinque ciao");
--i;
}
```

9) Creare un listato in PHP per convertire un valore booleano in stringa.

10) Scrivere un listato in C# dove una costante e una variabile interagiscono.

Riassunto

La prima parte era dedicata alle principali differenze tra linguaggi di programmazione interpretati e compilati. Poi siamo passati alle variabili nel contesto della programmazione, come manipolarle e visualizzare il testo sullo schermo utilizzando Java, C#, Python e PHP.

Abbiamo visto i sei operatori universali di confronto delle variabili e quali sono i due concetti fondamentali nella programmazione orientata agli oggetti (OOP), le basi del controllo di flusso programmatico, e, alla fine le iterazioni (i loop o cicli).

 # 2 – La programmazione OOP

Classi e ereditarietà
Polimorfismo e sovraccarico
Le classi e gli oggetti

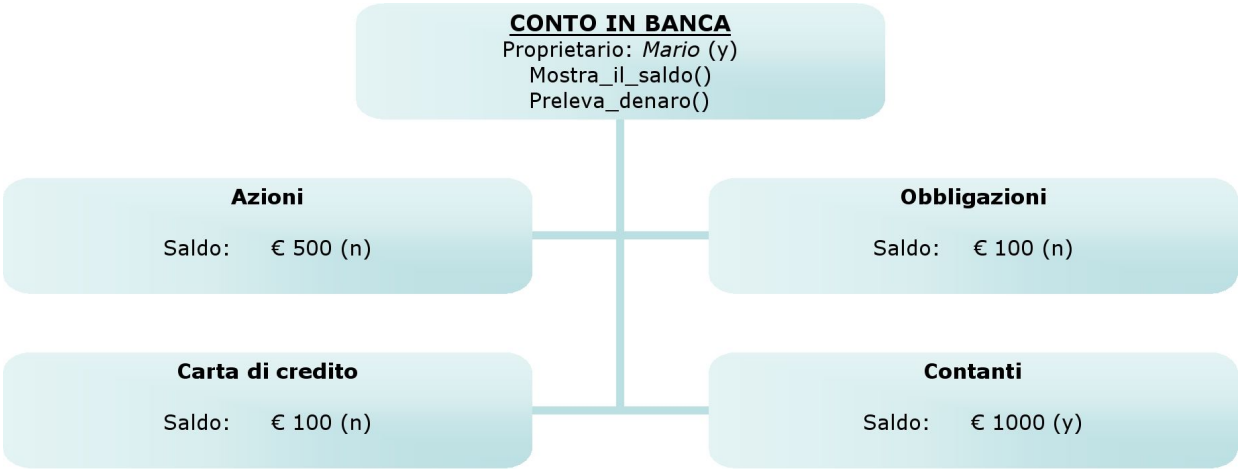
In questo capitolo tratteremo in dettaglio la cosiddetta *programmazione orientata agli oggetti*, che, come abbiamo già accennato si contrappone a quella procedurale. Questi due stili, o, più tecnicamente paradigmi, si differenziano dal metodo con cui si arriva alla risoluzione del medesimo problema. Senza addentrarci troppo in ragionamenti filosofici o campanilistici potremmo condensare le differenze nel fatto che la programmazione procedurale segue una logica più rigida con un listato che è un susseguirsi di procedure che comunicano tra di loro. La sequenza del codice ha una sua rilevanza. Mentre, nella programmazione OOP i dati del listato sono utilizzati come oggetti con determinate proprietà e metodi, per poi relazionarsi tra di loro. In questo caso la sequenza del codice è relativamente importante. Inoltre, esistono molte funzionalità aggiuntive rispetto alle loro controparti procedurali, ad esempio l'incapsulamento, una tecnica utilizzata per isolare parti del codice dall'una e dall'altra implementando identificatori di accesso. Esamineremo l'incapsulamento e altre specifiche relative agli oggetti in modo più dettagliato in seguito. Da segnalare infine che questi due stili possono coesistere, soprattutto nei linguaggi OOP, quando scriveremo un listato, avremo sicuramente porzioni del nostro codice prettamente procedurali.

Oltre a questa distinzione c'è da segnalare la sussistenza della *programmazione funzionale*, che può tranquillamente coesistere con quella OOP, ma è bene distinguere. Essenzialmente l'approccio è diverso, ci si comporta come se stessimo stilando una funzione matematica, per esempio la conversione tra il Dollaro e l'Euro o tra gradi Celsius e Fahrenheit.
Fatta questa precisazione, torniamo a noi, nel paragrafo continueremo esaminando molti dei concetti cruciali per il paradigma orientato agli

oggetti, usando principalmente il linguaggio Java per chiarezza, non abbandonando completamente C#, PHP e Python ovviamente.

Classi ed ereditarietà

In questo paragrafo parleremo di due concetti cardine della programmazione orientata agli oggetti (OOP) stiamo parlando delle classi e dell'ereditarietà degli oggetti. Per quanto riguarda le prime dovremo sapere che una classe essenzialmente è un "progetto" per creare e quindi poi interagire con degli *oggetti software*. Vediamo un esempio schematico che ci mostra quello di cui stiamo parlando:



In questo caso "*Conto in banca*" è una classe astratta. Fondamentalmente, queste sono classi che non possono essere utilizzate per creare oggetti. Qual è il loro scopo allora? In pratica, le classi astratte possono contenere lo stesso tipo d'informazioni delle classi normali e possono avere anche sottoclassi. Infatti, lo scopo di una classe astratta è fornire alcune informazioni condivise tra le sue sottoclassi, per poi ereditarle e creare oggetti. In molti casi, il loro utilizzo semplifica il processo di progettazione.

Continuiamo con l'esempio pratico della pagina precedente: le quattro caselle sotto la nostra classe astratta sono conosciute come sottoclassi. Evidenziano il concetto di ereditarietà, poiché *Azioni, Obbligazioni, Carta di credito* e *Contanti* riceveranno tutte le variabili e i metodi che la loro superclasse, *Conto in banca*, ha integrato.

In tutte le classi avremo delle informazioni (in questo caso variabili) che potranno essere pubbliche o private, cioè accessibili a tutti o meno, in questo caso le abbiamo indicate tra parentesi: (*n*) per le variabili private e (*y*) per quelle pubbliche. Infine, elenchiamo tutti i metodi trovati nelle nostre classi, come abbiamo fatto con *Mostra_il_saldo()* e *Preleva_denaro()*. Questa

rappresentazione grafica, che ritroveremo anche nel linguaggio UML, come già detto ci servirà anche a capire subito le variabili che saranno ereditate tra le varie classi, quindi, nel nostro programma *Conto in Banca*, le varie classi avranno come unico proprietario *Mario*, il quale potrà accedere ai metodi e a tutte le variabili.

Le Classi in PHP

Come in Java e C# l'elemento costitutivo di base di PHP è la classe. Questo perché è un linguaggio orientato agli oggetti, come i primi due. Insomma, le somiglianze sono tante, le scopriremo addentrandoci negli argomenti. Vediamo adesso la sintassi base di una classe in PHP:

```
   <?php
class Prova {
   public $variabile = "Mario";
   }
?>

   <?php
$oggetto = new Prova();
var_dump($oggetto);
?>
```

Come risultato avremo l'oggetto della classe *Prova*, che ha come attributo la stringa "*Mario*". Attraverso la funzione *var_dump()* saremo in grado di visualizzare informazioni riguardo alle variabili che ci interessano. Per avviare la macchina dovremo istanziare la classe creando un oggetto da essa, come abbiamo fatto nella riga: $oggetto = new Prova().
Per impostazione predefinita, le proprietà e i metodi della classe sono pubblici. Ciò significa che sono liberamente accessibili per essere letti, scritti o chiamati dall'interno e dall'esterno della classe. Chiaramente questa modalità non sarà quasi mai quella che utilizzeremo per questioni di sicurezza e per evitare di esporre gli algoritmi a chiunque. Perciò li imposteremo su privati. Per accedere ai dati, come per Java, avremo i metodi *Get* e *Set*:

```
   class Prova
{
```

```
 private $uno;
 private $due;
 public function prendiUno()
 {
       return $this->uno;
 }
 public function setUno($value)
 {
  $this->uno = $value;
 }
 public function prendiDue()
 {
  return $this->due;
 }
 public function setDue($value)
 {
  $this->due = $value;
 }
}
$prova = new Prova();
$prova->setUno(' valore 1 ');
$prova->setDue(' valore 2 ');
echo $prova->prendiUno();
echo $prova->prendiDue();
```

Questa modalità in cui i dati della classe sono protetti si chiama *incapsulamento*, perché la classe è letteralmente avvolta da una capsula per isolarla dall'esterno.
Nel listato abbiamo anche visto che attraverso *$this* potremo identificare e fare riferimento alle istanze all'interno della classe.

Polimorfismo, Get e Set in Java

Abbiamo parlato di ereditarietà di attributi e metodi da un'altra classe, bene, il polimorfismo si servirà di questi ultimi per eseguire altri compiti. Vediamo un esempio:

```
   class Auto{
 public void costoAuto() {
```

```
   System.out.println("Le auto hanno prezzi diversi");
 }
}

   class Ferrari extends Auto {
  public void costoAuto() {
    System.out.println("La Ferrari costa 150.000 €");
  }
}

   class Fiat extends Auto {
  public void costoAuto() {
    System.out.println("La Fiat costa 15.000 €");
  }
}

   class Esempio {
  public static void main(String[] args) {
   Auto lamiaAuto = new Auto();
   Auto fiat = new Fiat();
   Auto ferrari = new Ferrari();
   lamiaauto.costoAuto();
   fiat.costoAuto();
   ferrari.costoAuto();
 }
}
```

In pratica con la combinazione di ereditarietà (con *extends*) e polimorfismo saremo in grado di riutilizzare attributi e metodi di una classe esistente, come abbiamo appena visto nel listato con le classi *Auto, Fiat* e *Ferrari.*

Parlando ancora di ereditarietà, come nel paragrafo precedente relativo al PHP, i concetti di astrazione e incapsulamento nella programmazione a oggetti sono strettamente correlati. Il primo si riferisce alla tecnica per nascondere informazioni a prescindere che siano rilevanti o meno all'utente/codificatore di un programma. Quest'ultimo si occupa del funzionamento interno del software; nell'incapsulamento, sia i dati sia le funzioni per la loro manipolazione sono archiviati all'interno delle classi.

Ciascuna classe dispone quindi di controlli per i quali è possibile accedere ai dati da altre classi. Più avanti vedremo una dimostrazione sul campo di questi concetti, prima, però dovremo sapere che per utilizzarli ci serviremo delle parole chiave *Get* e *Set*.
In concomitanza con il comando *return*, sono utilizzati per il recupero e la definizione dei dati all'interno delle classi. Ricordiamoci che una funzione (ovvero un metodo) è un pezzo di codice che sarà eseguito solo al momento del bisogno.
Osserviamo la definizione di classe in Java con le funzioni Get e Set (nomi dei file *Prova.java* e *Principale.java*, quest'ultimo sarà il file di partenza):

```
 public class Principale {
public static void main(String[] args) {
Prova oggetto = new Prova();
oggetto.setName("Mario");
System.out.println(oggetto.getName());
}
 }

 public class Prova {
private String name;
public String getName() {
return name;
}
public void setName(String newName) {
this.name = newName;
}
 }
```

Iniziamo la definizione della nostra classe assegnandole un nome, che deve essere anche il nome del file. Nel nostro esempio, sarà *Principale*. La prima cosa che facciamo con esso è definire una variabile del tipo *String* (ovviamente per memorizzare i nomi di *Prova*). La parola chiave *private* prima della definizione della variabile è nota come modificatore di accesso.

È possibile accedere a metodi e variabili privati solo all'interno della classe in cui sono definiti (nel nostro caso solo da *Prova*).
Il metodo *getName* utilizza la parola chiave *return* per recuperare il nome di Prova. Il metodo è creato sia pubblico sia del valore *String*. Questo è

semplicemente perché dovrebbe restituire il valore di una variabile *String*.
Analizziamo quindi il metodo *setName* che abbiamo definito. La coppia di parole chiave *public void* specifica un metodo che non restituisce un valore. Va bene con le funzioni degli insiemi in generale poiché sono utilizzate per definire variabili e non per recuperare valori da esse. Infine la parola chiave *this* in Java si rivolge semplicemente all'oggetto contenente il metodo.
Le classi sono ottime per organizzare i dati, ma possono fare molto di più come vedremo. Useremo nuovamente *Prova* e ovviamente il nome del file che conterrà il codice sarà lo stesso della classe dichiarata: *Prova.java*. Ogni classe del programma finale sarà racchiusa in un file con estensione *.java*, e ognuno di esso dovrà contenere il metodo speciale *main*.
Come punto di partenza, aggiungiamo il codice per creare un vero oggetto in Java:

```
   public class Prova {
private String testo;
  public String getName() {
return testo;
  }
  public void setName(String newName) {
  this.testo = newName;
}
public static void main(String args[]) {
  Prova pr2 = new Prova();
  pr2.setName("Mario");
System.out.println("Ecco il nuovo direttore "+pr2.testo+" !");
System.out.println("Viva il direttore "+ pr2.getName());
  }
}
```

In questo listato aggiungiamo un metodo principale in modo da poter effettivamente sperimentare l'uso degli oggetti. Iniziamo creandone uno usando la classe *Prova* come costruttore, attraverso la sintassi Prova pr2 = new Prova(); Chiameremo il nostro oggetto "*pr2*". Poi invochiamo il metodo *setName* per dare un nome a *Prova*, accediamo al nostro oggetto per scrivere un nome in due modi diversi. Nel listato utilizziamo sia il cosiddetto operatore punto (cioè *pr2.testo*) sia il metodo *getName* per leggere i dati dal nostro oggetto. L'operatore punto è anche chiamato operatore membro.

Avendo a che fare con i listati Java capiremo subito che, come già detto, è attraverso la linea: *public static void main(String[] args)* che ogni programma comincia la sua elaborazione e le cose iniziano ad apparire sullo schermo per l'utente. La parte che definisce: (*String[] args)* fondamentalmente vuole dire che il programma accetta una singola stringa di testo come input da chiunque lo esegua, quindi, ad esempio "*buongiorno Mario*" invece di "*Mario*", non sarà mai accettato. Ovviamente lo stesso vale per gli argomenti (*args*), sempre in base alla presenza o meno delle parentesi quadre.

Classi e oggetti in C#

Per il linguaggio di casa Microsoft valgono gli stessi concetti di base sugli oggetti software e le classi, essenzialmente i primi sono semplicemente delle porzioni di dati rappresentati qualunque concetto, mentre le classi sono dei contenitori per oggetti o dati legati a dei metodi. Esplicitiamo tutto ciò con Il listato seguente, il quale ha esattamente le stesse funzionalità dell'ultimo appena visto. Ciò dimostrerà le molte somiglianze nella sintassi condivisa da Java e C#. Andiamo quindi a vedere una definizione di classe con un metodo principale in C# (nome del file: *Prova.cs*)

```
   using System;
public class Prova {
   private String testo;
   public String getName() {
     return testo;
  }
   public void setName(String altroTesto) {
     this.testo = altroTesto;
  }
public static void Main(string[] args) {
  Prova pr2 = new Prova();
 pr2.setName("Mario");
  Console.WriteLine("Ecco il nuovo direttore "+pr2.testo+" !");
  Console.WriteLine("Viva il direttore "+pr2.getName());
 }
}
```

I listati Java e C#, a parte piccole differenze di alcuni metodi che comunque svolgono lo stesso compito, sono praticamente identici. Questo sicuramente ci potrà aiutare nel nostro studio, sempre tenendo a mente le differenze.

Inizializzare le variabili

Nei linguaggi OOP, una volta che abbiamo evocato una classe, avremo bisogno di un mezzo per inizializzare le variabili dichiarate e che ci serviranno nel prosieguo del listato. Stiamo ovviamente parlando dei *costruttori*, che, anche se in modi leggermente differenti, saranno utili proprio a questo scopo. Per essere precisi, in informatica un costruttore indica dei metodi, i quali sono associati alle classi, proprio con l'obiettivo di inizializzare le variabili d'istanza. Per questo, alla creazione di un nuovo oggetto di una classe, in automatico sarà svolta la chiamata, poiché nella maggior parte dei casi non sarà possibile farlo in un secondo momento.
Esistono diversi tipi di costruttore:
- *Privato* non è visibile dalle altre classi, si utilizza per controllare e gestire la creazione degli oggetti.
- *Predefinito* non ottiene alcun parametro.
- *Implicito* è un costruttore predefinito che con un'unica istruzione crea e istanzia le variabili.
- *Copia* questi costruttori ottengono come parametro un'istanza dalla stessa classe, con le medesime proprietà.
- *Statico* inizializza i membri statici del tipo.

Il costruttore al lavoro

Come abbiamo accennato poche righe fa, un oggetto riceve i valori iniziali per tutte le sue variabili dalla definizione della classe. Tuttavia, ogni volta che è creato un oggetto, è possibile chiamare un costruttore per impostare tutti, o parti, di questi dati, cioè essenzialmente variabili.
Possiamo creare nuovi costruttori semplicemente definendo metodi che accettano attributi extra. Questi attributi sono quindi passati a ciascun oggetto, se necessario, per sostituire i valori originali definiti nella classe.
Un metodo costruttore deve avere un nome identico alla classe in cui è incluso. Vediamone la sintassi di base per ogni linguaggio, partiamo dal C#:

```
   public class Caffe
{
```

```
  private string uno;
  private string due;

  public Caffe(string deca, string normale)
  {
     uno = deca;
     due = normale;
  }
}
```

Vediamo adesso il medesimo listato con la sintassi di Java:

```
   class Caffe{
  private String deca;
  private String normale;

  public Caffe(String deca, String normale) {
   this.deca=deca;
   this.normale=normale;
  }
}
```

Osserviamo la sintassi di PHP:

```
   <?php
class Caffe {
  public $deca;
  public $normale;

  function __construct($deca, $normale) {
   $this->deca = $deca;
   $this->normale = $normale;
 }
}
?>
```

E, ultimo, ma non meno importante, Python:

```
   class Caffe:
def __init__(self, deca, normale):
self.deca = deca
self.normale = normale
```

Indubbiamente in quest'ultimo esempio abbiamo certamente potuto apprezzare la stringatezza della sitassi di Python rispetto agli altri linguaggi. A parte ciò nel listato abbiamo usato la parola *self*, la quale non fa parte della sintassi, anche se la troverete tantissime volte nelle applicazioni. Fa parte di una convenzione della programmazione, questo vuol dire che, volendo, potremo anche usare *Mario* al suo posto, ma è sempre meglio adeguarsi a questo tipo di standard, per facilitare la comprensione e l'usabilità delle nostre applicazioni.
Detto questo andiamo avanti e proviamo un listato più congruo. In questo programma di esempio, i due costruttori sono entrambi intitolati *Pugili* secondo la loro classe di origine, vediamoli all'opera con Java:

```
   public class Pugili {
  // Variabili di classe e i loro valori predefiniti
  private String titolo="Myke Tyson";
  private int campione = 1986;


  // Questo è un costruttore predefinito
   public Pugili() {
   }

  // metodo di costruzione per il nome e l'anno del titolo
public Pugili(String nome, int anno) {
  campione = anno;
  titolo = nome;
}

   public Pugili(String nome) {
campione = 1986;
titolo = nome;
}
public static void main(String[] args) {
```

```
  // Crea tre oggetti basati sulla classe "Pugili"
  Pugili primo = new Pugili("Myke Tyson");
  Pugili sec = new Pugili("Floyd Mayweather", 1998);
  Pugili terzo = new Pugili();

  // Visualizza i dati memorizzati in questi tre oggetti
 System.out.println(primo.titolo + " (" + primo.campione+")");
  System.out.println(sec.titolo + " (" + sec.campione+")");
  System.out.println(terzo.titolo + " (" + terzo.campione+")");
  }
}
```

In questo programma evochiamo tre oggetti tutti basati sulla classe *Pugili*. Tutti questi oggetti utilizzano metodi di costruzione diversi per dar loro vita.
Il primo oggetto, a che fare con il pugile Mike Tyson, viene creato utilizzando il metodo del costruttore che accetta una singola stringa. Il secondo oggetto usa il costruttore più versatile che accetta sia una stringa che un intero.
Il terzo e ultimo oggetto nel nostro esempio viene creato utilizzando il costruttore più semplice disponibile, che non accetta alcun attributo; assegnerà i valori predefiniti ("*Mike Tyson*" e *1986*) al nostro oggetto, come inserito nella definizione della classe.
Come esercizio create un listato, negli altri tre linguaggi, che riporti il medesimo output di quello appena visto, cioè:

Mike Tyson (1986)
Floyd Mayweather (1998)
Mike Tyson (1986)

Sovraccarico dei metodi

Nel linguaggio orientato agli oggetti possiamo avere diversi metodi con lo stesso nome senza problemi, purché la quantità e il tipo di dati dei loro parametri siano diversi. Questa modalità viene denominata “sovraccarico dei metodi”. Qualcuno si domanderà: ma non è la stessa cosa del polimorfismo? la risposta è si e no.. ricordiamo che il polimorfismo è la definizione di

un'interfaccia comune per tipi di dati diversi, ma anche l'uso di un simbolo per rappresentare più tipi diversi. Quindi esistono diverse tipologie di polimorfismo, il *polimorfismo ad hoc*, o quello *parametrico*. Nel primo avremo il sovraccarico delle funzioni, dove queste potranno essere applicate ad argomenti di tipo diverso e si comporteranno in base all'argomento. Essenzialmente parliamo di sottigliezze e di intrecci (dove il sovraccarico fa parte del polimorfismo) che coglieremo studiando le varie differenze anche tra gli altri linguaggi, ed è bene sapere che ci sono. Per approfondire il discorso esistono tante risorse online: https://en.wikipedia.org/wiki/Polymorphism_(computer_science) .

Adesso vediamo un esempio di sovraccarico in Java:

```
   public class Sovraccarico{
static int Nome(int intero) {
  return intero;
   }
   static String Nome(String a, String b, String c) {
  return a + b + c;
   }
   public static void main(String[] args) {


  int uno = Nome(33); // '!' nel sistema ASCII
  char due =(char)uno;
  String b = String.valueOf(due);
  String completo = Nome("Mario ", b, " Rossi ", b);
  System.out.println("Salutiamo il sig.: " + completo);
}
   }
```

Come abbiamo appena visto disponiamo di due metodi chiamati *Nome*. Il primo metodo accetta valori interi, mentre il suo omonimo poco più avanti accetta tre stringhe (a, b e c). Affinché il primo svolga appieno il suo lavoro, il valore intero viene prima convertito in una variabile a carattere singolo. Questo viene fatto utilizzando la funzionalità di *typecasting* di Java e fa sì che il carattere "!" venga memorizzato nella variabile *uno*. Questa variabile viene quindi convertita in una stringa utilizzando la funzione *valueOf* di

Java. Infine, combiniamo le tre stringhe, inclusa l'iniziale centrale, nel nome e cognome più comune in Italia.

Modificatori di accesso

Ora approfondiremo la scrittura di vari tipi di metodi. Nei linguaggi orientati agli oggetti ne esistono fondamentalmente due varietà, detti modificatori di accesso: statico e pubblico. Li abbiamo visti negli ultimi listati di esempio.

In alcuni casi sono anche ambivalenti, quindi un metodo può effettivamente utilizzare entrambi i modificatori, come, ad esempio, nel metodo principale di Java: public static void main().

Ora, la principale differenza tra le due varietà è che i metodi statici non hanno bisogno di essere evocati usando un oggetto, saremo in grado di chiamarli senza istanze specifiche di una classe. Tuttavia, i metodi statici non possono in nessun caso utilizzare variabili di classe come dimostreremo nel prossimo listato:

```
   public class Prova {

  private int x=3, y=8;
  static void statico() {
  System.out.println("Proviamo ad usare x o y.");
   // Come vedremo, questa riga in basso darà errore
  System.out.println(x + " + " + y + " = " + (x+y));
  }
  public void pubblico() {
  System.out.println(x + " + " + y + " = " + (x+y));
  }
public static void main(String[] args) {
statico(); // Chiama il metodo statico
   Prova oggetto = new Prova();
oggetto.pubblico(); // Chiama il metodo pubblico dell'oggetto
}
   }
```

Come abbiamo visto un metodo statico non può utilizzare le nostre variabili di classe, al contrario un metodo pubblico le potrà adoperare per alcune rudimentali operazioni aritmetiche. Visto che ne stiamo parlando, andiamo a vederli in maniera ancora più approfondita dato che sono molto importanti. Per prima cosa osserviamo che non sono solo privati o pubblici.

Noteremo che abbiamo accorpato i modificatori di C#, Java e PHP per via della loro similitudine:

Modificatatori	Accessibilità	Linguaggio
abstract	La *Classe* non può istanziare oggetti.	**Java**
default	Accessibile solo nel listato.	**Java**
final	la *Classe* non può essere ereditata, metodi e variabili immutabili.	**Java**
private	Accessibile solo dalla *Classe* dichiarata.	**Java, C#, PHP**
public	Accessibile.	**Java, C#, PHP**
Internal	Accesso limitato all'interno dell'as-sembly corrente.	**C#**
Protected	Accessibilie dalla *Classe* dichiarata e dai suoi derivati.	**C#, PHP**
Private protected	Accessibilie dalla *Classe* dichiarata e dai suoi derivati solo nello stesso assemby.	**C#**
Protected Internal	Accessibilie dalla *Classe* dichiarata e dai suoi derivati, solo nello stesso assemby.	**C#**

Ricordiamo che in Java, le classi possono avere accesso ai *pacchetti*. Con pacchetto ci si riferisce a un gruppo di classi correlate. Per ottenere ulteriori funzionalità useremo la parola chiave *import* nei nostri listati per portare classi specifiche (ad es. *import nome.pacchetto.classe*) o pacchetti completi (ad es. *import nome.pacchetto.**) nei nostri progetti.
Cosa possano offrire effettivamente i modificatori di accesso è una buona domanda da porsi, perciò proveremo a mostrare alcuni scenari in cui forniscono dei vantaggi concreti. Uno di questi scenari prevede il lavoro di squadra.
I dati incapsulati proteggono i progetti particolarmente grandi dagli errori umani. Con il codice incapsulato i programmatori non devono

necessariamente sapere come funziona un metodo; ciò che conta è l'output. Inoltre, l'uso corretto dei modificatori di accesso renderà i programmi generalmente più leggibili. L'aggiornamento e la manutenzione dei progetti software incapsulati è generalmente più semplice di quelli di tipo procedurale. Ricordiamoci che l'incapsulamento in OOP copre due significati.
Innanzitutto, è un termine utilizzato per descrivere l'approccio di associazione dei dati con i metodi che utilizzano le classi.
In secondo luogo, si riferisce alla limitazione dell'accesso diretto ai dati a livello di programmazione mediante modificatori di accesso. Nella tabella della pagina precedente, abbiamo visto anche i modificatori di accesso inerenti a C#. Possono sembrare abbastanza intercambiabili in questo momento, ma quando finiremo questo libro, sicuramente avremo acquisito familiarità con tutti loro rendendoci conto di quanto siano effettivamente necessari.

Polimorfismo, ereditarietà e modificatori di accesso in C#

Finora nel capitolo abbiamo trattato questi argomenti esclusivamente con Java. E' quindi ora che anche C# dimostri il suo valore, vediamo un listato che mostra l'ereditarietà e l'uso dei modificatori di accesso:

```
   using System;
class Chiusa {
  protected String testo;
  public Chiusa()
  {testo = "Maradona"; }
}

   // Crea una nuova classe derivata usando l'operatore :
class Figlia : Chiusa {
// Questo metodo di Figlia può accedere a Chiusa
public String getTesto()
{ return testo; }
}

   class Codice {
  // L'esecuzione principale inizia di seguito
  public static void Main(string[] args)
```

```
  {
    // Crea due oggetti, uno per ogni classe
    Chiusa Uno = new Chiusa();
    Figlia Due = new Figlia();
    // Visualizza la stringa con un metodo dalla classe derivata
        Console.WriteLine("Il  tuo  calciatore  preferito  è:  {0}",
Due.getTesto());
  }
}
```

Partiamo con la creazione di una classe che chiameremo *Chiusa* per motivi di chiarezza. Questa classe contiene un costruttore che imposta una variabile protetta, *testo*. Successivamente viene creata una seconda classe, *Figlia*, che eredita gli attributi della prima classe utilizzando l'operatore due punti (:). *Figlia* può ora accedere ai dati di *Chiusa*, quindi anche alla sua variabile protetta. Questo può essere fatto usando un metodo *get*, quindi ne creeremo uno solo per questa classe e lo chiameremo *getTesto()*.
Successivamente, passiamo al metodo principale dove creiamo due oggetti, uno per ogni classe che abbiamo definito in precedenza. Si noti come la sintassi per la creazione di oggetti in C# sia identica a quella utilizzata da Java.
L'ultima riga del nostro metodo principale mostra un messaggio con il contenuto della variabile stringa; questo è noto come una stringa formattata. In C#, le variabili vengono visualizzate nel testo usando parentesi graffe (le lingue derivate dal C le adorano in generale). L'elemento {0} si riferisce alla prima (e, in questo caso, l'unica) variabile che andremo a visualizzare accanto al messaggio. Se avessimo una seconda variabile da stampare nella nostra stringa, lo faremmo con {1} e così via.
Per quanto Java e C# siano vicini nella sintassi e nei loro approcci logici, ci sono alcune sottili differenze che all'inizio possono creare confusione. Ad esempio, il modificatore di accesso protetto viene trattato in modo diverso. In Java, *protected* equivale a *protected internal* di C# poiché è accessibile solo dalla classe dichiarante o derivata, o da una classe nello stesso pacchetto (in Java) o assembly (in C#).
Come abbiamo visto dalla tabella precedente un modificatore *Protected (*protetto) in C# è accessibile solo dall'interno della classe dichiarante e da qualsiasi derivato dalla classe originaria.
Rimanendo sul tema dell’ereditarietà, per quanto riguarda il polimorfismo e

il sovraccarico abbiamo tre differenti modalità:
1) possiamo avere una classe *Base* con il metodo *A* e le sue istanze indipendenti da *Base* con il metodo *A* ridefinito e sovrascritto.
2) possiamo avere una classe *Base* con il metodo *A* e le sue istanze con un metodo *A* sovrascritto, ma non completamente, il metodo originale *A* verrà chiamato se necessario. In entrambi questi casi (uno e due) dovremo utilizzare le parole chiave *virtual* per la classe base e *override* per le classi derivate, esempio: public virtual void Metodo()

```
    public class Base
{
   public virtual void Prova() { }
   public virtual int Esempio
   {
       get { return 0; }
   }
}
public class Derivata : Base
{
   public override void Prova() { }
   public override int Esempio
   {
       get { return 0; }
   }
}
```

3) possiamo avere una classe *Prova*, non istanziabile, come classe astratta con un suo metodo *A* astratto, vuoti, da riempire con codice delle classi e dei metodi derivati. In questo caso useremo la parola chiave *abstract* per le classi e i metodi derivati.

```
    public class Prova
{
    public virtual void Esempio(int i)
}
public abstract class P2 : Prova
{
    public abstract override void Esempio(int i);
```

```
}
public class P3 : P2
{
    public override void Esempio (int i)
}
```

Classi e oggetti in Python

Prendiamo spunto dall'introduzione delle classi in Python per riproporne il concetto. La classe potremo assimilarla ad un progetto, quindi ad esempio a quello di una casa, dove il rislutato del programma sarebbe la casa stessa. Quindi la classe è il progetto di un oggetto. Per costruire una casa servono mattoni e cemento, nel computer servirà la memoria. Anche in Python questo è uno dei principi cardine. Sebbene il linguaggio supporti tutte le principali tecniche OOP, la sintassi è abbastanza diversa da quella di Java e C#. Infatti, le parentesi graffe praticamente sono del tutto assenti. Inoltre, i costruttori sono definiti con la parola chiave *__init__()*, la quale ha due caratteri di sottolineatura su ciascun lato.
In Python, gli spazi bianchi diventano un fattore molto importante. Come accennato nel primo capitolo, l'*indentazione* è parte integrante della sintassi e viene utilizzata per denotare blocchi di codice all'interno di elenchi.
Ora, vediamo un semplice listato dove metteremo al lavoro una classe (*Prova*) per poter dimostare come funzionano gli oggetti in Python:

```
   class Prova:
     def __init__(self, b):
            self.b = b
oggetto = Prova("Mario")
print(oggetto.b, "ama le rosse e le ambrate, non le chiare")
```

Osservando il listato noteremo immediatamente le differenze rispetto a Java e C#. Innanzitutto, la classe *Prova* è composta da tre blocchi di codice separati, separati da tre diversi livelli di *indentazione*. Python lancerà effettivamente fuori un errore se questa logica di formattazione non verrà seguita. Ricordiamoci sempre di osservare questa regola, anche se praticamente tutti gli IDE correggeranno automaticamente l'eventuale errore. Oltre a questo In Python, le dichiarazioni di classe e la sintassi che prevede l'utilizzo del costruttore terminano con due punti (:).
L'espressione *self* è usata per indirizzare le variabili delle classi per ogni

oggetto che produce. Creare oggetti in Python è piuttosto semplice; diamo loro un nome e assegniamo una classe con un costruttore di nostra scelta. Nel nostro listato ci sarà solo un costruttore, e vedremo come utilizzarlo per creare una nuova classe, alla fine del programma:

```
   numero = 0
class Cioccolata:
def __init__(self, *args):
global numero
self.varieta = "Fondente"
self.purezza = args[0]
self.zona = args[1]
numero += 1


   if len(args) > 2 :
self.varieta = args[2]

   def mostraInf(self):
print("Il mio cioccolato viene dalla ", self.zona, "puro al ",
self.purezza, "%")
if self.varieta != "":
print("Varietà ", self.varieta)

   cioccoA = Cioccolata(70, "Colombia")
cioccoB = Cioccolata(90, "Venezuela", "Senza zucchero", "Quarto
argomento che non verrà considerato")

   cioccoA.mostraInf()
cioccoB.mostraInf()
print("Cioccolata mangiata:", numero)
```

Un listato leggermente più lungo e complesso di quelli incontrati finora che introduce diversi nuovi concetti. Uno di questi riguarda le variabili globali, difatti partiamo con il listato proprio creandone e inizializzandone una. Queste variabili possono essere utilizzate in qualsiasi momento all'interno di un listato, sia all'interno che all'esterno dei metodi di ogni classe.

Proseguendo abbiamo la riga *def __init__(self, *args):* che è l'unico costruttore per la nostra classe *Cioccolato*. Invece di accettare tipi di dati specifici, richiede un elenco di argomenti come indicato dall'espressione *args.
Python non supporta il sovraccarico dei metodi di per sé, come visto in Java e C#. Se dovessimo inserire un numero qualsiasi di metodi in una classe per scopi di sovraccarico, ciascuno di questi metodi sovrascriverebbe semplicemente il precedente.
Per assegnare un argomento a una variabile, utilizziamo *args[0]* per il primo argomento e *args[1]* per il secondo. Come si può notare, Python inizia a contare gli argomenti da zero.
Abbiamo anche una pratica funzione integrata per determinare la lunghezza degli elenchi e di altre strutture di dati, len. Questo viene utilizzato nella riga seguente nel nostro elenco *if len(args) > 2:* che significa semplicemente "se la lunghezza degli argomenti supera due". Fondamentalmente il nostro programma accetta fino a tre argomenti; il resto viene semplicemente scartato. Questo a sua volta è dimostrato con l'oggetto *cioccoB* quando gli diamo un totale di quattro argomenti; l'ultimo argomento non viene considerato da Python.
Per quanto riguarda *numero*, la nostra variabile globale, il suo valore viene aumentato di uno ogni volta che viene istanziata un nuovo oggetto dalla classe *Cioccolato*. Ciò riflette naturalmente il numero totale di oggetti *Cioccolato* in modo piuttosto accurato, difatti il risultato del listato sarà il seguente:

Il mio cioccolato viene dalla Colombia puro al 70 %
Varietà Fondente
Il mio cioccolato viene dalla Venezuela puro al 90 %
Varietà Senza zucchero
Cioccolata mangiata: 2

Ereditarietà e polimorfismo in Python

Come per molte altre caratteristiche, l'ereditarietà in Python è abbastanza semplice da implementare, questo perchè è proprio nella natura di questo linguaggio. Vediamo nel dettaglio come funziona questa caratteristica nel listato:

```
    class Prova:
        def __init__(self, *args):
            self.tipo = args[0]
            self.auto = args[1]
"Definiamo un metodo per visualizzare il tipo"
def esempio(self):
            print("Uso la", self.tipo, "come mia auto.")
    "Crea una classe figlio, Erede"
    class Erede(Prova):
        def __init__(self, *args):
            Prova.__init__(self, *args)
            self.colore = args[2]
    "Crea un oggetto usando la classe Erede"
    prova = Erede("Ferrari", "600", "rosso")
    prova.esempio()
    print("L'auto ha", prova.auto, "cavalli!")
    print("È un incredibile missile", prova.colore)
```

Possiamo subito osservare che la classe a metà del listato: *Erede(Prova):* denota l'inizio di una classe ereditata, *Erede*, la quale ottiene tutte le variabili della sua classe originaria, *Prova*.

Nel nostro esempio, questo significa che il tipo di stringhe e i *cavalli* ora diventano parte della classe *Erede*. Inoltre, dichiariamo una variabile aggiuntiva, *colore*, all'interno della nostra classe ereditata, assegnandole tre variabili da utilizzare in totale.

Naturalmente, l'ereditarietà in Python non si applica solo alle variabili; anche i metodi vengono trasmessi. Nel listato, *esempio* è un metodo originato dalla classe principale (Prova). Tuttavia, possiamo evocarlo utilizzando un oggetto istanziato dalla classe *Erede*.

Quindi, parlando di polimorfismo In Python, oltre all'ereditarietà, acclarato che il sovraccarico non è supportato, esiste un'altra modalità per

assegnare determinati attributi utilizzando la parola chiave *pass*. Proviamo un esempio che ci dimostra le differenze con l'ereditarietà:

```
    class Auto:
        pass
    "Creiamo due oggetti dalla classe Auto"
uno = Auto()
    due = Auto()
    "Aggiungi le variabili di costo e marca nella classe"
uno.costo = 60000
uno.marchio = "Tesla"
due.costo = 10
due.marchio = "Elettrica"
print("Acquisterei una", uno.marchio, due.marchio)
print("I costi saranno", uno.costo, " euro e", due.costo, "euro,
inanzitutto per comprarla e poi per fare 200km.")
```

Utilizziamo subito la parola chiave *pass* per creare una classe senza variabili o metodi. In Python, possiamo anche creare un'istanza di queste classi vuote, che è ciò che viene fatto dopo. E ora per la parte più interessante, possiamo salutare la modifica degli oggetti ad hoc. In Python, facendo riferimento a variabili inesistenti all'interno di istanze di classi vuote, creiamo nuove strutture di dati dette istanze. Questo attributo è conosciuto come *binding*.
Il *binding* degli attributi funziona anche per le classi, vediamo un listato che dimostra l'associazione degli attributi per le classi:

```
    class Auto:
pass

    "Creiamo un oggetto con le variabili costo e marchio incluse "
uno = Auto()
uno.costo = 60000
uno.marchio ="Tesla"
"Aggiungiamo costo e marchio alla classe"
Auto.costo = 10
Auto.marchio = "Elettrica"
"Crea un oggetto dalla classe modificata Auto"
```

```
due = Auto()
print("Acquisterei una", uno.marchio, "per", uno.costo, "euro")
print("L'alimentazione", due.marchio, "costa", due.costo, "euro ogni
200km")
```

Anche qui definiamo una classe vuota. Tuttavia, questa volta leghiamo gli attributi a questa classe e non solo ai suoi oggetti. Come è evidente dall'output dell'elenco, qualsiasi aggiunta di dati in una classe non sovrascrive i collegamenti precedenti dell'oggetto.

Gli oggetti in PHP

Abbiamo già parlato e visto la classe in PHP (e negli altri linguaggi), sappiamo che la sua definizione essenzialmente porta alla creazione di una struttura pronta ad eseguire alcune azioni, cioè i *metodi*, e ha anche degli attributi, cioè le *proprietà*, le quali ci serviranno anche per creare gli oggetti. Vediamo che anche in PHP non ci sono grandi differenze rispetto a ciò che abbiamo visto fin'ora:

```
   <?php
class Casa {
       public $calda;
       public $ariaCondizionataOn = false;
       public function accendiAriaCondizionata() {
       $this->ariaCondizionataOn = true;
    }
  }
class CasadiMario extends Casa
```

In questo esempio abbiamo visto come la classe Casa abbia le sue proprietà, cioè $calda e i suoi metodi, cioè accendiAriaCondizionata().
In quanto oggetto *Casa* potrà estendere metodi e proprietà alla *CasadiMario*, concetto che conosciamo già come ereditarietà.

Ereditarietà e polimorfismo in PHP

Il modello di ereditarietà in PHP si basa sull'ereditarietà singola. Ciò significa che una classe può avere un solo genitore. Le classi genitori possono avere un numero illimitato di classi figlie.

```
   class Prova
{
  protected int $foo = 1;
  private string $private = 'hidden';
}
class Provaprova extends Prova
{
  private int $bar = 2;
  public function getFoo(): int
{
  // accesso alle proprietà di provaprova
  return $this->foo;
}
public function getBar(): int
{
// accesso alle proprietà della classe
return $this->bar;
}
public function getPrivate(): string
{
// Impossibile dato che è privato
// return $this->private;
return 'Dati privati, impossibile accedere';
}
}
```

Anche con PHP avremo accesso al polimorfismo con il sovraccarico (*early binding*), che consiste nel definire più metodi con lo stesso nome assegnati ad oggetti differenti o con l'ereditarietà (*late binding*) dove, in fase di esecuzione, stabiliamo qual è il metodo appropriato da utilizzare in base alla classe di istanza dell'oggetto.

Il costruttore e il distruttore in PHP

Gli oggetti in PHP potranno essere costruiti ma anche distrutti attraverso i metodi *__construct()* e *__destruct()* i quali sono stati introdotti dalla versione 5 di PHP e si riconoscono dal fatto che hanno due righe di trattino basso davanti. Tecnicamente vengono definiti metodi magici, eccoli in azione:

```
   <?php
class Caffe {
 public $nome;
 function __construct($nome) {
   $this->name = $nome;
 }

  function __destruct() {
    echo "Il caffè è una miscela di {$this->name}.";
  }
}

   $arabica = new Caffe("Arabica");
?>
```

Come abbiamo avuto modo di vedere il distruttore viene chiamato quando l'oggetto viene distrutto o il listato termina. Quindi la funzione *__destruct()* è richiamata automaticamente alla fine del programma. Nello specifico una funzione *__construct()* viene richiamata automaticamente quando si crea un oggetto da una classe e poi il distruttore verrà evocato alla fine.

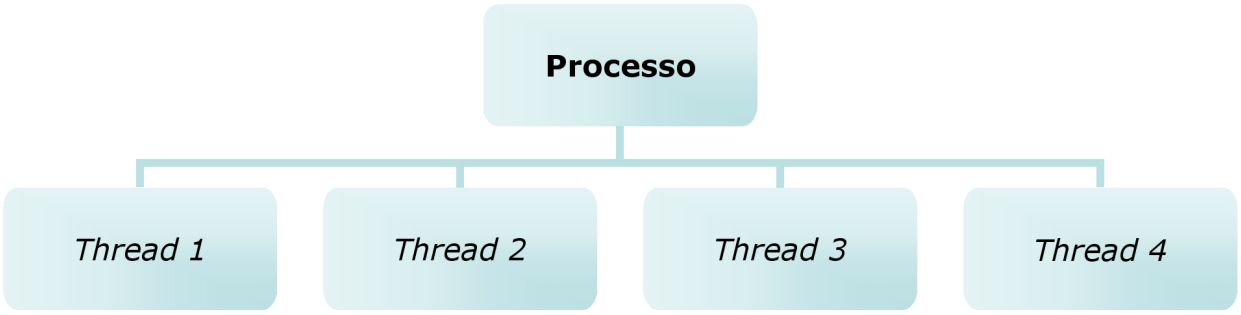
Il multithreading nella programmazione

Apriamo una parentesi su un argomento che tratteremo nel dettaglio nei prossimi paragrafi. Parliamo del multithreading che è la capacità di un programma o di un sistema operativo di abilitare più di un utente alla volta senza richiedere più copie del programma in esecuzione sul computer. Il multithreading può anche gestire più richieste dallo stesso utente.

Ogni richiesta dell'utente per un programma o un servizio di sistema viene tracciata come un thread con un'identità separata. Poiché i programmi funzionano per conto della richiesta di thread iniziale e vengono interrotti da altre richieste, viene tenuto traccia dello stato del lavoro della richiesta iniziale fino al suo completamento. In questo contesto, un utente può anche essere un altro programma.

Per il multithreading sono necessari un'elevata velocità dell'unità di elaborazione centrale (CPU) e grandi capacità di memoria. Il singolo processore esegue pezzi, o thread, di vari programmi così velocemente che sembra che il computer stia gestendo più richieste contemporaneamente.

Con il multithreading, mentre il processore del sistema informatico esegue un'istruzione alla volta, thread diversi da più programmi vengono eseguiti così velocemente da far sembrare che questi vengano eseguiti contemporaneamente.
(nella figura seguente vediamo lo schema di un processo multithreading)



Le velocità di elaborazione estremamente elevate dei microprocessori odierni rendono possibile il multithreading. Ogni ciclo della CPU esegue un singolo thread che si collega a tutti gli altri thread nel suo flusso. Questo processo di sincronizzazione si verifica così rapidamente che sembra che tutti i flussi siano in esecuzione contemporaneamente. Ciò può essere descritto come un programma multithread, poiché può eseguire molti thread durante l'elaborazione. Ogni thread contiene informazioni su come si relaziona al programma generale. Durante il flusso di elaborazione asincrono, alcuni thread vengono eseguiti mentre altri aspettano il proprio turno. Questo richiede ai programmatori di prestare molta attenzione per prevenire condizioni di sovrapposizione o di bug in genere.
Il multithreading viene utilizzato in molti contesti diversi. Un esempio si verifica quando i dati vengono inseriti in un programma grafico (come ad esempio il programma Photoshop) e utilizzati per un'applicazione in tempo reale. Quando si lavora su questo tipo di applicazione, un utente carica un'immagine e, tra le altre cose, potrebbe verificarsi quanto segue:

- L'immagine potrebbe essere ridimensionata.
- Potremmo applicargli un filtro.
- Salveremo più volte il progetto mentre ci lavoriamo.

Ogni attività si verifica perché più thread vengono generati ed elaborati per ciascuna attività senza rallentare il funzionamento generale

dell'applicazione, cioè del programma grafico in questo caso.
Il multithreading è diverso dal multitasking e dal multiprocessing. Tuttavia, il multitasking e il multiprocessing sono correlati al multithreading nei seguenti modi:

- Il multitasking è la capacità di un computer di eseguire due o più programmi simultanei. Il multithreading rende possibile il multitasking quando suddivide i programmi in thread eseguibili più piccoli. Ogni thread ha gli elementi di programmazione necessari per eseguire il programma principale e il computer esegue ogni thread uno alla volta.
- Il multiprocessing utilizza più di una CPU per accelerare l'elaborazione complessiva e supporta il multitasking.

L'elaborazione parallela è quando due o più CPU vengono utilizzate per gestire parti separate di un'attività. È possibile eseguire più attività contemporaneamente in un sistema di elaborazione parallelo. Ciò è diverso dall'utilizzo di un singolo processore in cui viene eseguito un solo thread alla volta e le attività che costituiscono un thread sono pianificate in sequenza.
I processori multicore su una scheda madre CPU hanno più di un'unità di elaborazione indipendente o core. Si differenziano dalle CPU single-core, che hanno una sola unità di elaborazione. Le CPU multicore offrono maggiore velocità e reattività rispetto ai processori single-core.
I processori multicore possono eseguire in parallelo tanti thread quanti sono i core della CPU. Ciò significa che parti di un'attività vengono completate più velocemente. Su un sistema single core, i thread di un'applicazione multithread non vengono eseguiti in parallelo. Al contrario, condividono un unico core del processore.

Quiz & esercizi

1) Definire il concetto di classe in un linguaggio OOP.
2) Cosa sono gli oggetti software e a cosa servono?
3) Scrivere un listato per tutti e 4 i linguaggi dove una classe figlia eredita i metodi della classe padre.
4) Qual è la differenza tra polimorfismo e sovraccarico?
5) Scrivere un listato dove due classi abbiano gli stessi metodi.
6) Scrivere un listato per dimostrare il sovraccarico in Python.
7) Scrivere un listato per ogni linguaggio, dove lavorerà un costruttore

predefinito.
8) Correggere il seguente listato in Python:

```
   class Prova:
def __init__(d, a):
a = b
c = Prova("Python")
print(b.a, "è un ottimo linguaggio")
```

9) Qual è la differenza tra multithreading e multitasking?
10) Correggere il seguente listato in PHP:

```
   <?php
class Mario {
 public $a;
 function __construct($a) {
   $this->Mario = $a;
   echo "Buongiorno da {$this->a}.";
 }
}
?>
```

Riassunto

Abbiamo studiato le principali differenze tra la programmazione procedurale e programmazione orientata agli oggetti (OOP) e a cosa si riferiscono l'astrazione, l'ereditarietà e l'incapsulamento nel contesto della programmazione a oggetti. Come definire classi e creare oggetti basati su di esse sia in Java, PHP e in C#. Le differenze tra i metodi pubblici e statici in OOP, nonché le basi dei modificatori di accesso.
Infine, cosa sono i costruttori, come usarli anche come metodi di sovraccarico, chiudendo con l'introduzione al multithreading.

 # 3 – Approfondiamo C#

Calendari e file
Multithreading
Programmazione asincrona

Iniziamo l'approfondimento, poiché a questo punto dovremmo aver acquisito sufficiente familiarità con la sintassi base e la configurazione di un ambiente di sviluppo integrato. Partiremo con uno dei linguaggi più versatili in circolazione: C#. Con i primi esempi avremo certamente notato che è un linguaggio di programmazione semplice, moderno, orientato agli oggetti e indipendente dai tipi, cioè non converte in maniera esplicita o implicita i valori da un tipo all'altro, per ragioni di sicurezza e stabilità. Tuttavia il typecasting sarà sempre possibile, anche se solo in alcuni casi limitati.
Com'è ovvio che sia, C# ha le sue radici nella famiglia di linguaggi C, e ciò consente agli sviluppatori di creare una vasta gamma di applicazioni per .NET Framework. È possibile usare C# per creare applicazioni client Windows, servizi Web, applicazioni client-server, di database più o meno complessi e molto altro ancora.
La sua sintassi è basata sulle parentesi graffe e risulterà immediatamente riconoscibile per chiunque abbia familiarità con i linguaggi C, C++ o Java. Gli sviluppatori che conoscono uno di questi linguaggi di solito sono in grado di iniziare a lavorare in modo produttivo in C# dopo un breve periodo di tempo. La sua sintassi semplifica molte delle complessità presenti in C++ e include potenti funzionalità.

Partiremo esplorando alcuni degli argomenti più avanzati relativi a questo robusto linguaggio, tra cui le basi della gestione del tempo, il multithreading, ma anche la possibilità di adattarsi alle diverse culture.
Prima di tutto però andremo a esplorare alcune delle ultimissime novità di C# 11, disponibili dalla fine di Settembre 2022:
- *Attributi generici* si potrà dichiarare una classe generica per rendere

disponibile una sintassi più pratica per l'uso di *System.Type*, vediamolo in un esempio pratico, prima:

```
    public class Esempio : Attributo{
   public Esempio(Type a) => TipoPar = a;
   public Type TipoPar { get; }
}

    [Esempio(typeof(string))]
public string Method() => default;
```

e dopo l'introduzione della classe generica:

```
    public class Esempio<A> : Attribute { }

    [Esempio<string>()]
public string Method() => default;
```

- *Valori stringa letterali non elaborati* permettono di contenere all'interno di ben tre virgolette caratteri speciali, anche virgolette e molto altro:

```
    string testoVario = """
   Testo complesso e
   particolare, anche nella
       formattazione.
               Tutto sarà rispettato.
   Inoltre potremo usufruire delle "virgolette".
   """;
```

Oramai bisogna abituarsi all'idea che tutti i linguaggi di programmazione subiscono periodici aggiornamenti, più o meno congrui e sicuramente vantaggiosi per gli sviluppatori, quindi non scordiamoci di controllare sempre sui siti ufficiali. Per il nostro C# avremo la sezione "Novità" sulla pagina principale: https://learn.microsoft.com/it-it/dotnet/csharp/ .
Parliamo adesso delle impostazioni locali riferite ai parametri inerenti una specifica nazione. Parliamo di globalizzazione e localizzazione. La prima si riferisce alla progettazione di un'applicazione per utenti di qualsiasi parte del

mondo. La seconda si riferisce al processo di adattamento di un'applicazione per conformarsi ai requisiti di una cultura specifica. In pratica, questi concetti riguardano i diversi calendari, valute, lingue e luoghi offerti dal nostro pianeta. In C#, abbiamo un gran numero di metodi e proprietà con cui presentare informazioni localizzate. Molti di loro hanno origine nello spazio dei nomi del modulo *System.Globalization,* come vedremo nel prossimo paragrafo.

Le impostazioni locali

Uno strumento molto utile riguarda la possibilità di poter usufruire dei vantaggi forniti dal sistema operativo della Microsoft inerenti le impostazioni locali di ogni nazione. Infatti, la classe *CultureInfo* ci fornirà i mezzi per localizzare i nostri programmi in C#. Ciò include il sistema di scrittura, la visualizzazione d'informazioni su ora, calendario e valuta personalizzate praticamente per tutti i paesi.

Per ottenere l'accesso alla classe *CultureInfo*, dovremo ancora una volta fare in modo che il nostro programma utilizzi lo spazio dei nomi chiamato *System.Globalization*.

In C# si usano i codici abbreviati delle nazioni che ci interessano. Nel prossimo listato, ne useremo quattro, cioè *en-US*, *it-IT*, *zh-CN* ed *es-ES*. Le prime due lettere in queste sigle rappresentano una lingua, mentre la seconda coppia di lettere maiuscole si riferisce a un paese. Ad esempio, la lingua Italiana, abbreviata in *it,* potrà essere rappresentata con altri codici cultura aggiuntivi, come *it-DE* (Germania) oppure *it-ES* (Spagna).

I codici appena mostrati si basano sull'elenco dei paesi in formato ISO 3166-1. A oggi sono supportati un totale di circa 250 codici lingua/nazione. Vediamo la sintassi di base della classe:

```
public class CultureInfo : ICloneable, IFormatProvider
```

IFormatProvider ci permetterà di usufruire di un sistema per ottenere un oggetto che controllerà la formattazione con il metodo *GetFormat*. Mentre *ICloneable*, come suggerisce il suo nome, attraverso il metodo *Clone* permette l'istanza di una classe clonata da una esistente, quindi con gli stessi valori.

Proseguiamo e osserviamo un listato che dimostra l'uso della localizzazione con la classe *CultureInfo:*

```
 using system;
 using System.Globalization;
 public class EsempioCodici
 {
public static void Main()
{
 // Creiamo un array di quattro oggetti con CultureInfo
CultureInfo[] prova = new CultureInfo[] {
new CultureInfo("en-US"), // USA
new CultureInfo("it-IT"), // Italia
new CultureInfo("zh-CN"), // Cina
new CultureInfo("es-ES")}; // Spagna

 DateTime data = new DateTime(1970, 12, 8, 12, 15, 00);

 // Visualizziamo la data non formattata
 Console.WriteLine("Il compleanno di Tony è " + data + " nella tua
lingua corrente.\n");

foreach(CultureInfo i in prova)
{
Console.WriteLine("Questa data in {0} è.. {1} ({2})",
i.EnglishName,data.ToString(i), i.Name);
}
}
 }
```

Abbiamo creato quattro istanze della classe *CultureInfo* per dimostrare le località di Stati Uniti, Italia, Cina e Spagna.
Abbiamo scelto la soluzione per utilizzare un elemento *foreach* durante la visualizzazione di oggetti in *CultureInfo*. Servirsi di *foreach* in C# è un'alternativa al ciclo *for* e tende a essere più adatto durante l'iterazione degli array (insiemi). Questo elemento funziona magnificamente con le istanze di *CultureInfo*.

Ora è il momento di fare un piccolo esempio di formattazione della valuta:

```
   using System.Globalization;
public class Valute
   {
  public static void Main()
  {

  int contanti = 500;
// Visualizziamo l'importo senza il formato valuta
Console.WriteLine("Senza formato: " + contanti.ToString());
// Impostiamo CurrentCulture su Italia
Thread.CurrentThread.CurrentCulture = new CultureInfo("it-IT");


   // Aggiungiamo il formato valuta usando "c"
Console.WriteLine("Italia: " + contanti.ToString("c"));

   Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
Console.WriteLine("America: " + contanti.ToString("c"));

   Thread.CurrentThread.CurrentCulture = new CultureInfo("zh-CN");
Console.WriteLine("Cina: " + contanti.ToString("c"));
}
   }
```

Abbiamo appena creato e visualizzato tre diversi formati di valutari rispettivamente per le lingue Italiana, Americana e Cinese. Il metodo *ToString("c")* ci è servito per assegnare una valuta.

Typecasting

Sappiamo già che, come detto all'inizio del capitolo, il typecasting in C# è possibile solamente in alcuni casi, vediamone alcuni in questa tabella esplicativa:

Da..	A..
booleano, char	**double** (le dimensioni maggiori o minori possibili di un numero)
int, long	**long** (numero intero molto grande, non

	rappresentabile da *int*)
float	**double**

La gestione del tempo

La gestione del tempo, del calendario e dell'orologio è abbastanza importante e la vedremo anche nei prossimi capitoli con gli altri linguaggi. Ugualmente C# offre tutto ciò di cui avremo bisogno per svolgere diversi compiti sull'argomento. Ecco un listato che dimostra l'uso dell'oggetto *DateTime*:

```
   using system;
public class Prova
{
  public static void Main()
  {

  DateTime data = new DateTime(1970, 2, 6, 12, 00, 0);
   Console.WriteLine("Mario è nato il {0}", data.ToString());


  // Recuperiamo e visualizziamo anno e giorno
   Console.WriteLine("Nel{0}era{1}", data.Year,data.DayOfWeek);

     // Creiamo un secondo oggetto per la prossima frase
      DateTime data2 = DateTime.Now;

    Console.WriteLine("Adesso invece siamo nel {1} di {0}",
data2.Year, data2.DayOfWeek);
  }
}
```

Iniziamo creando subito un oggetto che abbiamo scelto di chiamare *data*. Deve essere un'istanza della struttura *DateTime* e utilizzare un costruttore specifico per impostare l'anno, il mese e l'ora del giorno. Andando avanti visualizziamo questa data utilizzando un'istruzione *Console.Write* con un segno di formattazione {0} all'interno per includere, in questo caso, solo il

primo parametro. L'oggetto *data* viene anche convertito in testo utilizzando il metodo *ToString*.
Le strutture *DateTime* fanno parte dello spazio dei nomi System in C#, quindi non devono essere incluse nei nostri progetti in nessun altro modo.
Inoltre include una pletora di proprietà per l'accesso ai dati riguardanti il calendario dall'anno al millisecondo. Vediamo nel dettaglio alcune proprietà principali incluse:

Now (adesso), *Millisecond* (millisecondo), *Second* (secondi), *Minute* (minuti), *Hour* (ore), *Day* (giorno), *Month* (mese), *Year* (anno), *Date* (data), *TimeOfDay* (ora del giorno), *DayOf Week* (giorno della settimana) e *DayOfYear* (giorno dell'anno).

La gestione dei file

Quando dovremo avere a che fare con le tipiche operazioni solitamente impiegate con i file, come spostamento, copia, creazione, apertura, eliminazione e tutte le altre, disporremo di una classe chiamata semplicemente *File*. Potremo leggere e visualizzare file di testo, crearne di nuovi e recuperare numerosi tipi di attributi come la data di creazione e molto altro, come già detto.
Nel prossimo listato creeremo un nuovo file nominandolo *mario.txt*. Al suo interno scriveremo un piccolo messaggio ("Saluti da Mario!"). Infine procederemo a leggere e visualizzare il contenuto del file appena creato.
Per utilizzare la nostra classe, avremo bisogno di utilizzare *System.IO* per usufruire dello spazio dei nomi.
Vediamo sul campo un esempio che include *Writeline*, *StreamWriter, StreamReader* e *ReadLine,* ci servirà una cartella *prova* sul nostro disco rigido:

```
   using System;
using System.IO;

   class Prova
{
 public static void Main()
  {
     string path = @"c:\prova\prova.txt";
     if (!File.Exists(path))
```

```
      {
        // Creiamo un file
        using (StreamWriter x = File.CreateText(path))
        {
           x.WriteLine("Ciao");
          x.WriteLine("Da");
           x.WriteLine("Mario");
        }
   }

    // Apriamo il file
    using (StreamReader y = File.OpenText(path))
     {
        string a;
        while ((a = y.ReadLine()) != null)
        {
           Console.WriteLine(a);
         }
     }
 }
}
```

Ecco un listato che dimostra come servirsi delle classi *TextWriter* e *TextReader*:

```
   using system;
using System.IO;
class Prova
{
   static void Main(string[] args)
 {
  using (TextWriter scrivere = File.CreateText("c:\\mario.txt"))
  {
  scrivere.WriteLine("Saluti da Mario!");
  }
Console.WriteLine("Il file di testo creato in C:\n\nIt dice:");
using (TextReader leggi = File.OpenText("c:\\mario.txt"))
{
```

```
  Console.WriteLine(leggi.ReadToEnd());
  }
}
}
```

Oltre ai classici file di testo esiste un altro formato che adopereremo. Stiamo parlando dei file binari. Questo formato è molto versatile e può memorizzare testo, numeri e praticamente qualsiasi cosa potremmo mai usare. Adesso proveremo un listato dove con le classi *BinaryWriter* e *BinaryReader* creeremo un file binario e memorizzeremo un numero in virgola mobile, una stringa di testo e una variabile booleana. Vediamo l'esempio (dove utilizzeremo ancora la classe *System.IO* per lo spazio dei nomi):

```
   using system;
   using System.IO;
class Prova {
   static void Main(string[] args)
   {
  string nomefile = "c:\\prova.dat";
  using (BinaryWriter scrivi = new BinaryWriter(File.Open(file1,
FileMode.Create)))
  {
   scrivi.Write("Mario ama il mare");
  scrivi.Write(123456789);
   scrivi.Write(false);
  }
   Console.WriteLine("Dati scritti nel file binario " + file1 +
"!\n");
   using (BinaryReader leggi = new BinaryReader(File.Open(file1,
FileMode.Open)))
  {
   Console.WriteLine("Stringa: " + leggi.ReadString() );
   Console.WriteLine("Numero: " + leggi.ReadSingle() );
   Console.WriteLine("Booleano: " + leggi.ReadBoolean() );
 }
}
}
```

In questo listato abbiamo appena visto come lavorano *BinaryWriter* e *BinaryReader*, nelle prime righe abbiamo scritto tre tipi diversi di variabili (un numero in virgola mobile, un valore booleano e una stringa) in un file binario per poi andarli a leggerli e visualizzarli su schermo nella parte finale.
Piccola nota finale: quando si lavora con file binari non specifici, è in qualche modo consuetudine utilizzare l'estensione del file: *.dat*.

FileInfo

Questa classe, alternativa alla classe *File*, ci dà la possibilità di ottenere proprietà e metodi dell'istanza per tutte le operazioni sui file, oltre a facilitare la creazione di oggetti *FileStream*, offre anche un maggiore controllo ed è più utile in determinate condizioni.
Diamo un'occhiata a come accedere agli attributi dei file utilizzando *FileInfo* in questo listato che dimostra l'accesso agli attributi dei file:

```
   using system;
using System.IO;
class Prova
{
public static void Main()
{
  string nome = @"C:\mario.txt";

  // Qui impostiamo il nostro file di destinazione
  FileInfo esempio = new FileInfo(nome);
  string a = esempio.FullName;
Console.WriteLine("Dettagli del nostro file " + a + ":");
  string b = esempio.Extension;
Console.WriteLine("Estensione file: " + b);
  bool c = esempio.IsReadOnly;
Console.WriteLine("File di sola lettura: " + c);
  long d = esempio.Length;
Console.WriteLine("Dimensione: " + d + " byte");
  DateTime e = esempio.CreationTime;
Console.WriteLine("File creato: " + e);
  DateTime f = esempio.LastAccessTime;
Console.WriteLine("Ultimo accesso: " + f);
```

```
}
}
```

Si, è vero, all'inizio del nostro esempio abbiamo notato un simbolo che forse ci è sembrato avulso dal contesto: Il segno della chiocciola, ovvero @. Ne avremo bisogno per escludere la stringa che segue dall'interpretazione del compilatore. Quindi ciò che seguirà il simbolo verrà utilizzato così com'è, per evitare che tra i nomi delle directory possa celarsi una sintassi o qualche istruzione da eseguire. Quindi @ verrà di solito associato ad un percorso di un file o di una directory.
A parte questo dettaglio, nel listato è richiesto un file di testo, *mario.txt*, nella directory principale del nostro disco rigido (il listato ovviamente andrà adattato per chi usa MacOs o Linux). Ovviamente è possibile modificare il *filename1* in modo che punti a una posizione diversa e/o un file di testo diverso.
Abbiamo parlato degli attributi dei file e quindi è bene sbirciarne alcuni di quelli più comunemente usati e accessibili dalla classe *FileInfo*. Nell'elenco qui sotto possiamo osservarne le proprietà:

- Attributes riporta o cambia un attributo di un file.
- CreationTime riporta o cambia l'attributo della data di creazione del file.
- Directory riporta il percorso della directory.
- Exists viene utilizzato per determinare se esiste un file specifico.
- Fullname riporta il percorso completo del file e della directory (ad es. C:\Mario\testodiMario.txt).
- LastAccessTime riporta o modifica l'ora dell'ultimo accesso a un file o una directory.
- Length riporta la dimensione di un file in byte.
- Name riporta il nome di un file.

Diamo quindi un'occhiata a un esempio decisamente più completo relativo alle operazioni sui file. Infatti nel prossimo listato ci serviremo della classe *FileInfo* in modo più esteso. Ciò includerà: la copia di un file utilizzando il metodo *CopyTo* e la rimozione dei file utilizzando il metodo *Delete*:

```
   using system;
using System.IO;
```

```
   class Prova
{
public static void Main()
{
// Stabiliamo i percorsi della stringa originale e della copia
string nomefile1 = @"C:\prova.txt";
string file2 = @"c:\prova_copia.txt";
FileInfo esempio = new FileInfo(uno);

   // Creiamo un nuovo oggetto, anche per la copia
FileInfo esempio_copia = new FileInfo(due);

   // Usiamo Exists per sapere se il file è già stato creato
if(!esempio.Exists) {

   File.Create(uno).Dispose();
Console.WriteLine("File non trovato. Creazione " + uno);
} else Console.WriteLine("File trovato.");

   // Visualizziamo il nome del file
Console.WriteLine("Stiamo copiando " + esempio.FullName);

   // Se la copia esiste ok, altrimenti duplicheremo il file
if(!esempio_copia.Exists) { esempio.CopyTo(due);
Console.WriteLine("{0} copiato come {1}", esempio, due);
} else Console.WriteLine("Il file già esiste!");

   Console.WriteLine("Vuoi eliminare questi file? (s/n)");
char testo = (char)Console.Read();

   if(testo=='y') {
  Console.WriteLine("I file saranno immediatamente distrutti!");
  esempio.Delete();
  esempio_copia.Delete();
} else Console.WriteLine("File non cancellati!");
  }
}
```

Proseguiamo ed esaminiamo il programma nel dettaglio. Innanzitutto, definiamo due stringhe: *nomefile1* e *file2*. Queste stringhe contengono percorsi di file completi e puntano a due file nella directory principale di Windows C:. Potremmo anche semplicemente lasciare fuori i percorsi e restare solo con il nome del file (cioè, *prova.txt* e *prova_copia.txt*).
Successivamente, creiamo un oggetto usando la classe *FileInfo*, chiamandolo *esempio*. Un altro oggetto, *esempio_copia*, è quindi creato come duplicato del file principale. Era necessario creare un'istanza di *FileInfo* per utilizzare i metodi della classe in un secondo momento.
La riga *if(!esempio.Exists)* serve per esaminare la proprietà *Exists* piuttosto autoesplicativa dell'oggetto *esempio*. Con un punto esclamativo davanti, la clausola condizionale recita "*se il nostro file NON esiste*". Proseguendo, la riga: *File.Create(uno).Dispose();* contiene due istruzioni. Innanzitutto, il metodo *Create* serve per creare un file effettivo sul nostro disco rigido (o equivalente). Mentre il metodo *Dispose* sostanzialmente rilascia lo stato di accesso del file. Senza di esso, potremmo riscontrare, più avanti nel programma, problemi durante la manipolazione di questo file.
Ora, torniamo alla nostra clausola condizionale. Termina con la parola chiave *else*, che legge "*se il nostro file esiste*" e in questo scenario genera il messaggio "*File trovato.*". Successivamente, proviamo a copiare l'oggetto nel nostro file. Infine, nell'altra clausola condizionale: *if(!esempio_copia.Exists) { esempio.CopyTo(due);* esamineremo nuovamente l'esistenza di un file, questa volta concentrandosi su quell'altro che abbiamo definito in precedenza, *esempio_copia*. Se non lo troviamo, ad esempio perché è stato spostato, richiediamo il metodo *CopyTo* per copiare il nostro file (cioè, *prova.txt*) su *file2* (cioè, *prova_copia.txt*). Abbiamo di nuovo una parola chiave *else* alla fine della nostra clausola condizionale, ci servirà per visualizzare il messaggio "*Il file già esiste*!".
Passiamo ora ad alcune interazioni con l'utente. Gli sarà chiesto di digitare "y" (e premere Invio) se desidera eliminare sia il file originale sia il suo duplicato. Ciò si ottiene utilizzando il metodo *Delete*. Se l'utente inserisce qualsiasi altro carattere invece di "y", i file rimarranno intatti e verrà visualizzato il messaggio "*File non cancellati*!".

L'imbarazzo della scelta

Molti avranno notato la sovrapposizione tra le due classi che gestiscono i file. Tuttavia, ci sono ragioni per cui rimangono entità separate. La classe *FileInfo* è preferibile quando sono necessarie diverse operazioni per un file

dato, che ha un controllo più preciso sulla loro lettura e scrittura, mentre *File* funziona meglio per singole operazioni e può garantire prestazioni maggiormente rapide in alcuni scenari, ad esempio con il traffico di rete. *FileInfo* offre anche un minimo di controllo manuale attraverso le sue manipolazioni con le dimensioni dei byte.
I metodi all'interno di *File* sono statici, mentre quelli all'interno di *FileInfo* sono basati su istanze. I metodi statici accettano più argomenti, come percorsi di directory completi. A sua volta la classe *File*, in scenari specifici, come ad esempio, la codifica per ambienti basati sulla rete, potrebbe effettivamente rendere più efficaci i risultati. A questo punto proviamo un listato che dimostra il controllo manuale per la lettura dei dati utilizzando le classi *FileInfo* e *FileStream*:

```
   using system;
using System.IO;

   FileInfo oggetto = new FileInfo(@"c:\mario.txt");
FileStream leggi = oggetto.Open(FileMode.Open, FileAccess.Read);
byte[] insieme = new byte[leggi.Length];

   int grandezza = (int)insieme.Length;
int leggibyte = 0;

   while(grandezza > 0)
{
  int i = leggi.Read(insieme, leggibyte, grandezza);
  if (i == 0) break;
  grandezza-=i;
    leggibyte+=i;
}

   string testo = Encoding.UTF8.GetString(insieme);
Console.WriteLine(testo);
```

Per prima cosa creiamo un nuovo oggetto testo (chiamato appunto *oggetto*) che non potrà essere vuoto e lo apriremo per poterlo leggere.
Come avremo notato, la classe *FileStream* ci consente di leggere e scrivere file *byte per byte*. Questo è fatto usando il metodo *Open*. E' un metodo

versatile che richiede diversi attributi; per aprire il nostro file in modalità di sola lettura, inseriamo gli argomenti *FileMode.Open* e *FileAccess.Read* nel metodo *Open*. Successivamente nell'elenco, creiamo un array di byte, utilizzando la parola chiave *byte[].* Ricordiamoci che un array è una raccolta di valori, un insieme; potrebbe essere costituito da numeri, caratteri, stringhe o byte.
In C#, gli insiemi sono definiti usando parentesi quadre (come secondo promemoria, i byte sono costituiti da otto bit ciascuno e sono usati frequentemente per denotare i caratteri alfabetici, per uno).
Proseguiamo, successivamente nel listato, con la riga: int grandezza = (int)insieme.Length; creiamo un numero intero, *grandezza*, in cui applichiamo un metodo *Length* per scoprire la dimensione del nostro file in byte. Nel successivo ciclo *while*, i byte vengono letti utilizzando il consueto metodo di lettura.
Per essere più precisi, questo ciclo *while* deve essere eseguito mentre la variabile *grandezza* rimane maggiore di zero. In caso contrario il loop terminerà. Il flusso di file che abbiamo definito in precedenza, *leggi*, ha il metodo *Read* applicato con l'array di byte (insieme), la quantità di byte letti finora (*leggibyte*) e la quantità totale di byte da leggere (*grandezza*) come argomenti. Quest'uscita è immessa nella variabile *i*. Usciamo da questo ciclo quando si raggiunge lo zero usando la parola chiave *break*.
Gli ultimi passaggi del listato consistono nel convertire prima *insieme* in caratteri *Unicode* leggibili e per poi memorizzarlo in *testo,* dopodiché, la vedremo su schermo e questo porrà fine alla nostra applicazione.

Gestione della memoria RAM

La gestione della RAM in C# avviene automaticamente attraverso la cosiddetta *garbage collection* (GC, tradotto: raccolta di spazzatura).
Questo meccanismo serve per allocare e liberare la RAM in base alle esigenze di un programma; i linguaggi di programmazione con supporto per la GC liberano il programmatore da queste attività. La maggior parte dei linguaggi moderni, incluso C#, la supporta completamente. Questa funzionalità può essere aggiunta, anche in tutti quei linguaggi senza supporto nativo, sotto forma di librerie software aggiuntive.
Man mano che un progetto di programmazione cresce, di solito include sempre più variabili e altre strutture di dati, che consumano RAM se non adeguatamente curate. La gestione di queste risorse può diventare una gran seccatura per il programmatore. Inoltre, l'allocazione della memoria

compromessa potrebbe causare problemi di stabilità.
La *garbage collection* in C# è richiamata negli scenari seguenti:

- Il tuo computer sta esaurendo la memoria fisica. Più RAM ha il tuo sistema, meno frequentemente succederà.
- GC non riesce a recuperare gli oggetti con le giuste tempistiche, probabilmente i loro riferimenti sono errati.
- GC ha delle pause troppo lunghe, potrebbe capitare nelle operazioni di pulizia simultanee dovute a diversi thread da gestire, dove le pause tecniche diventano minime. In tal caso bisognerà ottimizzare gli eventi ETW (https://learn.microsoft.com/it-it/dotnet/standard/garbage-collection/performance).
- È superata una soglia specifica per la memoria utilizzata dagli oggetti allocati. Questa soglia è aggiornata in tempo reale dai componenti della *garbage collection*.
- Il metodo *GC.Collect* è richiamato dal programmatore. Non è utilizzato per progetti più piccoli.

Di casistiche ve ne sono altre, ma, prima di approfondire è importante capire il meccanismo di base. Infine, è bene sapere che ogni volta che un'applicazione è eseguita, il sistema operativo riserverà uno spazio nella RAM per la sua gestione. Questo spazio prende il nome di *heap nativo*.
La memoria *heap nativa* è allocata e deallocata al volo ogni volta che è eseguito un programma. L'area di memoria denominata *heap gestito* è un'entità diversa. Ogni processo separato ottiene il proprio *heap gestito*. Tutti i thread all'interno di un processo condividono anche questo spazio di memoria.

Multithreading

Ovviamente, lavorando con i sistemi della Microsoft, e dei PC oramai con minimo 2 core, il multithreading è supportato benissimo in C#, vediamo:

```
using System;
using System.Threading;

public class ProvaThread {
 public static void ThreadVai() {
    for (int i = 0; i < 10; i++) {
```

```
            Console.WriteLine("ThreadVai: {0}", i);
            Thread.Sleep(0);
        }
    }

    public static void Main() {
Console.WriteLine("Il Thread principale inizia un 2° thread.");
    Thread t = new Thread(new ThreadStart(ThreadVai));
    t.Start();
   for (int i = 0; i < 4; i++) {
Console.WriteLine("Il Thread principale esegue I suoi compiti");
    Thread.Sleep(0);
     }

    Console.WriteLine("Chiamato  Join(),  in  attesa  che  ThreadVai
finisca");
    t.Join();
Console.WriteLine("ThreadVai.Join  ha  finito,  premi  Enter  per
uscire");
Console.ReadLine();
     }
}
```

L’output che ne deriva:

```
    Il Thread principale inizia un 2° thread.");
Il Thread principale esegue I suoi compiti
ThreadVai: 0
Il Thread principale esegue I suoi compiti
ThreadVai: 1
Il Thread principale esegue I suoi compiti
ThreadVai: 2
Il Thread principale esegue I suoi compiti
ThreadVai: 3
Chiamato Join(), in attesa che ThreadVai finisca
ThreadVai: 4
ThreadVai: 5
ThreadVai: 6
```

```
ThreadVai: 7
ThreadVai: 8
ThreadVai: 9
ThreadVai.Join ha finito, premi Enter per uscire
```

Adesso invece vediamo qualcosa di leggermente più complesso:

```
   using System;
using System.Threading;

   public static void ProvaThread() {
Random casuale = new Random();
int delay = casuale.Next(0, 10000);
int  id  =  System.Threading.Thread.CurrentThread.ManagedThreadId;
Console.WriteLine("Il nostro thread " + id + "inizia.");
    Console.WriteLine("Il  thread  "  +  id  +  "è  sospeso  per  {0}
secondi..", Math.Round(delay * 0.001, 2) );
  Thread.Sleep(delay);
  Console.WriteLine("Il thread " + id + " ora riprenderà.");
}

   static void Main(string[] args) {
 ThreadStart figlio = new ThreadStart(ProvaThread);
 for(int i=0; i<4; ++i) {
   Thread ProvaThread = new Thread(figlio);
   ProvaThread.Start();
  }
}
```

Innanzitutto, definiamo un metodo personalizzato, *ProvaThread()*, per sviluppare nuovi thread. In seguito creeremo quattro thread che utilizzeranno simultaneamente con il metodo *Sleep*, che interromperà semplicemente l'elaborazione di un thread.
All'interno dei nostri thread, a questi metodi è assegnato un valore casuale compreso tra 0 e 10000 che corrisponde a un massimo di dieci secondi di ritardo, poiché quel valore sarà calcolato in millisecondi, tutti da conteggiare contemporaneamente secondo il paradigma basato sulla funzionalità del thread. Difatti, come sappiamo già, i programmi sono costituiti da

processi. I thread sono le entità all'interno dei processi che possono essere pianificati in base alle loro necessità. Anche in C# possiamo esercitare il controllo sulla creazione e su altri eventi del ciclo di vita dei thread.
Continuiamo, successivamente, nel nostro metodo personalizzato, ci serviremo di una proprietà piuttosto prolissa chiamata:

```
System.Threading.Thread.CurrentThread.ManagedThreadId
```

per identificare in maniera univoca ogni thread in esecuzione, memorizzandolo poi in *id*.
Quando s'impostano le informazioni sul thread, il ritardo variabile è moltiplicato per 0,001 per visualizzarlo in secondi. Utilizziamo anche il metodo *Round* della classe *Math* per arrotondare il nostro numero intero di ritardo a due decimali.
Inteso ciò, continuiamo con la creazione di un oggetto (chiamato *casuale*) e poi l'impostazione di un ritardo tra zero e dieci secondi utilizzando il nostro oggetto, utile al funzionamento del programma, passa in secondo piano.
Infine ci renderemo anche conto che il metodo principale che abbiamo appena osservato è piuttosto semplice. In pratica usando un ciclo *for*, creiamo esattamente quattro istanze di *ProvaThread*, iniziando poi la loro esecuzione ovviamente servendoci di *Start()*.

Sincronizzare i thread

Questa è un'operazione essenziale per il buon funzionamento dei nostri programmi e, infatti, come per gli altri linguaggi, anche con C# avremo accesso a un meccanismo di blocco per le applicazioni con thread.
Parliamo dei cosiddetti *Lock,* o meglio, appunto: blocchi. Con essi potremo sincronizzare i nostri thread per ottenere dei risultati più logici o precisi. Procediamo osservandolo meglio in questo programma e nel suo output:

```
   using System;
using System.Threading;
 public class Sincronizzare
 {
   public void Prova()
   {

  Thread info = Thread.CurrentThread;
```

```
  // Applichiamo un blocco per sincronizzare il thread
    lock (this){
    Console.WriteLine("(Questo thread è " + info.ThreadState+" con "
+ info.Priority + " priority)");

   for(int i=0; i<4; ++i) {
   Console.WriteLine(info.Name + " ha dormito per " +i+ " ore");
Thread.Sleep(650);
   }
 }
}
}
  public static void Main()
  {
  LockingThreads prova = new LockingThreads();
  Thread uno = new Thread(new ThreadStart(prova.OurThread));
  Thread due = new Thread(new ThreadStart(prova.OurThread));
  uno.Name="Mario";
  due.Name="Paola";

   uno.Start();
due.Start();
}
```

L'output che ne deriva sarà:

```
   Mario ha dormito per 0 ore
Mario ha dormito per 1 ora
Mario ha dormito per 2 ore
Mario ha dormito per 3 ore
Paola ha dormito per 0 ore
Paola ha dormito per 1 ora
Paola ha dormito per 2 ore
Paola ha dormito per 3 ore
```

In pratica, con il meccanismo di blocco in atto, il thread *uno* sarà elaborato fino al completamento, a quel punto partirà il thread *due*.
Inoltre, senza il meccanismo di blocco, osserveremo righe su *Mario* e *Paola*

alternarsi casualmente sullo schermo, senza una sequenza precisa. Come esercizio riscrivete il listato senza *Lock*, per testare la differenza.

I metodi Yield e Join

Oltre ai blocchi, avremo bisogno di qualcosa che ci permetta di gestire i thread in maniera più sostanziale. Un'istruzione l'abbiamo già utilizzata, ci ha fatto compagnia in diversi listati visti finora, stiamo parlando del metodo *Sleep*. Sappiamo che essenzialmente dice a un thread di andare a prendersi un caffè, e quindi di aspettare un tempo che andremo a specificare nel listato. E' ragionevolmente un'ottima possibilità, ma con buone probabilità possiamo avere di meglio. È ora quindi di parlare dei metodi *Yield* e *Join*. La prima parola chiave ha più significati secondo il contesto, ad esempio in un loop con *yield return* sarà restituito un elemento alla volta, mentre con *yield break* usciremo dall'iterazione. Vediamo:

```
    public class Raddoppio
{
   static void Main()
   {
       foreach (int i in Doppio(2, 8))
       {
           Console.Write("{0} ", i);
       }
   }
        public   static   System.Collections.Generic.IEnumerable<int>
Doppio(int numero, int esponente)
   {
       int ris = 1;
       for (int i = 0; i < esponente; i++)
       {
           ris = ris * numero;
           yield return ris;
       }
   }
}

    Risultato: 2 4 8 16 32 64 128 256
```

Piccola parentesi su *System.Collections.Generic* il quale serve per racchiudere interfacce e classi che circoscrivono raccolte generiche, le quali consentono quindi agli utenti di crearne di fortemente tipizzate per fornire indipendenza dai tipi e prestazioni migliori rispetto alle raccolte fortemente tipizzate non generiche. In questo caso abbiamo usufruito di IEnumerable<int> per evidenziare l'enumeratore, il quale ha supportato un'iterazione semplice su una raccolta di un tipo specificato. Parliamo di una funzionalità di .NET 6 che, per ragioni di spazio non possiamo includere nel paragrafo, ma che è disponibile qui: https://learn.microsoft.com/it-it/dotnet/api/system.collections.generic?source=recommendations&view=net-6.0
Torniamo a noi, e all'oggetto del paragrafo: il multithread, in quest'ambito *Yield* dice a un thread di entrare in uno stato di attesa indeterminata. Un thread sospeso sarà riattivato ogni volta che sarà necessario. Ciò può verificarsi in pochi millisecondi o richiedere molto più tempo. Sostanzialmente *Yield* libera la CPU dall'esecuzione di un thread specifico per elaborarne altri più urgenti. Il livello di questa urgenza è in definitiva decisa da ciò che governa il nostro dispositivo, stiamo parlando ovviamente del suo software di gestione, cioè il sistema operativo (Windows, MacOS, Android, Linux, ecc), vediamolo subito:

```
   using System;
using System.Threading;
public class Prova
{
 private int conteggio;
   public void ConYield()
   {
   Console.WriteLine("1° thread con un loop infinito");
   while(true)
     {
     Thread.Yield();
     ++conteggio;
   }
}
public void Secondo()
{
Console.WriteLine("1° thread raggiunto dal 2°" + conteggio);
```

```
}
}
public class Esempio
{
 public static void Main()
 {
   Prova oggetto = new Prova();
   Thread uno= new Thread(new ThreadStart(oggetto.ConYield));
   Thread due = new Thread(new ThreadStart(oggetto.Secondo));
   uno.Start();
   due.Start();
  }
}
```

In questo caso, dopo aver dichiarato la classe *Prova,* ne definiamo una privata chiamata *conteggio*, che riporti un numero intero. Ci servirà per documentare la durata (all'incirca) d'esecuzione del primo thread prima di far partire *Yield*. Avvieremo quindi due thread, il primo entrerà in un ciclo infinito, mentre il secondo servirà per visualizzare l'importo memorizzato nella controvariabile (*ConYield*). Senza l'ausilio di *Yield*, il contatore visualizzerà una lettura di diversi ordini di grandezza superiore, causando alla fine la mancata risposta del programma, mentre usando *Yield*, possiamo aspettarci che la maggior parte delle volte sia ben di sotto i due o tre secondi. Anche in questo caso, il sistema operativo è l'entità finale che pianifica il thread che effettua una richiesta *Yield*, in base allo stato e alle priorità degli altri thread.
Da notare che avremo anche la possibilità di emulare *Yield*, almeno parzialmente, sempre tramite il metodo *Sleep*. I vecchi framework .NET (precedenti alla versione 4.0) non erano ancora compatibili con *Yield*. In questi casi, digitando *Sleep(0)* otteniamo qualcosa di abbastanza simile.
Dulcis in fundo parliamo di uno dei metodi più importanti per il lavoro con thread in C#, che è chiamato *Join*. Ci servirà per far attendere i thread mentre gli altri terminano l'elaborazione. Naturalmente *Join* può essere chiamato solo da un thread che ha già iniziato la sua esecuzione, come illustrato nel prossimo listato:

```
   using System;
using System.Threading;
```

```
   static void Prova() {
  for (int i = 0; i < 5; ++i)
  Console.Write(i + " ");
}
static void Main(string[] args) {
  Thread uno = new thread(Prova);
  uno.Start();
  uno.Join();

  Console.Write("5"); // Termina la lista con il numero 5
}
```

Essenzialmente questo è un esempio basico del metodo *Join*. L'output che otterremo sarà, se tutto è corretto: 0 1 2 3 4 5.
Senza *Join*, potremmo ottenere risultati imprevisti, come 100 1 2 3 4 5. Questo perché all'ultimo *Console.Write* non è detto in modo specifico di attendere finché *uno* (il thread) non completa il suo lavoro. Sarà, infatti, *Join* che assicurerà che un thread completi la sua elaborazione prima che l'elenco continui. Vediamolo meglio in un altro listato:

```
   using System;
using System.Threading;

   public class Prova
{
  static Thread uno, due;
  public static void Main()
  {
     uno = new Thread(Lavoro);
     uno.Name = "Thread1";
     uno.Start();
     due = new Thread(Lavoro);
     due.Name = "Thread2";
     due.Start();
  }

  private static void Lavoro()
  {
```

```
     Console.WriteLine("\nThread all'opera in questo momento: {0}",
Thread.CurrentThread.Name);
      if (Thread.CurrentThread.Name == "Thread1" && due.ThreadState
!= ThreadState.Unstarted)
        due.Join();
     Thread.Sleep(3000);
      Console.WriteLine("\nThread all'opera in questo momento: {0}",
Thread.CurrentThread.Name);
     Console.WriteLine("Thread1: {0}", uno.ThreadState);
     Console.WriteLine("Thread2: {0}\n", due.ThreadState);
  }
}
```

Programmazione asincrona

Dopo aver compreso abbastanza bene il multithreading, sicuramente avremo intuito che si trattava di aver scalfito solamente la superficie di un qualcosa di tremendamente intricato e complesso. Non facciamola troppo drammatica, in sostanza si tratta di mantenere l'ordine. Infatti, in quest'ottica la programmazione asincrona, fondamentalmente, è l'arte di avere un programma composto di numerose attività che non entreranno in conflitto tra loro (o altri programmi). Quest'approccio è spesso anche più facile per gli occhi del programmatore poiché solitamente si associa a una disposizione del codice maggiormente chiara.

L'attuale implementazione dell'elaborazione asincrona in C# sfrutta una classe denominata *ThreadPool*. Questa classe adotta un approccio diverso rispetto alla classe *Thread* (discussa in precedenza in questo capitolo). Essenzialmente, la prima classe menzionata crea solo thread in background con priorità basse e non avrà nulla a che fare con l'assegnazione della priorità ai thread, ambito della seconda classe.

Prima di procedere oltre è bene sapere che ci sono tre principali paradigmi di programmazione asincrona in C#. Solo uno di questi, chiamato *TAP*, è consigliato da Microsoft a partire dal 2021. Vediamoli nel dettaglio:

- *Modello di programmazione asincrono* (APM): quest'approccio usa l'interfaccia *IAsyncResult* e i relativi metodi *BeginOperationName* ed *EndOperationName*. Microsoft non consiglia più di utilizzare questo modello di progettazione alquanto complesso.
- *Pattern asincrono basato su eventi* (EAP): è stato ideato per offrire i vantaggi del paradigma del software asincrono, mantenendo le cose

relativamente semplici dal punto di vista del programmatore. Anche questo modello di progettazione è considerato un approccio obsoleto.
- *Pattern asincrono basato su attività* (TAP): è il modello di progettazione più aggiornato ed elegante per la scrittura di software asincrono. A differenza di APM ed EAP, TAP usa un unico metodo per rappresentare l'inizio e il completamento di un'operazione asincrona ed è incentrato sulle classi *Task* e *Task<TResult>*.
Per un esempio su di un pattern asincrono che utilizza TAP, andiamo subito a vedere il prossimo listato:

```
    class Prova
{
   public static async Task<int> Main(string[] args)
   {
       Console.Title = "async Task<int> Main";
       int uno = 10, due = 12;
          Console.WriteLine($"La somma di {uno} e {due} è: {await
AdditionAsync(uno, due)}");
       Console.WriteLine("Premi un tasto per uscire.");
       Console.ReadKey();
       return 0;
   }
   private static Task<int> AdditionAsync(int a, int b)
   {
       return Task.Run(() => SUM(a, b));


       //Il codice funzione per la somma
       int SUM(int x, int y)
       {
           return x + y;
       }
   }
}
```

L’output sarà:

```
La somma di 10 e 12 è: 22
```

Ora, in questo semplice esempio, dove da un sito Web viene scaricata un'immagine, ci serviremo di *async* e *await* per gestire la situazione:

```
    static Task<byte[]> ScaricaImmagine(string url)
{
  var client = new HttpClient();
  return client.GetByteArrayAsync(url);
}

    static async Task SalvaImm(byte[] bytes, string imagePath)
{
       using    (var    fileStream    =    new    FileStream(imagePath,
FileMode.Create))
  {
    await fileStream.WriteAsync(bytes, 0, bytes.Length);
  }
}
```

Ovviamente non abbiamo utilizzato *Task.Run* essendo il download e il salvataggio operazioni di I/O (input/output), per cui ci siamo serviti di *FileStream* e *HttpClient* di .NET e ovviamente di *async*. Usando la definizione *async* accanto a un metodo, lo designiamo come asincrono. Questo ci consentirà quindi di utilizzare la parola chiave *await*, che fornisce il controllo al suo metodo di chiamata.
Il meccanismo *async/await* consente ai programmi di lavorare in background su calcoli potenzialmente pesanti, il che spesso si traduce in interfacce utente esenti da *glitch* (piccoli bug o tentennamenti nel codice) e/o trasferimenti di file di rete più veloci.
Durante la creazione di metodi, *TAP* ha alcune parole chiave specifiche. L'attività è lì per denotare metodi che non restituiscono alcun valore. Nel nostro elenco troveremo anche un'istanza *Task<int>*, questa espressione denota un metodo che deve restituire un numero intero. Le attività possono restituire qualsiasi tipo di variabile. Tutte le attività asincrone create saranno pianificate per l'esecuzione su thread gestiti dalla classe *ThreadPool*. Il comando *await Task.Run* avvia un'attività nel proprio thread utilizzando la classe *ThreadPool*. Ora, ci sono fondamentalmente tre modi per invocare questa meccanica servendosi della sintassi vista nel precedente listato, con: await Task.Run(()=> accettiamo metodi precedentemente dichiarati senza

parametri; si passa semplicemente il nome di un metodo. Con: await Task.Run(()=> Metodo(codice..)); ci serviranno metodi che accettano argomenti. Infine con: await Task.Run(()=> { codice..; return 0; }); potremo usare la sintassi basata su blocchi. Il codice di base è:

```
   async void Azione()
{
  await Task.Run(() => codice..);
}
```

Detto questo, come ultima raccomandazione, per evitare fastidiosi bug, è necessario prestare attenzione all'uso corretto di parentesi, parentesi graffe e altri caratteri speciali.

Quiz & esercizi

1) In C# è possibile utilizzare una stringa lunga più di cento righe mantenendo il testo con la medesima formattazione?
2) Creare un listato dove un numero intero è convertito in long
3) Completare le parti mancanti del seguente listato:

```
   public class Prova {
  .. static .... Esempio() {
       ..(int i ... 0; i < 2; ...++) {
           Console.WriteLine("Esempio: ...", i);
           Thread......(0);
       }
   }
```

4) In quale modo sincronizziamo i theread e perché?
5) A cosa serve l'istruzione *Thread.Sleep()* ?
6) Scrivere un listato utilizzando l'alternativa s *Thread.Sleep()*
7) Quale classe utilizzerò per la programmazione asincrona?
8) Completare la seguente riga: `a...t ....R..(() ... codice..);`
9) A cosa serve il simbolo chiocciola (@) nel codice C#?
10) Scrivere un listato che riporti la data attuale e i giorni che mancano all'estate.

Riassunto

Alla fine di questo capitolo, abbiamo imparato un bel po' su gli usi comuni per lo spazio dei nomi con *System.Globalization* e alcune delle sue classi, incluse *Calendar* e *CultureInfo*, incluse le operazioni di base sui file usando le classi *File* e *FileInfo.*

Nozioni di base sulla Garbage Collection (GC) e l'implementazione del multithreading di base in C#, inclusi alcuni dei metodi associati più rilevanti: *Join, Start, Sleep* e *Yield*.

Infine nozioni di base sulla programmazione asincrona usando il modello di progettazione basato su attività (TAP) e il suo meccanismo di attesa.

 # 4 – Approfondiamo JAVA

Scope e blocchi
Operazioni sui file
Il multithreading

Come già sappiamo Java è un linguaggio di programmazione orientato agli oggetti indipendente dalla piattaforma che utilizzeremo, modellato dai linguaggi C e C++ di cui mantiene le caratteristiche.
Questa indipendenza è ottenuta grazie all'uso della *Java Virtual Machine*, la quale è un componente essenziale per l'interazione delle applicazioni Java con il Web. Sostanzialmente traduce le istruzioni del codice generato dal compilatore in istruzioni eseguibili dalla macchina locale. La natura di linguaggio a oggetti di Java ci consente di sviluppare applicazioni utilizzando oggetti software o concettuali piuttosto che procedure e funzioni, il che consente al programmatore di scrivere codice stabile e riutilizzabile utilizzando il paradigma object oriented secondo il quale il programma è scomposto in concetti piuttosto che funzioni o procedure.
La sua stretta parentela con il linguaggio C a livello sintattico, fa si che un programmatore che abbia già fatto esperienza con linguaggi come C, C++, Perl sia facilitato nell'apprendimento di Java.
Inoltre, questo linguaggio contiene alcune caratteristiche che lo rendono particolarmente adatto alla programmazione di applicazioni Web (client-side e server-side).

Ricordiamoci che un'applicazione Java è composta di una o più parti di codice sorgente definite come *unità di compilazione*, a sua volta suddivisa in:

- *fully qualified name* il nome del pacchetto riferito all'unità di compilazione.

- il nome dei pacchetti importati, utili a eseguire funzionalità aggiuntive.
- le dichiarazioni delle classi.

Da questo insieme ne risulta il nostro programma, con le sue regole di sintassi che abbiamo in parte esplorato.
Un argomento importante valido per tutti i linguaggi di programmazione riguarda l'ambito (conosciuto meglio come lo *scope*) in cui le nostre variabili sono valide.

Lo scope

Come in parte accennato, in Java le variabili sono accessibili solo all'interno della regione in cui sono state create. Questo è quello che definiamo *scope* o ambito. Essenzialmente ne esistono di due tipologie:

- *Ambito del metodo* dove le variabili dichiarate direttamente all'interno di un metodo sono disponibili ovunque partendo dalla riga di codice in cui sono state dichiarate:

```
   public class Main {
  public static void main(String[] args) {

   // Prima di questa riga la variabile a non sarà valida
    int a = 1;
    System.out.println(a);
  }
}
```

- *Ambito di blocco* dove le variabili dichiarate si riferiranno solamente a tutto il codice racchiuso tra le parentesi graffe {}. Di conseguenza le variabili dichiarate all'interno di blocchi saranno accessibili solo dal codice racchiuso tra le parentesi, seguente la riga in cui è stata dichiarata la variabile:

```
   public class Main {
 public static void main(String[] args) {
   { // a è valida solo dopo la sua dichiarazione
     int a = 1;
     System.out.println(a);
```

```
   }
 // Dopo la fine del blocco, a non sarà valida
 }
}
```

Il blocco try catch e finally

Parliamo della gestione degli errori, la quale ci sarà utile anche per altre cose, infatti, come vedremo, gli sviluppatori utilizzano spesso il blocco *try catch,* ad esempio, come via di uscita da un'iterazione. Il codice all'interno del blocco *try* è monitorato per rilevare eventuali errori e, quando se ne verifica uno, il programma passa al codice all'interno del blocco *catch*. Quest'approccio viene anche definito gestione delle eccezioni. Vediamone una semplice dimostrazione nel prossimo listato:

```
   public class Prova {
public static void main(String[ ] args) {
  int numero = 8;
  try {
     numero /= 0;
    System.out.println(numero);
    } catch (Exception e) {
    System.out.println("errore..");
  }
  }
}
```

Oltre alla gestione degli errori con *try catch* avremo a disposizione un blocco opzionale chiamato *finally*, eseguito indipendentemente dal fatto che si verifichi un'eccezione nel precedente blocco o meno. Ci servirà per garantire che eventuali azioni critiche specifiche siano eseguite in un preciso momento. Osserviamo la prova sul campo di *finally*:

```
   public class Prova
   {
  public static void main (String[] args)
  {
  try {
```

```
        int vettore[] = new int[4];
        vettore[5] = 14;
       }
    catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("errore..");
      }
   finally {
   System.out.println("opzione due");
     }
   }
}
```

In questo listato prima impostiamo un vettore (array) e poi facciamo in modo che sia riportata un'eccezione, infine inseriamo l'istruzione *finally* la quale sarà eseguita indipendentemente da ciò che è accaduto prima.

La gestione delle eccezioni

La gestione delle eccezioni ci servirà per tenere sotto controllo eventuali situazioni anomale o di bug, dette appunto eccezioni, che potrebbero verificarsi durante l'esecuzione della nostra applicazione. Java ne mette a disposizione diverse che vedremo in un elenco più avanti. Per usufruire di questa tecnica utilizzeremo l'istruzione *throw* da cui otterremo messaggi di errore più dettagliati che indicano le righe problematiche nel nostro codice. Vediamo subito un listato con l'uso dell'istruzione *throw*:

```
   public class Carbonara {


   static void controllaIngredienti(String ingredienti) {
   if (!ingredienti.equals("Guanciale")) {
   throw new RuntimeException("Solo guanciale per favore!");
   }

  else {
    System.out.println("Ottimo guanciale, bene!");
   }
  }
```

```
    public static void main(String[] args) {
  controllaIngredienti("Guanciale");
  controllaIngredienti("Pancetta");
 }
}
```

In questo breve listato, faremo in modo che nella carbonara sia utilizzato il guanciale, piuttosto che la pancetta, quindi creeremo un metodo per controllare gli ingredienti, cioè *controllaIngredienti,* e in caso si presenti la pancetta, sarà visualizzato un messaggio di errore attraverso il meccanismo delle eccezioni.
Le eccezioni in Java sono fondamentalmente segnali che qualcosa è andato storto durante l'esecuzione di un programma, ne sono disponibili numerosi tipi, tutti a disposizione del programmatore per diverse evenienze. Il termine *RuntimeException* si riferisce a un tipo generale di errore; abbiamo scelto di utilizzarlo nel caso non incontrassimo il guanciale. Poiché eseguiamo il metodo *controllaIngredienti()* sia per il "Guanciale" che per la "Pancetta", otteniamo sia un messaggio senza eccezioni, sia uno con un'eccezione, vediamo insieme l'output del listato:

```
   Ottimo guanciale, bene!
Exception in thread "main" java.lang.RuntimeException: Solo guanciale per favore!
at Carbonara.controllaIngredienti(Carbonara.java:4)
at Carbonara.main(Carbonara.java:13)
```

Adesso studieremo un elenco della maggior parte delle eccezioni disponibili in Java, in modo da non trovarsi impreparati, nel malaugurato caso appaiano:

- *ArrayIndexOutOfBoundsException*: Eccezione generata quando si tenta di accedere a un elemento di un vettore (array) attraverso un indice errato.
- *ArithmeticException*: Lanciato quando un'operazione aritmetica fallisce.
- *ClassCastException*: Errore generato quando si tenta di fare un *cast* non permesso ad esempio di un oggetto che non è istanza della classe.
- *ClassNotFoundException*: Errore che è generato quando si tenta di accedere a una classe che non esiste.
- *NullPointerException*: Generato quando il riferimento all'oggetto di una

variabile è *null.*
- *NumberFormatException*: Generato quando un metodo non è in grado di convertire una stringa in un numero.
- *NoSuchFieldException*: Lanciato durante l'indirizzamento di una variabile che manca da una classe.
- *StringOutOfBoundsExceprion*: Si genera nel tentativo di accedere a una posizione inesistente in una stringa.
- *RuntimeException*: Come visto nel listato precedente questa eccezione è utilizzata per un errore generico durante l'esecuzione del programma.

Oltre a questo elenco è bene sapere che esistono essenzialmente due categorie di eccezioni in Java: *selezionate* e *di runtime*. Le prime saranno utilizzate quando si genereranno errori rilevati fuori della portata del programma, come ad esempio problemi con operazioni sui file o con trasferimenti di rete. Le eccezioni non controllate, a volte chiamate anche *eccezioni di runtime*, gestiscono problemi all'interno della logica di programmazione come argomenti non validi e operazioni non supportate. Riportiamo in una tabella le diverse tipologie in modo da poter avere una visione più chiara:

	Eccezioni di runtime	***Eccezioni selezionate***
Livello di allarme	*Preoccupante*	*Massimo*
Area d'effetto	*Nel nostro listato*	*Al di fuori del listato*
Momento in cui il problema si manifesta	*All'esecuzione del programma*	*Mentre si compila il programma*
Possibili errori rilevati	*Errori di sintassi, con operazioni matematiche, di logica, con le classi*	*Problemi con la connessione, con la rete, con fonti esterne*

In pratica, la dettagliata gestione delle eccezioni del nostro Java, sarà utilissima anche per produrre del codice che sia il più pulito possibile.

La gestione dei file

Abbiamo visto nel capitolo precedente come le più svariate operazioni riguardanti i file siano possibili e ampliamente supportate da C#. Ovviamente sappiamo già che si riferiscono alla lettura, scrittura e cancellazione di dati archiviati su un dispositivo come un disco rigido, un'unità a stato solido (SSD) o altri sistemi di archiviazione. Con Java avremo il medesimo supporto. Per impostare un progetto utile per operare sui file, dovremmo aggiungere il pacchetto *import java.io.File* all'inizio di un listato. Vediamolo meglio in un listato focalizzato sull'argomento, dove andremo a creare un semplice file di testo:

```
   import java.io.File;
   public class Prova {
   public static void main(String[] args) {
   File FileDiTesto = new File("SalutiDaMario.txt");
 try {
       boolean ok = FileDiTesto.createNewFile();
       if (ok == true) {
       System.out.println("Il file " + FileDiTesto.getName() + " è
stato creato correttamente!");
     }
  else {
       System.out.println("Il file è già stato creato.");
  }
  }
  catch(Exception e) {
  System.out.println("Guru meditation... ");
  }
}
}
```

Attraverso il pacchetto *File* andremo a creare il file *SalutiDaMario.txt* (senza un contenuto), poi con il metodo *createNewFile* e l'utilizzo di una variabile booleana (*ok* corrisponde a *true*) stamperemo su schermo il risultato.

Il pacchetto *java.io.File* offre molto di più rispetto a quello che abbiamo appena visto. Di seguito alcuni metodi comuni per le operazioni sui file:

- *canRead()* Prova se un file è leggibile o meno.
- *canWrite()* Verifica se un file è scrivibile o meno.
- *createNewFile()* Crea un nuovo file.
- *delete()* Elimina un file.
- *exists()* Verifica se un file esiste.
- *getAbsolutePath()* Recupera il percorso completo di un file.
- *lenght()* Restituisce la dimensione di un file in byte.
- *mkdir()* Crea una nuova directory.

Proviamo adesso a creare un file di testo e ad accedervi:

```
   import java.io.*;
public class Prova{
public static void main(String args[]) {
 String data = "Buongiorno da Mario!";
   Try {
         FileWriter uno = new FileWriter("mario.txt");
         uno.write(data);
         uno.close();
         File uno = new File("mario.txt");

  // Creiamo un nuovo FileInputStream chiamato "prova"
    FileInputStream prova = new FileInputStream(uno);
System.out.println("Il file " + uno.getName() + " contiene:");
int numero=0;

   // Creiamo un ciclo per vedere ogni carattere in "mario.txt"
while((numero = prova.read())!=-1)
{
System.out.print((char)numero);
}


  prova.close();
 }
```

```
  catch (Exception e) {
   e.getStackTrace();
  }
}
}
```

In questo listato abbiamo messo parecchia carne al fuoco. Prima di tutto, *import java.io.*;* ci servirà per fornire l'accesso a tutte le classi disponibili nel pacchetto *java.io*.
Successivamente, abbiamo la classe *Filewriter*. Come si può prevedere, essa fornisce i mezzi per scrivere dati e creare nuovi file. Istanziamo un oggetto da *Filewriter*, chiamato *uno* e gli assegniamo il file *mario.txt*, fornendo sia un nome al file sia un percorso della directory.
Scendendo nell'elenco, evochiamo il metodo *write* dalla classe *Filewriter*, passandogli il messaggio dai dati della stringa. Il nostro *mario.txt* ora contiene un messaggio di testo. Successivamente, eseguiamo il metodo *close*; questo deve essere fatto affinché i metodi successivi ottengano l'accesso al file di testo.

Dopo apriamo di nuovo il nostro file, questa volta per stamparne il contenuto sullo schermo. Lo facciamo istanziando un oggetto usando la classe File di Java, cioè *File uno = new File("c:\\mario.txt")*.

La riga *FileInputStream prova = new FileInputStream(uno)* ci dà accesso alla classe riguardante il flusso di input in un file, permettendone una lettura passo passo, anche incrementando solo un byte per volta.

Nel nostro caso, questo significa leggere e visualizzare un file di testo (ad esempio, mario.txt) un carattere alla volta. Per questo, implementiamo un ciclo *while* che utilizza un metodo (cioè, *read*) dalla classe *FileInputStream*.

Stampiamo il contenuto di *mario.txt* (un carattere alla volta, secondo com'è implementato l'accesso ai file *FileInputStream*). Un valore negativo (-1) segnala la fine del file, interrompendo il ciclo *while*. Dopo di questo, è solo tempo di pulizia. Chiudiamo il file e usciamo dal programma.

Detto ciò, abbiamo ancora qualche istruzione da vedere. Proveremo a servirci degli attributi dei file, come ad esempio scoprire la loro dimensione, ma anche ordinarne la cancellazione:

```
import java.io.*;
public class Prova {
```

```
   public static void main(String args[]) {
// Apriamo il file per l'accesso
File mario = new File("c:\\mario.txt");
// Iniziamo un blocco try catch
try {
    // Per prima cosa controlliamo se il file è presente
    if (mario.exists()) {
    // Se lo è, mostriamo la conferma
    System.out.println("Il file " + mario.getName() + " c'è.");
    // Evochiamo il metodo length() per la dimensione del file
   System.out.println("Dimensioni in kb " + mario.length());
   // Controlliamo se la dimensione del file supera i 25 byte
   if(mario.length()>25) System.out.println("File grande.");
   // Visualizziamo le informazioni di lettura/scrittura
   if (mario.canRead()) System.out.println("File leggibile.");
   if (mario.canWrite()) System.out.println("File scrivibile.");
  System.out.println("Eliminiamo " + mario.getName() + "...");
   // Evochiamo il metodo delete() sul povero mario
   mario.delete();

  // Se il file non esiste, lo segnaliamo con un messaggio
   } else System.out.println("Impossibile trovare il file.");
  }
catch (Exception e) {
                e.getStackTrace();
     }
 }
}
```

Avremo sicuramente notato che l'implementazione delle operazioni di base sui file in Java è piuttosto semplice. Ovviamente non è tutto qui, ci sono ancora diverse cose che si possono fare, proviamole in questo listato:

```
   import java.io.*;
   class FileStreamTesting {
   public static void main(String args[]) {

   // Iniziamo il blocco try catch
```

```
   try {
  // Definiamo una stringa come nome per una directory
  String qualsiasi = "Cartella";
  File uno = new File(qualsiasi);
  // Creiamo una nuova directory in C:
  uno.mkdir();
  // Creiamo un array di sei caratteri
  char Nome[] = {'M','a','r','i','o','!'};
  // Creiamo un'istanza dalla classe OutputStream
   OutputStream x = new FileOutputStream("c://Cartella// nome.txt");
   for(int y = 0; y < Nome.length ; y++) {

   // Scriviamo ogni carattere dal nostro array nel file
x.write(Nome[y]);
   }
   x.close();

   } catch (IOException e) {
  System.out.print("Si è verificato un errore.");
  }
}
   }
```

Gli ultimi due listati hanno utilizzato le classi di streaming dei file. Il primo ha utilizzato un metodo della classe *InputStream* per stampare il testo sullo schermo. Mentre nel secondo, abbiamo un'istanza di *OutputStream* che scrive un array di sei caratteri (cioè il contenuto di *Nome[]*) in un file.

Piccola annotazione finale: quando abbiamo specificato un percorso della directory, i due listati avevano una notazione diversa: nel primo una barra rovesciata (cioè *c:\\mario.txt*), mentre nell'altro, siamo andati con la barra in avanti, come in *c://Cartella*. Entrambi gli approcci sono ampliamente validi e funzionali.

Il multithreading

Abbiamo parlato del multithreading e sappiamo che rende molto più rapidi i nostri algoritmi. Java supporta ampliamente il multithreading, che sostanzialmente si riferisce alla divisione di un programma in più parti che sono in esecuzione contemporaneamente. Anche qui dovremo essere molto

attenti, poiché talvolta il multithreading può diventare "*concorrenza*", ciò accade quando vengono eseguiti due o più thread contemporaneamente.
I thread multipli, come abbiamo già visto, sono un concetto che va di pari passo con i core del processore di un computer, detta CPU. Una CPU di vecchia generazione, ad esempio, precedente al 2006 ha in genere un unico core, mentre attualmente sia gli smartphone, sostituti contemporanei dei notebook, che i computer in genere, difficilmente hanno meno di due o quattro CPU, anzi, oramai la fascia di prezzo media comincia ad averne otto (octa-core), più altrettante virtuali nel caso dei computer. I thread, come gli eventi o il ciclo di vita di una funzione, possono essere classificati in vari stati, che quindi costituiscono appunto il ciclo di vita del thread. Esaminiamo insieme queste fasi al dettaglio:

- **nuovo** (*new*): la nascita di un nuovo thread.
- **eseguibile** (*runnable*): come suggerisce il nome, un thread in questa fase è pronto per eseguire le sue attività.
- **in attesa** (*waiting*): un thread entra in questa fase quando sta lavorando con altri thread e ha per il momento completato la sua elaborazione, lasciando che altri prendano le redini. Di conseguenza è in attesa dell'ok da parte degli altri per riprendere il suo carico di lavoro.
- **in attesa a tempo**: i thread entrano in questa fase d'inattività quando hanno completato le loro attività per il momento, ma sono necessari in futuro in un punto o momento specifico del programma.
- **terminato**: Quando un thread ha completato tutti i suoi compiti, termina.

Esiste un ulteriore stato che riguarda i thread e lo sfruttamento della CPU: la cosiddetta *fame,* com'è denominata nel gergo di Java.
Abbiamo imparato che con il multithreading, le attività all'interno di un unico programma sono suddivise in singoli thread di esecuzione, ebbene, proprio in questa fase potrebbe capitare che sopraggiunga la "*fame*". Essa entra in vigore quando a un thread (o gruppo di essi) non sono fornite abbastanza risorse CPU da essere eseguito correttamente, perché queste sono monopolizzate da altri thread più avidi. Un sistema di distribuzione equa di quelle preziose risorse della CPU è chiamato appunto *equità*. Uno dei principali colpevoli della *fame* in Java risiede in realtà nella sincronizzazione basata sui cosiddetti *"blocchi"*. Per questo motivo la corretta definizione delle priorità nei thread utilizzando i *blocchi* sarà il metodo principale per affrontare la fame. Lo vedremo poco più avanti. Infine è bene sapere che

anche i thread sono oggetti e di conseguenza potremo sfruttarli come tali ad esempio con la classe *Thread* e l'interfaccia *Runnable* facenti parte di *java.lang*.
Adesso diamo un'occhiata a un programma che ci mostra il multithreading in Java:

```
   class Prova extends Thread {
 public Prova(String stringa){
  super(stringa);
  }
  public void run() {
    for (int i = 0; i < 6; i++) {
     System.out.println("Compiti "+ i + " del " + getName());
      }
  System.out.println(getName() + " concluso.");
 }
}
 public class Seconda {
   public static void main(String[] args){
    new Prova("Thread colazione").start();
    new Prova("Thread lavoro").start();
    new Prova("Thread divertimento").start();
    new Prova("Thread dormire").start();
  }
}
```

Per prima cosa un'annotazione: per far funzionare correttamente il programma dovremo dividerlo in due file distinti per ogni classe. *Seconda.java* sarà di partenza, *Prova.java* sarà di supporto, e ovviamente ognuna conterrà la sua porzione di codice. Il principio sarà valido per tutti i programmi Java.
In questo listato notiamo che la parola chiave, *extends* già vista con PHP. In Java, funziona esattamente allo stesso modo, cioè eredita i metodi e i costruttori di un'altra classe. Nel nostro caso, *Prova* riceve la funzionalità della classe *Thread* (denominata *Java.lang.Thread* in Java SDK). Poi, con *super* potremo accedere ai dati della classe genitore. Poi, attraverso un loop ogni thread per sei volte visualizzerà i suoi passaggi su schermo. Infine, più avanti nel nostro elenco, vedremo in azione alcuni metodi derivati da thread,

vale a dire i metodi *getName()* e *start()*. Il primo restituisce il nome di un thread, mentre il secondo lo esegue.
Sebbene l'output del listato sia presentato in modo ordinato, i quattro thread all'interno sono eseguiti contemporaneamente secondo l'approccio del multithreading. Tuttavia, eseguendo l'elenco più volte di seguito, scopriremo che i diversi thread mostrano il loro output in ordini diversi. Questo è perfettamente normale per un programma non sincronizzato.
La potenza di elaborazione fornita da una CPU è una risorsa limitata. Sono quindi assegnati più livelli di priorità ai thread di esecuzione che aiuteranno il nostro sistema operativo a procedere alla loro elaborazione. Questo è un meccanismo per lo più automatizzato che dipenderà dalla disponibilità nelle risorse del nostro computer in quel momento.
Soprattutto nei progetti più grandi, più thread dovranno assolutamente lavorare insieme. Per fortuna, Java prende seriamente quest'approccio. La sincronizzazione tra i thread si riferisce al loro controllo sull'accesso promiscuo a una risorsa condivisa. Perché sincronizzarli? Perché altrimenti diversi thread potrebbero tentare di accedere agli stessi dati contemporaneamente. Questo non andrebbe assolutamente bene per l'ordine dei dati e la stabilità in genere.
Per implementare la sincronizzazione in Java in modo semplice, si utilizza l'istruzione *synchronized*. Quest'approccio può essere utilizzato su tre livelli: metodi statici, metodi d'istanza e blocchi di codice. Per esempio, nel listato precedente, solamente con una leggera modifica nella dichiarazione di funzione, cioè aggiungendo a *public void run ()* :

```
public synchronized void run()
```

otterremo un output più ordinato dal programma.
Non dovremo nemmeno sincronizzare interi metodi. In pratica possiamo scegliere dove usare la tecnica, infatti le porzioni di codice sincronizzato associano un *blocco del monitor* a un oggetto. Tutti i blocchi sincronizzati della stessa varietà di oggetti consentiranno quindi a un solo thread di eseguirli contemporaneamente. Per aggiungere una *sincronizzazione* a livello di blocco, indicheremo quest'ultimo all'interno di un metodo con *synchronize(this) { ... } ,* dove la parola chiave *this* si riferirà a un metodo di cui farà parte.

La proprietà di blocco

In Java, qualunque oggetto software ha insito in se ciò che è noto come la cosiddetta *proprietà di blocco,* o meglio, il *blocco sincronizzato*. Questo meccanismo consente solo ai thread a cui è stata specificamente concessa l'autorizzazione di accedere ai dati degli oggetti. Dopo che un thread ha finito di lavorare con un oggetto, esegue un rilascio del blocco, che lo riporta in uno stato di disponibilità. L'uso dei blocchi sincronizzati è la forma più sofisticata e funzionale in Java rispetto all'approccio basato sul *metodo sincronizzato*. Essenzialmente si differenziano giacché il primo ferma il codice solamente all'interno del blocco, mentre il metodo sincronizzato blocca tutto l'oggetto. Vediamo in dettaglio le principali differenze:

Blocco sincronizzato	***Metodo sincronizzato***
Blocca solo l'oggetto tra parentesi.	Blocca l'intero oggetto.
Viene utilizzata l'espressione come un blocco.	Utilizza l'istruzione che riceve il metodo come blocco: *this* per i metodi non statici e la classe per quelli statici.
Il blocco è mantenuto solo nel suo utilizzo.	Il blocco è mantenuto per tutto il metodo.
Il blocco è rilasciato solo alla sua naturale scadenza.	Il blocco può essere rilasciato anche da un'eccezione.

I metodi dei thread

Ora sarebbe un buon momento per ricapitolare alcuni metodi chiave trovati in *Java.lang.Thread*. Questi e molti altri metodi ci diventeranno sempre più familiari man mano che procederemo con questo libro. Adesso andiamo a vedere in dettaglio alcuni metodi compresi nella classe Thread in Java (*Java.lang.Thread*):

- **currentThread()** Restituisce un riferimento del thread che è attualmente in esecuzione.
- **getName()** Restituisce il nome del thread.
- **getPriority()** Riporta la priorità del thread.
- **getState()** Riporta lo stato del thread (in attesa, terminato, e così via).
- **interrupt()** Arresta un thread.
- **interrupted()** Riporta se un thread è interrotto o meno.
- **notify()** Riprende un singolo thread che era stato di attesa.
- **setPriority()** Imposta la priorità di un thread con un numero compreso tra 1 e 10: Mario.setPriority(7); Prova.setPriority(2);
- **sleep()** Mette un thread in sospensione per una quantità specifica di millisecondi: Thread.sleep(200);
- **start()** Esegue un thread: Mario.start();
- **wait()** Forza un thread ad attendere finché un altro non esegue il proprio compito.

Misurare il tempo

Attraverso *LocalDate* potremo tenere traccia dello scorrere del tempo. Inoltre, la classe *java.time.format.DateTimeFormatter* ci sarà molto utile per visualizzare le date secondo un modello di formattazione personalizzato. Osserviamo subito un programma che dimostra il recupero della data e la sua formattazione:

```
   import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
public class Prova {
  public static void main(String args[]) {
    LocalDate tempo = LocalDate.now();
    LocalDate ieri = tempo.minusDays(30);
    LocalDate domani = tempo.plusDays(60);
```

```
    System.out.println("La data di oggi è " + tempo);
    System.out.println("Un mese fa era " + ieri);
    System.out.println("Tra due mesi sarà.. " + domani);

  // Creiamo e vediamo tre diversi modelli di formattazione
      System.out.println("Data odierna, primo formato: "+tempo.
format(DateTimeFormatter.ofPattern("dMuuuu")));
     System.out.println("Data odierna, secondo formato: "+tempo.
format(DateTimeFormatter.ofPattern("MMM d uuuu")));
    System.out.println("Data odierna, terzo formato: "+tempo.
format(DateTimeFormatter.ofPattern("d.MMM.uu")));
 }
}
```

Ed ecco, oltre a quelli appena visti, altri marcatori di formattazione del modello temporale da *java.time.format.DateTimeFormatter*:

d/dd Giorno del mese
W Settimana in un mese (da 0 a 5)
MM Mese
K Ora in un giorno (da 1 a 24)
uuuu/uu Anno (completo o ultime due cifre)
K Ora in un giorno, AM/PM (0–11)
w Settimana in un anno (da 1 a 53)
E Giorno della settimana

Chiaramente non avremo a disposizione solamente l'unità di misura relativa il giorno, i mesi o gli anni. Vediamo subito un listato che dimostra l'uso di altri dei metodi nella classe *LocalDate*:

```
   import java.time.LocalDate;
import java.time.LocalTime;
public class Prove {
public static void main(String[] args) {

   // Memorizza la data corrente
LocalDate adesso = LocalDate.now();
System.out.println("Tra 8 settimane: " + adesso.plusWeeks(8));
System.out.println("L'anno scorso: " + adesso.minusMonths(12));
System.out.println("Venti anni fa: " + adesso.minusYears(20));
```

```
System.out.println("Tra cento anni: " + adesso.plusYears(100));

   // Memorizza una data futura in domani
LocalDate domani = LocalDate.of(2028,1,1);
// Memorizza una data passata in ieri
DataLocale ieri = DataLocale.of(1728,1,1);
// Controlla gli anni bisestili
System.out.println("\nL'anno " + ieri.getYear() + " è stato
bisestile? "+ ieri.isLeapYear());
System.out.println("L'anno " + adesso.getYear() + " è
bisestile? " + adesso.isLeapYear());
System.out.println("L'anno " + domani.getYear() + " è un
anno bisestile? "+ domani.isLeapYear());
}
}
```

Abbiamo quindi scoperto come trovare gli anni bisestili con Java. Ma ancora non abbiamo visto le molte altre possibilità offerte della classe *Calendar*, proviamone alcune in questo breve listato d'esempio:

```
   import java.util.Calendar;
public class Prove {
public static void main(String[] args) {
// Crea un oggetto utilizzando la classe Calendar
Calendar data = Calendar.getInstance();

   System.out.println("Data corrente: " + data.getTime());
data.add(Calendar.DATE, -10);
System.out.println("Dieci gg. fa era: " + data.getTime());
data.add(Calendar.MONTH, 5);
System.out.println("Cinque mesi avanti: " + data.getTime());
calendar1.add(Calendar.YEAR, 8);
System.out.println("Otto anni nel futuro: " + data.getTime());
calendar1.add(Calendar.YEAR, -270);
System.out.println("270 anni fa: " + data.getTime());
}
}
```

Inizialmente creiamo un oggetto servendoci della classe *Calendar*. Poi procederemo con dei salti temporali, prima eliminiamo una decina di giorni dalla data che abbiamo visualizzato, poi salteremo avanti di cinque mesi, di otto anni e infine torneremo indietro di ben duecentosettanta anni.

La classe Locale

Normalmente avremo problemi per gestire un'applicazione funzionante in diverse nazioni con lingue e impostazioni locali differenti, compresa la progettazione del software in modo che possa essere localizzato senza importanti revisioni. Fortunatamente, con Java, avremo a disposizione i mezzi per localizzare date, valute e numeri in modo piuttosto dinamico.
La classe Java denominata *Locale* offre alcune opzioni per localizzare i nostri progetti. Effettivamente serve per dare ai programmatori l'opportunità di presentare messaggi o altre parti di un programma localizzate. Tecnicamente è utilizzata per identificare gli oggetti pur non essendo essa stessa un contenitore per oggetti.
La classe *Locale* funziona con tre dimensioni: lingua, paese e variante. I primi due sono auto esplicativi, mentre la variante è utilizzata per cose come il tipo e/o la versione di un sistema operativo su cui è eseguito il programma.
Effettivamente, in Java un'applicazione può avere diverse localizzazioni attive contemporaneamente. Ad esempio, è possibile combinare un formato data italiano e un formato numerico giapponese. Pertanto, Java è una buona scelta per creare applicazioni veramente multiculturali.
Diamo quindi un'occhiata a una dimostrazione su come usare la localizzazione con un listato che dimostra le tre dimensioni principali delle impostazioni locali in Java (lingua, paese e variante):

```
   import java.util.Locale;
public class Prova {
public static void main(String[] args) {
// Creiamo una lingua Italiana generica
Locale es1 = new Locale("it");
// Lingua Italiana impostata nel Regno Unito
Locale es2 = new Locale("it", "UK");
// Italiano, lingua Inglese a Mumbai
Locale es3 = new Locale("it", "UK", "Mumbai");
System.out.println("Locale 1. " + es1.getDisplayName());
```

```
System.out.println("Locale 2. " + es2.getDisplayName());
System.out.println("Locale 3. " + es3.getDisplayName());

   // Recuperiamo le informazioni del sistema operativo creando
un'altra istanza di Locale con il metodo getDefault()

   Locale informazioni = Locale.getDefault();

   System.out.println("Lingua del sistema operativo: " +
informazioni.getDisplayLanguage());
System.out.println("Paese del sistema operativo: " +
informazioni.getDisplayCountry());
}
}
```

Iniziamo creando tre diversi oggetti e visualizziamo alcuni dei loro contenuti usando il metodo *getDisplayName()* disponibile nella classe *Locale*.
Evochiamo anche un quarto oggetto per esaminare le impostazioni locali del sistema operativo dell'utente. Innanzitutto, il metodo *getDefault()* è utilizzato per recuperare questi dati e inserirli nell'oggetto che abbiamo chiamato *informazioni*. Infine, per visualizzare tutte le informazioni che ci interessano, abbiamo eseguito altri due metodi, in altre parole *getDisplayLanguage()* e *getDisplayCountry()*.

Typecasting

Anche in Java avremo la possibilità di convertire i dati. Essenzialmente vi sono due possibilità:
- *Narrowing Casting* in pratica la conversione manuale, da un tipo di dato grande a uno piccolo.
Da *byte* a *-> short -> char-> int -> long -> float -> double*
- *Widening Casting* parliamo della conversione automatica, da un tipo di dato piccolo ad uno grande.
Da *double* a *-> float -> long-> int -> char -> short -> byte*
Vediamo due esempi:

```
   public class Main {
  public static void main(String[] args) {
```

```
    double numero = 3.33d;
    int intero = (int) numero; // double -> int

  System.out.println(numero);   // 3.33
   System.out.println(intero);      // 3
 }
}
```

Il secondo:

```
   public class Main {
  public static void main(String[] args) {
    int intero = 3;
    double numero = intero; // int -> double

  System.out.println(intero);      // 3
   System.out.println(numero);   // 3.0
 }
}
```

L'istruzione Switch

Questa sintassi si rivelerà molto utile per eseguire parti di listato. Viene utilizzata assieme a *case* attraverso un confronto, il quale riporta ad una porzione di codice predeterminata:

```
   int dolci = 8;
switch (day) {
 case 1:
   System.out.println("Torta della nonna");
   break;
 case 2:
   System.out.println("Gelato");
   break;
 case 3:
   System.out.println("Cornetto");
   break;
 case 4:
   System.out.println("Bignè alla crema");
```

```
    break;
  case 5:
    System.out.println("Biscotti");
    break;
  case 6:
    System.out.println("Nutella");
    break;
  case 7:
    System.out.println("Miele");
   break;
  case 8:
    System.out.println("Cioccolato");
break;
}
```

```
    Output: Cioccolato
```

La ricorsione

Tecnica abbastanza complessa per eseguire operazioni che, in maniera diretta o non, richiamano se stesse. Essenzialmente nella ricorsione è invocato un metodo mentre questo è in esecuzione. Ciò significherà che il metodo chiamante resterà in attesa di quello chiamato e del suo risultato:

```
    public class Main {

   public static void main(String[] args) {
    int risultato = somma(5);
    System.out.println(risultato);
  }
  public static int somma(int numero) {
    if (numero > 0) {
      return numero + somma(numero - 1);
    } else {
      return 0;
    }
  }
}
```

In questo caso il risultato sarà 15. Come ci siamo arrivati? in pratica, quando evochiamo *somma()* aggiungiamo il parametro *numero* alla somma di tutti i numeri inferiori a *numero*, riportando poi, dopo tutti i calcoli, il numero 15, quindi 5+4+3+2+1, o meglio: 5 + somma(4) + somma(3) e così via. Arrivati a zero, la funzione non potrà più evocare se stessa e quindi il programma si fermerà riportando il risultato.
Bisognerà fare molta attenzione a utilizzare questa tecnica, evitando di scrivere del codice con funzioni che non hanno termine o che siano estremamente pesanti da elaborare, di contro questa funzionalità si presta a operazioni matematiche complesse.

Quiz & esercizi

1) Correggere questo listato che restituisce un errore:

```
   public class Main {
  public static void main(String[] args) {
   int x = 1;
   System.out.println(y);
   int y = 2;
  }
}
```

2) Nella gestione degli errori che utilità ha *finally* ?

3) Se volessimo verificare l'esistenza di un file in una cartella quale metodo andremo a utilizzare?

4) A cosa serve l'asterisco dopo l'importazione del pacchetto (cioè ad esempio import pacchetto.xx.*) ?

5) Modificare con l'inserimento di una singola istruzione/sintassi il seguente listato in modo che produca un output più ordinato:

```
   class Prova extends Thread {
 public Prova(String a){
  super(a);
  }
  public void run() {
     for (int i = 0; i < 4; i++) {
      System.out.println("Passaggio "+ i + " di " + getName());
      }
```

```
  System.out.println(getName() + " finito.");
 }
}
 public class Esempio {
   public static void main(String[] args){
    new Prova("Thread uno").start();
    new Prova("Thread due").start();
  }
}
```

6) Con il pacchetto *LocalDate* modificare il formato di visualizzazione standard della data in modo che mostri anno, giorno e mese.
7) In Java è possibile il Narrowing Casting da numero intero a numero in virgola mobile?
8) Correggere e completare il seguente listato:

```
   public ... Main {
  public static ... ....(String[] args) {
    ... a = 5.55555;
    int b = (int) a;

  ......println(b);
   ......println(int);
 }
}
```

9) A cosa serve la ricorsione?
10) Un listato Java con cinque file *.java quante classi avrà al suo interno?

Riassunto

Anche qui siamo partiti dalle nozioni basiche, cioè dall'ambito di validità delle variabili e da come aprire, creare ed eliminare file.
Poi siamo passati al multithreading e a com'è implementato, assieme alla sincronizzazione di base.

Abbiamo esplorato anche il funzionamento del blocco *try catch*, cosa significa gestione delle eccezioni e come sfruttarle. La differenza tra eccezioni selezionate e non controllate, infine alcuni utilizzi delle classi

concernenti la data, le basi della localizzazione e un accenno sulla tecnica della ricorsione.

 # 5 - Approfondiamo Python

Operazioni sui file
Le espressioni regolari
Multithreading e multiprocessing

Di Python forse non sappiamo che attualmente è uno dei linguaggi più popolari ed è utilizzato da grandi aziende come Google, Meta (Facebook e Instagram), Spotify e molte altre.

I motivi di questo successo sono diversi: innanzitutto la sintassi è piuttosto chiara e semplice, molto vicina all'inglese, offrendo quindi una miglior "leggibilità del codice" semplificando la vita a coloro che si approcciano per la prima volta alla programmazione. Inoltre, la sua sintassi ne agevola notevolmente il debug. Oltre alla presenza di un favoloso insieme di librerie standard incluse in Python, ci sono numerosissime librerie sviluppate da terze parti, tra le tante troviamo *requests*, *beautifulSoup*, *pipenv*.

Python è multipiattaforma, ciò significa che qualsiasi programma potrà girare sia su Windows, su Linux che su IOS (ovviamente a patto che Python sia installato sulla macchina). Inoltre è completamente open source e vanta di una grandissima community.

Inizieremo con le tuple, la gestione dei file, passando al multithreading e ad altri argomenti più avanzati nel prosieguo del capitolo.

Lo scopo è di crearci una solida base su alcuni dei meccanismi fondamentali di Python su cui basarci per le nostre esigenze future.

Nota importante, probabilmente lo avrete già sperimentato, in ogni caso lo ribadisco: quando proverete i listati di Python sicuramente avrete problemi con la corretta indentazione. Purtroppo, anche se stampati correttamente, gli interpreti del linguaggio non acquisiranno correttamente gli spazi, che come sapete equivalgono al punto e virgola, in pratica

all'indicazione per la prossima istruzione o alla corretta separazione tra di esse. Ricordatevi di usare sempre il tasto *tab* per una precisa indentazione. Con questo linguaggio dovrete abituarvi a tutto ciò, salvo che non installiate un buon software con un buon linter, come quelli indicati all'inizio del libro. Abbiate pazienza e perseveranza e riuscirete a utilizzare correttamente la tecnica dell'indentazione.

Le Tuple

In Python abbiamo la possibilità di lavorare con gli array o insiemi (con il modulo *array*) in vari modi, infatti, oltre ai classici *list*, *set* e *dictionary* abbiamo le *tuple*, che si differenziano per il fatto di essere delle raccolte non modificabili e, ovviamente, saranno utilizzate per memorizzare più elementi in una singola variabile:

```
    esempiotuple = ("Mario", "Maria", "Pino")
print(esempiotuple)
```

Gli elementi *tuple* sono ordinati, non modificabili, consentono valori duplicati e sono indicizzati, quindi il primo elemento avrà *index[0],* il secondo *index[1]* e così via.

Quando diciamo che le tuple sono ordinate, significa che gli elementi hanno un ordine definito e quindi immutabile. Inoltre non sono modificabili, il che significa che non possiamo cambiare, aggiungere o rimuovere elementi dopo che la tupla è stata creata.

Tuttavia, poiché le tuple sono indicizzate, possono avere elementi con lo stesso valore e potremo determinare quanti elementi possiede attraverso la funzione *len()*:

```
esempio = ("Mario", "Mario", "Maria", "Maria", "Pino")
print(len(esempio))
```

Per creare una tupla con un solo elemento, dovremmo aggiungere una virgola dopo di esso, pena la restituzione di un errore:

```
esempio = ("Mario",)
```

Gli elementi della tupla possono essere e possono contenere qualsiasi tipo di dati:

```
uno = ("Mario", "Maria", "Pino")
due = (1, 2, 3, 4, 5)
tre = (True, False)
quattro= ("Mario", 10, True, 20)
```

Dal punto di vista di Python, le tuple sono definite come oggetti con il tipo di dati *'tuple'*: <class 'tuple'>

```
esempio = ("Mario", "Maria", "Pino")
print(type(esempio))
```

È anche possibile utilizzare il costruttore *tuple()* per crearne una:

```
esempio = tuple(("Mario", "Maria", "Pino"))
print(esempio)
```

La gestione delle eccezioni

Come sappiamo già, Il blocco *try* consentirà di verificare la presenza di errori in una porzione di codice. Con Python avremo anche il blocco *except* che ci consentirà di gestire l'errore. Il blocco *else* servirà per eseguire codice quando non ci sono errori, mentre *finally*, visto in precedenza con Java, eseguirà il codice, indipendentemente dal risultato dei blocchi *try* ed *else*.
Vediamo un esempio, dove try genererà un'eccezione, poiché *a* non è stato definito prima:

```
try:
print(a)
except:
print("Abbiamo un problema")
```

Poiché il blocco *try* genera un errore, sarà eseguito il blocco di eccezione.
Senza il blocco *try*, il programma si arresterà in modo anomalo e genererà un errore:

```
print(a)
```

Potremo definire diversi blocchi di *eccezione* per lo stesso errore, ad esempio, se volessimo eseguire un blocco di codice specificando ed escludendo un tipo di errore:

```
    try:
print(a)
    except NameError:
print("La variabile a non è definita")
    except:
print("Abbiamo un problema")
```

In questo esempio, *try* non genera alcun errore:

```
    try:
print("Ciao da Mario")
    except:
print("Errore")
    else:
print("Tutto liscio")
```

Esempio di *finally*:

```
    try:
print(a)
    except:
print("Errore")
    finally:
print("Esempio terminato")
```

Proviamo ad aprire e poi scrivere su un file protetto:

```
    try:
a = open("prova.txt")
try:
a.write("Lorum Ipsum")
except:
print("Errore, file protetto?")
finally:
a.close()
```

```
except:
print("Errore nell’apertura del file")
```

Infine, con l’utilizzo della parola chiave *raise* potremo scegliere di generare un'eccezione se si verifica una condizione particolare:

```
   a = 0

   if a < 1:
raise Exception("Errore, a non può essere maggiore di 0")
```

La parola chiave *raise* è utilizzata per sollevare un'eccezione. Sarà quindi possibile definire il tipo di errore da generare e il testo da stampare per l'utente:

```
   a = "Mario"

   if not type(a) is int:
raise TypeError("Cerchiamo numeri, non stringhe")
```

La gestione dei file

Esattamente come per i suoi fratelli C#, Java, e PHP, anche Python avrà accesso a tutte le operazioni inerenti i file. Per aprirne uno, utilizzeremo la sintassi di base, cioè la funzione *open()*: *nome del file oggetto = open*("*nome del file di testo*", *access_mode, buffering*). Gli ultimi due attributi sono facoltativi. Vediamo subito degli esempi più concreti:

```
   open("directory/prova.txt")
esempio = open("prova.txt")

   esempio = open("dir/prova.txt", "a")
print(esempio.mode) # Output: "a"

   esempio = open("dir/prova.txt")
print(esempio.read()) # Output: (contenuto del file prova.txt)
```

Come appena visto dagli esempi, ci sono diverse modalità di accesso per le operazioni sui file. Vediamole di seguito:

- **a** Apre un file a cui potremo aggiungere una stringa. Se non esiste, ne sarà creato uno.
- **a+** Apre un file sia per l'aggiunta sia per la lettura. Se non esiste, ne sarà creato uno.
- **ab** Apre un file binario per l'aggiunta. Se non esiste, sarà creato.
- **ab+** Apre un file sia per l'aggiunta sia per la lettura in modalità binaria. Se il file non esiste, ne sarà creato uno.
- **b** Apre un file in modalità binaria.
- **r** Apre un file in modalità di sola lettura. Questa è la modalità di accesso predefinita in Python.
- **r+** Apre un file in modalità lettura/scrittura.
- **rb** Apre un file di sola lettura in modalità binaria.
- **rb+** Apre un file in lettura/scrittura in modalità binaria.
- **t** Apre un file in modalità testo.
- **w** Apre un file in modalità di sola scrittura; sovrascrive il file se esiste già.
- **w+** Apre un file in modalità lettura/scrittura; sovrascrive il file se esiste già.
- **wb** Apre un file in modalità binaria di sola scrittura; sovrascrive il file se esiste già.
- **wb+** Apre un file in modalità lettura/scrittura binaria; sovrascrive il file se esiste già.

Non dimentichiamoci che quando avremo finito le nostre operazioni sui file questi andranno "chiusi" attraverso il metodo *close()*.
Infine potremo decidere se utilizzare o no il *buffering* nel contesto delle operazioni sui file. Questo meccanismo riguarda una porzione di un file ed essenzialmente si riferisce al processo della sua memorizzazione in un'area di memoria temporanea fino a quando non sarà caricato completamente. Il valore zero disattiverà il buffering, mentre uno lo abiliterà. Ad esempio: *nome = open("prova.txt", "r", 1)* . Se non è fornito alcun valore, Python utilizzerà l'impostazione predefinita del sistema che non prevede l'utilizzo del *buffer*. Solitamente è una buona idea mantenerlo attivo per aumentare la velocità nelle operazioni sui file.

In Python abbiamo anche quattro attributi principali che ci serviranno per lavorare con i file. Di seguito l'elenco e la descrizione:

- **.closed** Restituisce "true" se il file è chiuso, cioè se non si può leggere o scriverci sopra.
- **.mode** Restituisce la modalità di accesso al file: print(esempio.mode)
- **.name** Restituisce un filename: print(esempio.name)
- **.softspace** Restituisce "*false*" se le istruzioni print devono avere uno spazio inserito prima del primo elemento. E' obsoleto da Python 3.0

Come abbiamo segnalato nella descrizione in Python un file deve rimanere aperto per essere manipolato. Un file impostato su *closed* non può essere scritto o non se ne possono esaminare gli attributi. Vediamo come funziona con un semplice esempio, nel quale utilizzeremo il modulo *time* per attivare delle pause tra le varie operazioni:

```
    import time
    file = open("mario.txt", "wb")
print("Il file si chiama: ", file.name)
print("Modalità di apertura: ", file.mode)
    print("Il file è chiuso?", file.closed)
    time.sleep(1)
    file.close()
    print("Ora abbiamo chiuso il file..");
    time.sleep(1.5)
print("File chiuso adesso?", file.closed)
```

Operazioni sulle directory

Una directory o una cartella sono "pezzi" fondamentali da gestire in un qualsiasi sistema operativo. Ovviamente anche Python offre modi per lavorare con loro. Diamo quindi un'occhiata al prossimo listato, inizieremo importando il modulo *os* proprio per le operazioni sulle directory:

```
    import os
print("Cartella corrente:", os.getcwd())
print("Elenco file:", os.listdir())
os.chdir('C:\\')
print("Nuovo percorso della directory:" , os.getcwd())
print("Creiamo una nuova directory e chiamiamola Mario")
```

```
os.mkdir('Mario')
print("Elenco dei file in Mario:", os.listdir('Mario'))
```

Abbiamo visto che con *getcwd()* visualizzeremo su schermo la directory, mentre con *listdir()* elencheremo i file al suo interno e con *mkdir()* saremo in grado di crearne una nuova.
Per quanto riguarda la ridenominazione di una directory, useremo *os.rename("Nome", "Altronome").* Una volta che una directory non è più necessaria ed è prima vuota di file, con l'istruzione *os.rmdir("Dirprova")* la rimuoveremo.
Se invece dovessimo localizzare i file che corrispondono a schemi di denominazione specifici, potremo utilizzare la corrispondenza al modello dei nomi dei file, vediamolo nel listato:

```
   import os
import fnmatch

   # Cerchiamo i file con estensione .jpg/.pdf
for filename in os.listdir('C:\\'):
if filename.endswith('.jpg') or filename.endswith('.pdf'):
print(filename)

   # Cerchiamo nella dir Mario i file con estensione .exe
for filename in os.listdir('/mario'):
if filename.endswith('.exe'):
print(filename)

   # Cerchiamo in Windows/doc i file .pdf che iniziano con 'm'
for filename in in os.listdir('/Windows/doc'):
if fnmatch.fnmatch(filename, 'm*.pdf'):
print(filename)
```

Qui abbiamo introdotto due utili funzioni per localizzare i file con convenzioni di denominazione specifiche, cioè *endswith()* e *fnmatch().* Quest'ultima implementa le cosiddette ricerche basate su caratteri *jolly* (ad esempio *.pdf per qualunque file in formato PDF) nei nostri programmi.
Per effettuare delle ricerche più complesse potremo adoperare l'estensione *glob* e la relativa istruzione (sempre *glob*). Il termine *globbing* si riferisce

all'esecuzione di ricerche di file molto specifiche all'interno di una directory indicata. Il termine *glob*, cioè nient'altro che l'abbreviazione di *global* (globale), ha le sue origini nel mondo dei sistemi operativi basati su Unix. Vediamo nel prossimo esempio un caso studio su come sfruttare questa tecnica:

```
   import pandas as pd
import glob

   # Lasciamo all'utente la scelta della cartella da esaminare
path = 'input'
files = glob.glob(path + '/*.csv')

   # Creiamo una lista vuota per inserirci i dati dei file excel
lista = []

   # Usiamo un ciclo per leggere e memorizzare i dati nella lista
for a in files:
lista_temp = pd.read_csv(a)



   # Concateniamo i dati per poterli visualizzare
lista.append(lista_temp)
print(a'Lista di {a} creata come {lista_temp.shape}')

   # Uniamo tutti i dati
dataframe = pd.concat(lista, axis=0)
print(dataframe.shape)
dataframe.head()
```

Si parte con il modulo *Pandas* (https://dataindependent.com/pandas/) utile per l'analisi dei dati, che sfrutteremo nel listato con la sintassi *pd*. I file che ci interessano sono di Excel, quindi, ovviamente dovremo prima accertarci che una cartella con questo tipo di file sia presente prima dell'esecuzione del nostro esempio, dove essenzialmente analizzeremo tutti i file *.csv presenti nella directory per poi estrapolarne i dati e inserirli in un *data frame* (un insieme di dati) che abbiamo chiamato *lista_temp* e infine visualizzarli su schermo.

La gestione del calendario

Abbiamo trattato dello stesso argomento nel paragrafo dedicato a Java e sicuramente noteremo la maggior semplicità nella gestione delle medesime funzionalità le quali dovranno essere adeguatamente importate dal modulo datetime. Osserviamo un esempio esplicativo:

```
    import datetime
from datetime import timedelta

    data = datetime.datetime.now()
print("L'ora è:", data)
print("Giorno e mese:", data.strftime("%A in %B"))
print("Ventidue anni fa era", data.year-22)

    nelfuturo = datetime.timedelta(days=10)
nelfuturo += data
print("Tra dieci giorni sarà", nelfuturo.strftime("%B"))
```

Come appena visto potremo visualizzare semplicemente l'ora o la data, e poi modificarla con le istruzioni appropriate, come abbiamo fatto con Java precedentemente.
Adesso vedremo i marcatori di formattazione disponibili in Python per tutte le nostre esigenze relative alla gestione del tempo:

- *%A* riporta il giorno
- %a giorno abbreviato
- *%B* riporta il mese
- *%b* mese abbreviato
- *%H* riporta l'ora suddivisa in 24h
- *%I* riporta l'ora suddivisa in 12h
- *%p* AM o PM
- *%Z* Fuso orario

Le espressioni regolari

Siamo giunti nel paragrafo solitamente definito come pesante, complesso e noioso. Non è così! Lo vedremo. Cercheremo prima di riassumere l'argomento che ci introdurrà in un mondo fantastico. Questo perché ci faciliterà tantissimo il lavoro. Andiamo con ordine, le espressioni regolari,

abbreviate in *RegExp*, sono un potente strumento e una necessità dello sviluppo di qualunque software. Di contro la grammatica e la sintassi appaiono molto complesse, in sostanza una lingua a parte che richiede parecchio tempo per essere padroneggiata.
Tuttavia, con l'uso appropriato delle espressioni regolari, qualcosa che normalmente richiederebbe una schermata di codice potrà essere riassunto in una singola affermazione. In realtà non è tutto qui, questa condensazione di una porzione del programma è solo la punta dell'iceberg.
Ovviamente non ci metteremo a elencare tutto ciò che saremo in grado di fare, ma dobbiamo anche renderci conto della potenza di questo strumento. Perciò, essenzialmente, le espressioni regolari ci serviranno anche per, ad esempio: trovare e modificare parti di una più ampia area di testo, velocizzare la creazione di codice votato all'analisi di stringhe, determinare se un elemento ha un nome di classe specifico, validare un nome, un'email o una password in un form, manipolare il HTML, e molto altro ancora.

Sintetizzando in maniera estrema, storicamente tutto nasce intorno agli anni 50, quando un matematico Statunitense di nome Stephen Kleene con i suoi studi ci regala i "linguaggi formali", che oggi ci permettono di comunicare con le macchine, nello specifico con i computer. Di fatto senza le espressioni regolari probabilmente la stessa internet avrebbe avuto enormi difficoltà a esistere.

Che cosa sono le RegExp? Cercando sempre la sintesi potremmo affermare che si tratta di uno strumento atto al controllo sintattico di una data stringa per verificare che essa coincida ad un *pattern* (o modello) definito. Il pattern sarebbe il motivo della ricerca generale, ed è composto di caratteri semplici, speciali o entrambi.

Vediamo un banale esempio di utilizzo di espressioni regolari in Python:

```
import re
testo = "Mario ama il calcio"
regex = re.findall("io", testo)
print("Ho trovato", len(regex), "corrispondenze di IO in", testo)
```

In questo esempio iniziamo importando il modulo *re*, utile per utilizzare il metodo *findall,* a sua volta adoperato per cercare le istanze di "io" nella variabile stringa *testo*. Abbiamo anche usato il metodo *len()* per contare le istanze memorizzate nella lista *regex*.
Vediamo un altro esempio, più lungo e con qualche metodo nuovo:

```
    import re
    ric1 = re.search('Mario', 'Mario ama il calcio')
    if ric1:
       print(ric1)
    stringa = "Mi chiamo Mario"

    ric2 = re.search('Mario', stringa)
    if ric2:
       print(ric2)

    ric3 = re.fullmatch('Mario', stringa)
    if ric3:
       print("Corrispondenza per 'Mario'!")
# Questo messaggio non verrà mai visualizzato

    else:
       print("Nessuna corrispondenza per 'Mario'")

    ric3 = re.fullmatch('Mi chiamo Mario', stringa)
    if ric3:
       print(ric3)
    ric4 = re.match('ama', 'Mario ama il calcio')
    if ric4:
       print(ric4)
else:
       print("Nessuna corrispondenza per 'ama'")

    ric5 = re.match('Mario', 'Mario ama il calcio')
    if ric5:
       print(ric5)
```

Prima di tutto una breve annotazione, nella terza riga abbiamo utilizzato *if ric1:* al posto di *if ric1 == True:* come comoda abbreviazione, prendetene nota, non ne farete più a meno.

Andiamo avanti, abbiamo usato *search()*, *match()* e *fullmatch()*. Quali sono le differenze tra di loro? Il metodo di ricerca *search* passa attraverso un'intera stringa per un determinato modello, mentre *fullmatch* restituisce

true solo se una stringa rispecchia completamente un modello. Infine il metodo *match* lo cercherà solo all'inizio di una stringa.

Metacaratteri

Le espressioni regolari sono sfruttate appieno attraverso i cosiddetti *metacaratteri*, che, fondamentalmente sono gli elementi costitutivi per ricerche più avanzate concernenti le stringhe. Vediamo un elenco di alcuni importanti metacaratteri in Python:

\w Qualsiasi parola. Di solito si riferisce a caratteri alfanumerici
\S Spazio bianco
\W Qualsiasi lettera
\S Non spazi bianchi
\D Qualsiasi cifra
. Qualsiasi carattere singolo
\D Qualsiasi cosa, non cifre
* Zero o più caratteri
\. Punto
$ Abbina fine riga
? Zero o un carattere
{ n } Si verifica per n volte
+ Uno o più caratteri
[az] Set di caratteri
^ Abbina inizio riga
[0-9] Set di caratteri numerici

Dopo aver visto questo elenco andiamo a provare alcuni metacatteri assieme alle RegExp in un semplice listato:

```
   import re
   ric1 = re.search('.....', 'Ciao da Mario!')
   if match1:
      print(ric1)

   ric2 = re.search('\d..', 'Mario555')
if ric2:
      print(ric2) # Visualizza 555

   ric3 = re.search('\D*', 'Mi chiamo Mario555.')
```

```
    if ric3:
        print(ric3) # Visualizza "Mi chiamo Mario"

    ric4 = re.search('y *\w*', 'Ciao. Come stai?')
    if ric4:
print(ric4) # Visualizza "stai"

    ric5 = re.search('\S+', 'Ciao. Che succede?')
if ric5:
print(ric5) # Visualizza "Ciao".
```

Nell'esempio appena visto con *ric4*, stiamo usando il metacarattere \w che si riferisce alla ricerca di corrispondenze con qualsiasi parola intera. Senza questo importante simbolo al suo posto, otterremmo un output di "s" invece di "stai".
Approfondiamo ancora l'argomento con un ulteriore esempio. Da notare come finora, anche grazie a questi banali listati, le espressioni regolari sembrino meno ostiche, forse addirittura utili e sicuramente non riuscirete più a farne a meno.

```
    import re

    stringa = 'Miiaoo Baaau-Baau Purrpurrpur-Pur-Pur'
ricerche = [r'Mi*', # M e zero o più i (*)
r'Mi+', # M e una o più i (+)
r'Ba?', # B e zero o una a (?)
r'ur{2}', # u e due r
r'[MBP]\w*', # Cerchiamo frasi con iniziali: M, B o P
r'^Pu\w*', # Cerchiamo una frase che inizia con "Be"
r'...$' # Cerchiamo gli ultimi 3 caratteri
]

    def risultati(ricerche, stringa):
for pattern in ricerche:
ricerche1 = re.compile(pattern) # Evochiamo compile()
print('Cerco {} in'.format(pattern), stringa)
print(re.findall(ricerche1, stringa)) # e findall()
```

```
    risultati(ricerche, stringa) # Eseguiamo il metodo risultati
```

I modelli di struttura del listato contengono diverse ricerche, mentre *stringa* memorizza tutto ciò che ne risulterà. Queste due strutture di dati devono essere inserite nel metodo *risultati* che abbiamo creato in seguito.
All'interno di questo nostro nuovo metodo, utilizziamo due delle funzioni RegEx di Python: *compile()* e *findall()*. Con il primo, stiamo convertendo i modelli RegEx in modelli oggetto, che sono quindi utilizzati per la corrispondenza delle ricerche. Quest'approccio è più efficiente negli scenari in cui i modelli di ricerca devono essere riutilizzati, ad esempio in un accesso a un database di prodotti.
Findall è utilizzato per scoprire tutte le occorrenze in un modello di ricerca all'interno delle stringhe. La *r'* nel nostro elenco fa sapere a Python che una stringa è considerata una "*stringa grezza*" o letterale, il che significa che i simboli in essa contenuti saranno interpretati senza funzioni speciali. Ad esempio, *\n* non indicherà una nuova riga in una stringa grezza.
Di seguito approfondiremo alcune funzionalità avanzate delle RegEx. In pratica adopereremo un nuovo approccio con *search()* e introdurremo l'utilizzo di due nuovi metodi: *group()* e *sub()*:

```
    import re
frase = "Il calciatore Totti è il preferito di Mario"
scelta = re.search(r"calciatore.*(Totti|Messi|Baggio)", frase)
if scelta:
print("Il migliore è", scelta.group(), "!")

    altro = "Che giornate stupende con loro"
altro = re.sub('giornate', 'partite', altro)
print(altro) # Che partite stupende con loro

    ultimo = 'meglio andare al mare'
ultimo = re.sub('([a-e])s*', 'Y', ultimo)
print(ultimo) # meglio aYYYYe al mYYe
```

In questo caso il metodo di ricerca è utilizzato per confrontare e individuare le stringhe che ora sono elencate come argomenti del metodo. In altre parole, *frase* è ricercato per un totale di tre differenti calciatori. Questi sono separati dall'operatore logico o, indicato dal carattere della linea

verticale (ad esempio, |). Il metodo è fatto per cercare qualcuno di questi, ma solo dovrebbero apparire accanto alla stringa "*calciatore*". Se esiste una tale corrispondenza tra *frase* e un elemento in *scelta*, il programma la visualizza.
A seguire abbiamo sostituito prima una frase e poi alcune lettere incluse in pattern definito.

Multithreading?

Abbiamo già parlato del multithreading nel capitolo precedente. Come Java e C#, Python è in grado di elaborare più thread di esecuzione. Tuttavia, ci sono alcune differenze sostanziali.
Il multithreading di Python in realtà non lavora i thread in modo parallelo, piuttosto, esegue questi thread in un modo che potremo definire "concorrente". Ciò deriva dall'implementazione in Python del cosiddetto blocco dell'interprete globale (GIL). Questo meccanismo serve per sincronizzare i thread e assicurarsi che l'intero progetto sia eseguito solo su una singola CPU; quindi non sono utilizzati i processori multicore attuali. Questo per varie ragioni, soprattutto di sicurezza e stabilità. Tuttavia non siamo obbligati a utilizzarlo, infatti, potremo sempre disabilitare il GIL.
Tornando al discorso precedente, sebbene concorrenza e parallelismo siano termini correlati, non sono la stessa cosa. Il primo si riferisce all'approccio dell'esecuzione simultanea di attività separate, mentre con il secondo, un'attività è suddivisa in altrettante secondarie che sono quindi eseguite contemporaneamente.
L'effettiva elaborazione parallela dei thread in Python si ottiene attraverso l'uso di più processi, tutti con il proprio interprete e GIL. Questo è noto come *multielaborazione* o *multiprocessing*.
L'interpretazione dei creatori di Python su questi concetti può essere alquanto complicata (rispetto, ad esempio, a Java). Ora, un processo in Python non è lo stesso di un thread. Sebbene entrambe siano sequenze indipendenti di esecuzione del codice, esistono diverse differenze. Un processo tende a utilizzare più memoria di sistema di un thread. Il ciclo di vita di un processo è generalmente anche più lento da gestire; la sua creazione e cancellazione richiede più risorse. Andiamo a vedere le principali differenze tra thread e processi in Python:

- *Thread*: possono usufruire del GIL, non supportano le CPU multi core, hanno una complessità del codice più alta, sfruttano meno RAM, non

possono essere bloccati, il loro utilizzo ideale è per applicazioni di rete e interfacce utente.

- *Processi*: non possono usufruire del GIL, supportano le CPU multi core, hanno una complessità del codice più bassa, sfruttano più RAM, possono essere bloccati, il loro utilizzo ideale è per rendering, mining, tutte quelle attività in cui la CPU è spremuta.

Python dispone di ben tre moduli di codice per l'elaborazione simultanea: *multiprocessing*, *asyncio* e *threading*. Perciò, ad esempio, per compiti in cui è richiesta un'attività intensa della CPU, andremo a utilizzare il modulo *multiprocessing*, mentre, per creare un'applicazione di chat sarà meglio affidarsi al modulo *threading,* e così via.

Multithreading!

Il multithreading in Python va di pari passo con il meccanismo di blocco dell'interprete globale (GIL) come discusso in precedenza in questo capitolo. Quest'approccio utilizza il *principio della mutua esclusione*, cioè essenzialmente evita il sovrapporsi di processi che accedono alle variabili comuni. Poiché la pianificazione nel multithreading di Python è eseguita dal sistema operativo, è inevitabile un certo sovraccarico (cioè un ritardo); questo è qualcosa di cui il multiprocessing generalmente non soffre.
Esaminiamo come viene implementato il multithreading in un esempio dove utilizzeremo tre thread:

```
   import threading
def num_moltplicati(n1, n2):
print("Moltiplica", n1, "con", n2, "=", (n1 * n2))
def num_divisi(n1, n2):
print("Dividi", n1, "con", n2, "=", (n1 / n2))
if __name__ == "__main__":
# Crea due thread
thread1   =   threading.Thread(target=num_moltiplicati,   args=(5,6))
thread2 = threading.Thread(target=num_divisi, args=(5,6))
# Avvia i thread..
thread1.start()
thread1.join() # Assicuriamoci che il thread1 sia eseguito
```

```
thread2.start() # Prima di iniziare il thread 2
thread2.join() print("Tutto apposto!")
```

In questo listato creiamo due funzioni, vale a dire, *num_moltiplicati* e *num_divisi*. Questi prendono due argomenti ciascuno. Con il primo moltiplichiamo *n1* per *n2*, mentre il secondo divide *n1* per *n2*. I risultati sono quindi semplicemente riportati sullo schermo.
C'è una funzione sulla quale dovremo prestare molta attenzione. Il metodo *join()* di Python serve per assicurarsi che un thread abbia completato la sua elaborazione prima di proseguire con il codice nel listato. Se dovessimo rimuovere la riga *thread1*. *join()*, l'output risultante sarebbe solo un gran casino.
Per restare in tema, una rara, ma non impossibile, condizione nel contesto dell'elaborazione simultanea, che riporta a uno scenario in cui due o più processi stanno modificando una risorsa, ad esempio un file, contemporaneamente, e Il risultato finale dipende da quale processo arriva per primo è classificata come non auspicabile. In pratica è meglio non impostare uno scenario del genere per evitare problemi.
Sebbene l'approccio del multiprocessing produca in generale codice altamente efficiente per la CPU, possono comunque verificarsi problemi minori, di sovraccarico (overhead, ritardo). Se un progetto multiprocessing in Python presenta questi problemi, in genere occorrono durante l'inizializzazione o la terminazione di un processo.
Bando alle ciance! vediamo subito un listato che illustra l'utilizzo del multiprocessing:

```
   import time
import multiprocessing

   def contatore():
name = multiprocessing.processo().name
print (name, "eccolo")



   for i in range(3):
time.sleep(1)
print (name, i+1,"/ 3")
```

```
    if __name__ == '__main__':
# Definiamo i processi da eseguire
a = multiprocessing.Process(name='Cont. 1', target=contatore)
b = multiprocessing.Process(name='Cont. 2', target=contatore)
c = multiprocessing.Process(target=contatore)
d = multiprocessing.Process(name='Cont. 3', target=contatore)

    a.start()
b.start()
c.start() # Questo contatore senza nome non funzionerà
d.start()
```

Ogni processo in Python ha il nome di una variabile, come abbiamo appena visto. Se non ne è definita nessuna, sarà automaticamente visualizzata un'etichetta generica con un numero di processo. In pratica non porterà ad alcun risultato utile.

Multitasking cooperativo e asyncio

Ci sono tre tipi importanti di meccanismi che elaborano dati in Python facili da confondere: *iterabili, generatori* e *coroutine*. Le liste (ad esempio, *listaGenerica[10,13,50]*) sono dei semplici dati iterabili. Possono essere lette tutte le volte che sono necessarie senza intoppi. Tutti i valori all'interno degli elenchi saranno conservati fino a quando non sono specificati per l'eliminazione.

I generatori sono iteratori a cui è possibile accedere solo una volta poiché non memorizzano i loro contenuti in memoria. Sono creati come funzioni comuni poiché hanno la parola chiave *yield* al posto di *return*. Ci serviranno quando dovremo leggere gli attributi di file di grandi dimensioni, come il conteggio delle righe. I generatori utilizzati a questo scopo produrranno dati non aggiornati e quindi non necessari, evitando errori di memoria e facendo funzionare i programmi Python in modo più efficiente.

Le Coroutine sono un tipo unico di funzione che può cedere il controllo a una funzione chiamante senza terminare il loro contesto nel processo. Mantengono il loro stato d'inattività senza intoppi in background. Essenzialmente sono utilizzate per il multitasking cooperativo.

Andiamo adesso a parlare di *asyncio*. Abbreviazione di input/output asincrono, è l'ultimo modulo di Python progettato per scrivere codice in esecuzione simultanea che studieremo. Nonostante abbia somiglianze sia

con il threading sia con il multiprocessing, in realtà rappresenta un approccio diverso, denominato *multitasking cooperativo*. Il modulo *asyncio* offre una sorta di pseudo-concorrenza utilizzando un unico processo in esecuzione in un singolo thread.
E cosa significa lavorare con codice asincrono in Python? Bene, il codice scritto con quest'approccio può essere messo in pausa in modo sicuro affinché altri frammenti di codice all'interno di un progetto eseguano le loro attività. Un processo asincrono offre i suoi tempi d'inattività agli altri processi per eseguire il loro corso; è così che si può effettivamente ottenere un tipo di concorrenza con il modulo asyncio.
Il componente principale di un progetto asincrono è il ciclo degli eventi; eseguiremo i nostri sottoprocessi da questo costrutto. Le attività sono pianificate in un ciclo di eventi e gestite da un thread. Fondamentalmente, il ciclo degli eventi è lì per coordinare le attività e assicurarsi che tutto funzioni senza intoppi quando lavoriamo con il modulo *asyncio*. Vediamo come implementare effettivamente attraverso un listato di coroutine concatenate asincrone e l'uso di un ciclo di eventi:

```
   import asyncio
async def controllo(x):
await asyncio.sleep(1.0)
testo = "%d non è un numero primo" % x
if x>1:
for i in range(2, x):
if (x % i) != 0:
testo = "%d è un numero primo!" % x
break
return testo

   async def risultati(y):
ris = await controllo(y)
print(ris)

   iterazione = asyncio.get_event_loop()
iterazione.run_until_complete(risultati(1))

   iterazione.run_until_complete(risultati(5))
```

```
    iterazione.run_until_complete(risultati(17))
iterazione.close()
```

Lo scopo, in questo caso, è quello di cercare dei numeri primi (cioè numeri che possono essere solo divisi per uno e per se stessi, quindi due, cinque, ma non quattro, in quanto è divisibile per uno, per se stesso ma anche per due). Dopo definiamo due funzioni/coroutine asincrone, *controllo(x)* e *risultati(y),* entrambe con le parole chiave *async def* poiché questa è la sintassi che questo modulo richiede.
Ora, la prima funzione che abbiamo definito inizia con la riga *await asyncio.sleep(1.0)*. Rispetto al normale metodo di sospensione in Python, questa varietà asincrona non blocca il programma; invece, fornisce un periodo di tempo specificato per altri compiti da completare in background.
La riga *ris = await controllo(y)* nella seconda coroutine è lì per assicurarsi che la prima funzione abbia completato la sua esecuzione. Attendere è davvero una parola chiave in qualche modo autoesplicativa e cruciale nell'approccio asincrono.
Per far funzionare completamente il nostro elenco asincrono, dobbiamo lavorare con il ciclo degli eventi. Lo facciamo creando un nuovo oggetto iterabile, chiamato senza troppa fantasia *iterazion*e, ed evocando una funzione chiamata *asyncio.get_event_loop()*. Sebbene esistano diverse varietà di funzioni per la gestione del loop di eventi, per motivi di chiarezza, in questo esempio tratteremo solo l'approccio sopra menzionato.
Successivamente, la funzione *run_until_complete* esegue effettivamente la nostra funzione *risultati(y)* fino al completamento delle rispettive attività. Infine, termineremo il nostro ciclo di eventi evocando il metodo *close*.

Funzioni anonime o lambda

Questo strumento ci servirà nei casi in cui il nostro listato sarà improntato verso la programmazione funzionale piuttosto che lo stile basato sulle classi. Rispetto alle normali funzioni la principale differenza è che per dichiarare una funzione anonima non c'è bisogno di utilizzare *def* come da normale prassi. Pensiamo a una funzione lambda come a una funzione usa e getta, essendo anonima non avrà il nome e solitamente è molto compatta. La sua sintassi è la seguente:

```
lambda (argomenti) : (espressioni)
```

```
    addizione = lambda x, y : x + y
```

Ma vediamole in opera in un listato leggermente più complesso:

```
    def prodotto(x):
return lambda a : a * x

    uno = prodotto(8)
due = prodotto(5)
tre = prodotto(6)
qua = prodotto(9)
print("Prima operazione:", uno(3))
print("Seconda:", due(5))
print("Terza:", tre(6))
print("Ultima:", qua(4))
```

In realtà le funzioni lambda sono state pensate anche per non essere definite attraverso def, infatti questo tipo di funzioni non hanno un nome per poterle richiamare, tuttavia potremo assegnarli una variabile:

```
    variabile = lambda x: x**2
variabile(3)
```

Python ci permette di associare alle funzioni dei metodi aggiuntivi: *filter(),* il quale prende una lista e ne crea una nuova composta da tutti gli elementi che restituiscono *true*, mentre *map()* crea un nuovo elenco di iterabili che elabora, senza un ciclo, infine *reduce(),* che applica una funzione a una lista (o altri iterabili) e restituisce un singolo valore, anche in questo caso a volte liberandoci del tutto dall'uso delle iterazioni (loop). Vediamo quindi un listato con una funzione lambda applicata a un elenco filtrato:

```
    from functools import reduce

    eta = [18, 32, 30, 22, 25]
print("Età dei giocatori della squadra di calcetto:", eta)

    eta = list(filter(lambda anni: anni>=30 and anni<40, eta))
print("Calciatori in età avanzata:", eta)
```

```
    anziani = list(map(lambda x: x * 3, eta))
print("Probabilmente vivranno fino a:", anziani)

    tot = reduce(lambda x, y: x + y, eta)
print("La somma degli anni dei due più anziani:", tot, "anni")
```

Partiamo importando il modulo necessario (*functools*) per servirci di *reduce(),* poi creeremo un elenco con sette valori concernenti l'età di alcuni calciatori. A noi interessano quelli dai trenta anni in su, quindi proveremo a filtrarli per poi compiere alcune operazioni con *map* e *reduce*.

Metodo Zip e archivio Zip

Adesso ci occuperemo di questo metodo che non ha nulla a che vedere con il conosciutissimo programma utilizzato per comprimere qualunque file al fine di accorparli e ridurne le dimensioni. In effetti, c'è anche questa possibilità in Python, lo vedremo più avanti. Perciò zippare, in questo caso, prende il nome dalla cerniera, quella di una giacca o dei jeans. Mentre in Python si riferisce a un metodo di combinazione degli elementi con due o più iterabili. Ad esempio, zippare è un ottimo strumento quando si creano strutture di dati di un dizionario.

Ora, una funzione zip prende due iterabili (ad es. liste) e restituisce un insieme di *tuple*:

```
    lettere = ['m', 'a', 'r', 'i', 'o']
numeri = [1, 2, 3, 4, 5]

    zip1 = zip(lettere, numeri)
    for i in zip1:
        print(i)

    lettere2 = ['M', 'A', 'R', 'I', 'O']
    zip2 = zip(lettere2, numeri, lettere)

    for i in zip2:
        print(i)
```

All'inizio definiamo due liste, poi creiamo un oggetto zip usufruendo delle liste appena create. A questo punto andremo a utilizzare il ciclo *for* per

ottenere le *tuple*.

C'è una serie di scenari in cui avremo dei problemi con lo *zip* di base. Uno di questi riguarda insiemi *zip* con *tuple* incomplete. Proprio per questi casi esiste un metodo chiamato *zip_longest*. E' parte integrante nel modulo di codice *itertools*, e, nel nostro caso, riempirà tutti gli elementi mancanti con i dati segnaposto di nostra scelta. Vediamone una dimostrazione che ci aiuterà a capire:

```
    from itertools import zip_longest

    lettere = ['x', 'y', 'z']
numeri = [6, 7, 8, 9]
mario = range(5)
zippato = zip_longest(numeri, lettere, mario, fillvalue='blank')
for i in zippato:
print(i)
```

Noteremo che alla quarta riga andremo a definire la lunghezza dello zip per poi salvarlo in *mario*. Subito dopo andremo a evocare *zip_longest(),* e infine otterremo la visualizzazione del contenuto con il ciclo *for*.
Alla fine scopriremo che l'archivio in Python può anche essere decompresso e lo osserviamo in questo esempio:

```
    lettere = ['A', 'B', 'C', 'D']
numeri = [1, 2, 3, 4]
zippati = zip(lettere, numeri)
dati_zippati = list(zippati)
print("Ecco lo zip:", dati_zippati)

    a, b = zip(*dati_zippati)
print("Ecco i dati, decompressi:")
print('Lettere:', a)
print('Numeri:', b)
```

Iniziamo comprimendo i dati, nello specifico gli insiemi di lettere e numeri. Dopo andremo a decomprimere i dati servendoci dell'operatore asterisco.
Come accennato in precedenza in questo capitolo, zip è una parola con

almeno due significati popolari nel contesto dell'informatica. Abbiamo esplorato la compressione (e la decompressione) in Python, coprendo il primo significato. Ora è il momento di lavorare con l'altro ZIP, il popolare formato di file di archivio, da qualsiasi IDE Python.
Ora, le directory (e le loro sottodirectory) compresse con il software ZIP finiscono come un unico file che tende anche a essere di dimensioni considerevolmente più piccole rispetto al suo contenuto non compresso. Questa è ovviamente un'ottima soluzione per la distribuzione di file basata sulla rete. Naturalmente, sono disponibili diverse soluzioni di compressione dei file ma ancora oggi, in sostanza nel 2023, ZIP è ancora un formato di archivio di file straordinariamente onnipresente, disponibile per tutti i sistemi operativi più diffusi.
Anche a Python piacciono i suoi file ZIP. Possiamo effettivamente gestirli dall'interno di un suo IDE. C'è un modulo di codice chiamato, senza troppa fantasia, *zipfile*. Vediamo una dimostrazione per come creare un file ZIP e come visualizzarne il contenuto:

```
    import os
    from os.path import basename
    from zipfile import ZipFile # Include il modulo ZipFile

    ricerca = "foto"
    def creare_zip(dirName, zipFileName, filter):
        with ZipFile(zipFileName, 'w') as zip_ogg:
            for folderName, subfolders, filenames in os.walk(dirName):
for file in filenames:
if filter(file):
filePath = os.path.join(folderName, file)
zip_ogg.write(filePath, basename(filePath))

    creare_zip('..', 'mario.zip ', lambda name: lookfor in name)
print("Tutti i file contenenti <<", cerca, ">> nel loro nome sono
ora all'interno di mario.zip!")

    # Creiamo un oggetto e carichiamo mario.zip al suo interno
(usando l'attributo r per la lettura)
with ZipFile('mario.zip', 'r') as zip_ogg2:
```

```
    # Evochiamo namelist() e salviamo il contenuto nella variabile
"zip_ogg2"
contenuto = zip_ogg2.namelist()

    # Scorriamo zip_ogg2 e stampiamo su schermo ogni elemento
print("\nQuesti file sono stati archiviati in mario.zip:")
for i in contenuto:
print(i)
```

La prima cosa da considerare è che l'esempio sarà eseguito nella nostra directory di Python, teniamone conto. Detto questo, nel listato vedremo in gioco due principali meccanismi del supporto ZIP: creare un nuovo archivio di file (come, ad esempio, *mario.zip*) e recuperare i nomi dei file archiviati al suo interno. L'esempio mostra anche come utilizzare la funzionalità basata su caratteri jolly durante la raccolta di file in un archivio. In questo caso, cercheremo solo i file con la stringa "foto" in qualunque posizione nel nome del file. Perciò creeremo il metodo *creare_zip* e l'oggetto *ZipFile* per avviare l'iterazione utile a selezionare i file da comprimere nell'archivio (cioè quelli con la stringa *foto*) avendo cura di inserire l'attributo *w* per la sovrascrittura dei file uguali. Dopo averli aggiunti tutti con il metodo *write* avvieremo il metodo *creare_zip* per eseguire tutte le operazioni che vedremo sullo schermo alla fine dell'esecuzione.
Le istruzioni *with* usate in tandem con un file Zip esistono fondamentalmente per fornire chiarezza al nostro listato, inoltre ci libereranno dall'usare metodi di chiusura (ad es. *mario.close()*) di solito necessari dopo che le operazioni sul file sono state completate.

La libreria NumPy

NumPy (Numerical Python) è una libreria utilizzata per lavorare con gli array, ha anche funzioni per farlo con l'algebra lineare, matematica avanzata e le matrici.
NumPy è stato creato nel 2005 da Travis Oliphant. È un progetto open source e quindi può essere usato liberamente.
Sappiamo già che in Python sono presenti diverse possibilità per lavorare con gli insiemi (array), come le liste o le tuple, ma, di contro, sono lenti da elaborare. La libreria NumPy mira a fornire un oggetto array fino a 50 volte più veloce degli elenchi Python tradizionali.
L'oggetto array in NumPy è chiamato *ndarray*, fornisce molte funzioni di

supporto le quali lo rendono anche molto semplice da gestire. Questa rapidità di elaborazione è vitale nella scienza dei dati (Data Science), ossia quella parte dell'informatica dedita allo studio e all'analisi dei dati e delle informazioni che ne derivano.
Per usufruire di questa libreria dovremo installarla, anche se nelle ultime versioni di Python dovrebbe essere inclusa (assieme a PIP), in ogni caso il sito di riferimento è questo: https://scipy.org/
Qui abbiamo la sintassi di base:

```
    import numpy as np
insieme = np.array([5, 10, 15, 20])
print(insieme)
```

Gli array che creeremo con NumPy possono avere diversi livelli di profondità, ossia saremo in grado di gestire *array nidificati*, essenzialmente array che contengono come elementi altri array. Ad esempio il listato precedente mostra il classico insieme utilizzato il più delle volte ed è definito monodimensionale. Un array sarà bidimensionale, tridimensionale e così via in base al suo contenuto. Ogni nidificazione in più aggiungerà una dimensione. Vediamone uno 3D:

```
    import numpy as np
insieme = np.array([[[5, 10, 15], [6, 12, 18]], [[7, 14, 21], [8,
16, 24]]])
print(insieme)
print(insieme.ndim)
```

Le coppie d'insiemi sono definite matrici, quindi una matrice ha due dimensioni, due matrici tre dimensioni. Per conoscere con precisione questa informazione potremo avvalerci dell'istruzione *.ndim*.
Per accedere ai dati negli insiemi lavoreremo su dimensioni e indice, quindi utilizzeremo numeri interi separati da virgole:

```
    import numpy as np
insieme = np.array([[[5, 10, 15], [6, 12, 18]], [[7, 14, 21], [8,
16, 24]]])
print(insieme[0, 0, 1]) # Output: 10
```

In pratica con *print(insieme[0, 0, 1])* abbiamo indicato tre dimensioni diverse, la prima riguarda i due array iniziali, la seconda, il primo dei due, la terza, il secondo numero dell'insieme.
Per quanto riguarda il contenuto, potremo avvalerci della gran parte dei tipi di dati che usiamo solitamente. Con l'istruzione *dtype()* saremo in grado di sapere il datatype presente nel nostro array:

```
print(nomearray.dtype)
```

Mentre con le funzioni *concatenate()* e *stack()* potremo unire diversi insiemi, con *stack()* specificando l'*axis*(0 oppure 1) gestiremo il contenuto verticalmente oppure orizzontalmente:

```
import numpy as np

uno = np.array([1, 2, 3])
due = np.array([4, 5, 6])
tre = np.array([5, 10, 15])
qua = np.array([20, 25, 30])

array = np.stack((tre, qua), axis=1)
print(array)

array = np.concatenate((uno, due))
print(array)
```

Con l'istruzione *where()* otterremo l'indice del tipo di dato a cui siamo interessati:

```
import numpy as np
insieme = np.array([5, 10, 10, 15])
a = np.where(arr == 10)
print(a)
```

Le potenzialità di questa libreria abbinata a Python sono veramente enormi, ovviamente qui abbiamo visto solo una piccolissima parte di ciò che può offrire, mettendo il primo mattone di base per la branca della Data Science e tutte le altre implicazioni possibili grazie a NumPy e le sue implementazioni.

Quiz & esercizi

1) Se dovessimo aprire circa trecento file, per velocizzare le operazioni quale opzione dobbiamo attivare?

2) In Python, dopo aver aperto e lavorato su un file bisognerà eseguire un'operazione, quale?

3) Completare questo listato in modo che cerchi file: *.txt (servirà un ciclo iterativo per completare il lavoro):

```
import glob
print("\nCerchiamo file di testo...
    print(name)
```

4) Scrivere una RegEx con un pattern utile per cercare qualunque carattere singolo senza spazi in una stringa qualsiasi.

5) Correggere e completare il listato:

```
import re
testo = "Adoro quando mi massaggiano"
x = search(r"mi.*(menano|massaggiano|parlano)", testo)
if search:
print("", scelta.group(),"")
```

6) Quali sono i vantaggi del multithreading e del multiprocessing in Python?

7) E gli svantaggi degli ultimi due rispetto all'approccio asincrono?

8) Quando uso una funzione anonima dovrò necessariamente definire la funzione?

9) Completare il listato in modo che riporti il numero 24:

```
def y(x):
return lambda a : a ... x
uno = ...(.)
print("Risultato:", uno(.))
```

10) A cosa serve l'istruzione *axis* in un vettore tridimensionale?

Riassunto

In questo capitolo abbiamo parlato di come aprire, creare ed eliminare file, la gestione delle directory di base.

Poi un argomento spinoso: le espressioni regolari (RegEx), come e quando utilizzare *match(), search()* e *fullmatch()* in esse.

Abbiamo implementato il multithreading e il multiprocessing e visto quali sono le loro principali differenze. Abbiamo anche imparato cenni sul blocco globale dell'interprete (GIL).

Abbiamo visto le basi della programmazione asincrona utilizzando asincio, infine le funzioni lambda, il metodo zip e la sua controparte per archiviare file, e la libreria NumPy.

 # 6 – Approfondiamo PHP

Operazioni sui file
Le funzioni e il costruttore
Sessioni e verifica

E’ sempre più utilizzato da moltissimi sviluppatori. Una delle chiavi del suo successo deriva dal fatto che è open-source, cioè gratuito da scaricare e adoperare. È molto semplice da imparare e da usare, è un linguaggio interpretato e non richiede un compilatore.

Il codice PHP è eseguito nel server e può essere integrato con molti database come Oracle, Microsoft SQL Server, MySQL, PostgreSQL, Sybase, Informix. È sufficientemente potente da contenere un sistema di gestione dei contenuti come WordPress e può essere utilizzato per controllare l'accesso degli utenti. Supporta i principali protocolli come HTTP Basic, HTTP Digest, IMAP, FTP e altri. Anche siti Web come *facebook.com* o *yahoo.com* sono basati su PHP.

Uno dei motivi principali alla base di ciò è che questo linguaggio può essere facilmente incorporato all’interno di un listato HTML e, allo stesso modo, il codice HTML può anche essere scritto in un file PHP.

La cosa che lo differenzia da un linguaggio lato client come l'HTML è che i codici PHP sono eseguiti sul server mentre i primi sono eseguiti e visualizzati direttamente sul browser.

I codici PHP sono prima eseguiti sul server e quindi il risultato è restituito al browser.

L'unica informazione che il client o il browser conosce è il risultato restituito dopo l'esecuzione dello script sul server e non il codice presente nel file o nel listato. Inoltre, il PHP supporta tranquillamente altri linguaggi di scripting lato client come CSS e JavaScript. Altre notevoli caratteristiche che lo contraddistinguono le possiamo riassumere così:

- *Semplice e veloce*
- *Efficiente*
- *Sicuro*
- *Flessibile*

- *Multipiattaforma*, funziona con i principali sistemi operativi come Windows, Linux, MacOS.

Operazioni sui file

Anche con PHP avremo accesso a svariate funzioni per gestire tutte le operazioni possibili con i file. Iniziamo con:

```
<?php
// Indichiamo il file sul quale desideriamo scrivere
$txt = 'documento.txt';

// Verifichiamo che il file esiste e che possa essere letto e scritto, se è tutto ok apriamo il file
if(!file_exists($txt))
{
exit('Il file '.$txt.' non esiste!');
}
 else if(!is_readable($txt))
{
exit('Il file '.$txt.' non può essere letto!');
}
 else if(!is_writable($txt))
{
exit('Il file '.$txt.' non può essere scritto!');
}

 // Se il form è stato inviato
 if(isset($_POST['scrivi']))
{
// Apriamo il file in modalità lettura-scrittura, se impossibilitati a farlo otterremo un errore
$puntatore = fopen($txt, 'a') or die ('Fallimento apertura del file '.$txt);

// Scriviamo sul file se possibile, altrimenti avremo l'errore
fwrite($puntatore, $_POST['stringa']) or die ('Fallimento scrittura del file '.$txt);

// Chiudiamo il file
fclose($puntatore);
}
```

```
    // Visualizziamo il contenuto del file
    $contenuto = file_get_contents($txt);
    echo $contenuto;
    ?>
    <form action="" method="post">
<input type="text" name="stringa" />
<input type="submit" name="scrivi" value="scrivi" />
</form>
```

Questo esempio mostra superficialmente alcune operazioni che potremo andare a eseguire sui file. Adesso osserviamo anche il lato di PHP rivolto al codice funzionale.

Funzioni

Oltre alle migliaia di funzioni PHP integrate o facilmente importabili disponibili per l'uso, possiamo ovviamente anche creare le nostre funzioni, vediamo la sintassi di base:

```
    function nome_funzione(attributo1, attributo2, ...) {
// Codice funzione
}

    nome_funzione(attributo, attributo, ...);

    // Istruzione successiva dopo che la funzione è stata completata
```

La parola chiave della funzione è minuscola. Il nome fornito per la funzione utilizza quasi lo stesso formato delle variabili, eccetto che non include $. Variabili o valori possono essere passati come parametri nella funzione tra parentesi. Tutto il codice va tra parentesi {}. La funzione è chiamata utilizzando il nome della funzione e il passaggio di eventuali attributi richiesti. Quando è richiamata, l'esecuzione del programma salta alla funzione. Dopo che tutto il codice è stato eseguito, il flusso del programma salta all'istruzione successiva alla chiamata. Vediamo un esempio basico:

```
    <?php
```

```
    function ciao_damario() { print "Ciao da Mario";
}

    ciao_damario();

    ?> // Risultato: Ciao da Mario
```

Abbiamo appena visto come anche se nessun valore è passato alla funzione, l'istruzione *print* l'ha eseguita.

```
    <?php

    function ciao_damario($valore) { print $valore;
}

    ciao_damario("Ciao da Mario");

    ?> // Risultato: Ciao da Mario
```

In questo caso otteniamo lo stesso risultato. Tuttavia, abbiamo una certa flessibilità consentendo all'utente di passare il valore da visualizzare. Si noti che la stringa è stata passata tra parentesi quando è stata chiamata la funzione. La stringa cadrà nella variabile *$valore* (determina dove vanno i valori in base alla posizione in cui sono passati). L'istruzione print nella funzione utilizza quindi la variabile *$valore* per visualizzare le informazioni. Questa funzione in realtà visualizzerebbe quasi tutto ciò che è stato passato (inclusi i numeri), anche se è chiamata *ciao_damario*. Osserviamo un'altra variante:

```
    <?php

    function sport_pref( $primo_sp, $secondo_sp = "nessuno") { print
" Il tuo sport preferito è il $primo_sp";

    if ($secondo_sp != "nessuno") {
print " Il tuo secondo sport preferito è il $secondo_sp";

    }
}
```

```
    sport_pref("Calcio."); sport_pref("Pugilato.", "Tennis.");


    ?> // Risultato: Il tuo sport preferito è il Calcio. Il tuo sport
preferito è il Pugilato.
Il tuo secondo sport preferito è il Tennis.
```

Abbiamo visto come la funzione *sport_pref* accetta due valori (*$primo_sp, $secondo_sp*). Tuttavia, fornisce anche un valore predefinito per il secondo parametro. Infatti, nella prima chiamata alla funzione, "*Calcio*" passerà in *$primo_sp*. Poiché non è stato passato un secondo parametro, *$secondo_sp* conterrà "nessuno". Sarà visualizzato "Il tuo sport preferito è il Calcio". L'istruzione *if* determinerà che un secondo valore non è stato passato e non tenterà di visualizzare *$secondo_sp*. Nella seconda chiamata vengono passati entrambi i valori. "Pugilato" verrà passato in *$primo_sp*. "Tennis" verrà passato in *$secondo_sp*. La funzione visualizzerà "Il tuo sport preferito è il Pugilato" e "Il tuo secondo sport preferito è il Tennis".

```
    <?php
    function somma($primo_val, $secondo_val) {

    $risultato = $primo_val + $secondo_val; return $risultato;
}
print somma( 30, 5);

    ?> // Otterremmo 35
```

Qui nella funzione abbiamo passato due valori numerici. La chiamata alla funzione fa sì che 30 sia passato in *$primo_val* e 5 sia inserito in *$secondo_val*. I due numeri sono sommati e il risultato è inserito in *$risultato*. L'istruzione *return* restituisce il valore al programma che l'ha chiamato (invece di visualizzarlo). Ciò consente al codice chiamante la flessibilità di determinare cosa fare con il valore restituito. In questo esempio, la funzione è stata chiamata all'interno di un'esecuzione di stampa. In questo modo sarà visualizzato il valore restituito dalla funzione *somma* (35).

```
    <?php declare(strict_types=1);
    function somma( int $primo_val, int $secondo_val): int {
```

```
$risultato = $primo_val + $secondo_val; return $risultato;
}
print somma ( 30, 5);

?> // Il risultato è sempre 35
```

In PHP 7 possiamo aggiungere suggerimenti di tipo *scalare* (solo numeri e unità di misura) per limitare il tipo di informazioni passato dentro e fuori una funzione. Nell'esempio precedente, i parametri passati sono limitati a numeri interi, solo come indicato dalla parola chiave *int* prima dei nomi della variabile. Il valore restituito è anche limitato a intero, come indicato da *: int* come parte dell'intestazione della funzione. *strict_type* deve essere impostato su 1 per l'imposizione. Se fosse impostato su zero, cioè l'impostazione predefinita, i tipi di dati visualizzati sarebbero ignorati.

```
<?php declare(strict_types=1);

function somma( int | float $primo_val, int | float $secondo_val)
: int
| float {

$risultato = $primo_val + $secondo_val; return $risultato;
}

print somma( 30.5, 5);

?> // Il risultato con la virgola è 35.5
```

PHP 8 introduce l'unione di diversi tipi di dati. Nell'esempio precedente, la funzione ci ha offerto la possibilità di accettare numeri interi o a virgola mobile per *$primo_val* e *$secondo_val*. Fornisce anche la possibilità di ottenere dall'output un numero intero o un *float*. Ciò fornisce una convalida aggiuntiva dei dati pur consentendo flessibilità nell'uso effettivo della funzione stessa.

A partire da PHP 8, gli attuali tipi di dati validi sono: *array* (insiemi o vettori), *bool* (booleani), *callable* (funzioni richiamabili in un secondo momento), *int* (numeri interi), *float* (numeri in virgola mobile), *null* (variabile con nessun valore), *object* (oggetti software), *resource* (un tipo di dato che si riferisce ad una risorsa esterna), *string* (stringhe di testo)

Le funzioni possono anche essere incluse in un file separato che potrà quindi essere importato nel programma PHP principale. Difatti, utilizzando le funzioni, scopriremo che alcune potrebbero essere utilizzate in altre applicazioni. È possibile spostare queste funzioni in un file separato e importarle in un'applicazione.

```
   <?php declare(strict_types=1);

function somma( int $primo_val, int $secondo_val): int {
  $risultato = $primo_val + $secondo_val; return $risultato;
   }

   ?>
```

Le funzioni che risiedono all'interno dei propri file dovrebbero comunque includere l'apertura e chiudendo i tag php come mostrato in precedenza.

```
   <?php

   ?>

   include "somma.php"; print somma(30, 5);
```

Il programma precedente importerà il file *somma.php* (che contiene l'omonima funzione *somma*). Una volta importato, può chiamare la funzione come mostrato.

Attenzione: quando s'importano più file, è possibile che si generi un conflitto se più funzioni importate hanno la stessa firma (nome e parametri).

La parola chiave *include* cercherà il file e tenterà di includerlo nel programma. Se il file non esiste, il programma continuerà. La parola chiave *include_once* è simile a *include*, effettuerà solamente un ulteriore controllo per scoprire se il file è già stato importato. In caso affermativo, ignorerà la richiesta senza produrre errori, al contrario dell'altra istruzione.

Anche la parola chiave *require* è simile alla parola chiave *include*. Tuttavia, se il file non esiste, sarà prodotto un errore. La parola chiave *require_once* è simile alla parola chiave *require* con il controllo aggiuntivo per non caricare il file se è già stato caricato.

Gli esempi mostrati non tentano di gestire eventuali errori. Ci sono più problemi possibili con questi esempi, se l'utente non inserisce ciò che si aspettava. Possiamo adattare il programma chiamante per gestire possibili problemi aggiungendo una struttura *try/catch*.

```
<?php

try {

}

include "dividi.php"; print dividi(30, 5);

catch(zeroException $e) {
print "Non è possibile dividere per zero!";
}

catch(Throwable $t) {
print $t->getMessage();
}

finally {
print "Questo messaggio è finito.";
}
?>
```

Nell'esempio precedente, sia l'istruzione *include* sia *print* sono posizionate in un blocco *try*. Il programma eseguirà le istruzioni finché non incontrerà un problema. Quando accade, cercherà un blocco *catch* per gestire il problema. Poiché l'istruzione *include* dipende da un file esistente esterno al programma, è importante che questo sia in grado di gestire la possibilità che il file non esista.
Questo esempio inserisce anche la funzione *dividi* all'interno del blocco *try*. Se questa funzione tentasse di dividere per zero, PHP solleverà un'eccezione. Il codice cattura in modo speciale l'eccezione *zeroException* che sarebbe sollevata da PHP se si tentasse di dividere per zero. In tal caso, sarà visualizzato il messaggio mostrato nel blocco e il programma si arresterà correttamente (non lo farà in modo anomalo).
Sono mostrati ulteriori blocchi di cattura. *Throwable* cattura tutte le altre

eccezioni causate dal programma. Se l'esecuzione del codice fa saltare il flusso in uno di questi blocchi, sarà visualizzato il messaggio di errore standard e il programma eseguirà il blocco *finally*. Il blocco *finally* è eseguito anche quando non sono rilevate eccezioni. È importante assicurarsi che i programmi live non vadano in crash. È meglio catturare eventuali problemi e quindi visualizzare un messaggio per l'utente che richiede di provare a utilizzare nuovamente il sistema in un secondo momento.

Il costruttore

Nel corso del nostro studio con il PHP potremo aver bisogno di creare diversi oggetti e assegnargli dei valori. Se fossero parecchi, avremmo dei problemi, sia di *boilerplate code* sia di possibili bug. Per fortuna possiamo utilizzare un modo molto più efficiente per aggiornare tutte queste proprietà contemporaneamente.

Quando un'istanza di una classe (oggetto) è creata in memoria, il sistema operativo esegue un metodo costruttore che crea l'oggetto con qualsiasi proprietà e metodo esistente. Il sistema crea anche tabelle in memoria per tenere traccia della posizione dell'oggetto e dei valori che esistono nelle proprietà dell'oggetto. Quando l'oggetto non è più necessario, il *Garbage Collector* del sistema operativo sarà chiamato dal metodo distruttore dell'oggetto, che lo rimuoverà dalla memoria.

Possiamo anche includere un metodo costruttore che sarà chiamato automaticamente quando l'oggetto è inserito in memoria. Quando sarà eseguita la riga di codice che crea l'oggetto (*$prova = new Oggetto;*), sarà cercato un metodo costruttore nell'oggetto. Se esiste, il metodo sarà eseguito. Possiamo passare tutti i valori iniziali per le proprietà in questo metodo di costruzione tramite la stessa riga che crea l'oggetto:

```
$prova = new Oggetto('Mario', 'Pallone', 'Rosso', 33);
```

Il metodo del costruttore è un formato generico con un costrutto del nome di funzione (ricordarsi bene che ci sono due caratteri di sottolineatura prima della parola *construct*):

```
<?php class Oggetto
{
function __construct($valore1, $valore2, $valore3, $valore4)
```

```
  {
codice che aggiorna le proprietà
    }
   metodi
}
   ?>
```

È possibile utilizzare il set di metodi esistenti nel costruttore per aggiornare le proprietà.

Dovremo raccogliere tutti i messaggi (TRUE/FALSE) e restituirli al programma chiamante (*prova.php*):

```
   <?php declare(strict_types=1); class Oggetto
   {

   // private int $peso_ogg = 1;

   private string $nome_ogg = "nome";
private string $tipo_ogg = "oggetto";
private string $colore_ogg = "colore";
private string $messaggio_err = "???";

   function __construct(string $valore1, string $valore2, string
$valore3, int $ valore4)
   {
   string $nome_err = $this->set_nome_ogg($valore1) == TRUE ?
'TRUE,' : 'FALSE,';
   string $tipo_err = $this->set_tipo_ogg($valore2) == TRUE ?
'TRUE,' : 'FALSE,';
   string $colore_err = $this->set_colore_ogg($valore3) == TRUE ?
'TRUE,' : 'FALSE';
string $peso_err = $this->set_peso_ogg($valore4) == TRUE ? 'TRUE' :
'FALSE';
$this->messaggio_err = $nome_err . $tipo_err . $colore_err .
$peso_err;
}
```

```
    public function __toString()
{
return $this->messaggio_err;
}
```

Non ci sono altre modifiche al codice per *prova.php* dopo questa riga.
Per ulteriori informazioni sui metodi del costruttore, visitare la pagina: https://www.php.net/manual/en/language.oop5.decon.php .
Innanzitutto, discutiamo l'uso del metodo speciale chiamato *__toString* nell'Esempio appena visto. Con i metodi del costruttore non possono essere restituite informazioni (per impostazione predefinita). L'istruzione *return* non può essere utilizzata all'interno del costruttore.
Per restituire i messaggi di errore creati nel costruttore dal listato chiamante (*prova.php*), dovremo ingannare il programma. Il metodo *__toString* consente al programmatore di decidere cosa accadrà se si tenta di utilizzare il metodo *print* (o *echo*) con il nome dell'oggetto (*print $prova;*). Questo può essere ignorato includendo il nostro metodo con un'istruzione che restituisce una stringa. Potremo superare questo problema di restituire i messaggi di errore consentendo la restituzione del valore nella proprietà *$messaggio_err* se l'istruzione *print $prova;* è eseguita.
Per ulteriori informazioni sul metodo *__toString* e altri metodi "magici", visitare: https://www.php.net/manual/en/language.oop5.magic.php .
Anche le costanti TRUE e FALSE restituite dai metodi *set* causano un problema perché sono costanti e non stringhe. Se tenteremo di convertire queste costanti in una stringa usando un metodo (come strval(TRUE);), i valori che rappresentano (1 per TRUE, 0 per FALSE) diventerebbero una stringa invece di 'TRUE' o 'FALSE' . Pertanto, non possono essere restituiti tramite il metodo *__toString*. Per superare questo problema, creiamo il codice seguente nel costruttore per eseguire una conversione da TRUE a 'TRUE' o FALSE a 'FALSE':

```
    string $nome_err = $this->set_nome_ogg($valore1) == TRUE ?
'TRUE,' : 'FALSE,';
```

L'ordine delle operazioni farà sì che il metodo *set_nome_ogg* sia eseguito prima di qualsiasi parte di questo codice. Il metodo restituisce *TRUE* o *FALSE* (costanti). Supponendo che il metodo restituisca un TRUE dopo l'esecuzione, la riga di codice sarebbe ora:

```
$nome_err == TRUE == TRUE ? 'TRUE,' : 'FALSE,';
```

L'ordine delle operazioni richiede quindi che il confronto (*TRUE == TRUE*) sia valutato. Naturalmente, questo restituisce VERO. Sarà estrapolata la dichiarazione tra il punto interrogativo e i due punti: *$nome_err = 'TRUE,';*
Perciò *$nome_err* è impostato sulla stringa "TRUE", è ora essa stessa lo è, non una costante. Si noti inoltre che è stata aggiunta una ',' in preparazione per il successivo valore 'VERO' o 'FALSO'.

Ogni valore passato (tranne l'ultimo) deve essere separato da una ',' per consentire la separazione della stringa in un secondo momento.

Le altre tre linee simili sono valutate e inseriscono anche un "VERO" o "FALSO" nel campo proprietà del messaggio di errore (la valutazione dello sbaglio di peso non include una virgola alla fine della stringa poiché è l'ultima valutata). Siamo all'ultima riga di codice del costruttore:

```
$this->messaggio_err = $nome_err . $tipo_err . $colore_err . $peso_err;
```

Questa riga inserisce i valori inclusi nelle diverse proprietà all'interno di *messaggio_err*. Se tutti gli aggiornamenti hanno avuto esito positivo, la proprietà *$messaggio_err* conterrebbe: "VERO, VERO, VERO, VERO".

Si noti che ogni elemento passato include una "," per la separazione tranne l'ultimo elemento. Ciò è necessario per spezzare i risultati della stringa.

Vediamo il file *prova.php* che chiama un costruttore:

```
<?php declare(strict_types=1);

require_once("prova.php");
$prova = new Oggetto('Mario', 'Pallone', 'Rosso', 33);
list($nome_err, $tipo_err, $colore_err, $peso_err) = explode(',', $prova);

print $nome_err == 'TRUE' ? 'Nome ok! <br/>' : 'Aggiornamento nome non riuscito<br/>';
print $tipo_err == 'TRUE' ? 'Oggetto ok! <br/>' : 'Aggiornamento oggetto non riuscito<br/>';
```

```
print $colore_err == 'TRUE' ? 'Colore ok! <br/>' : 'Aggiornamento colore non riuscito<br/>';
print $peso_err == 'TRUE' ? 'Peso ok! <br/>' : 'Aggiornamento del peso non riuscito<br/>';
```

Avrete notato una piccola variazione nella creazione dell'oggetto sulla terza riga dell'esempio appena visto:

```
$prova = new Oggetto('Mario', 'Pallone', 'Rosso', '33');
```

Stiamo ora passando i valori iniziali (*Mario, Pallone, Rosso* e *33*) nell'oggetto tramite il costruttore. Altrimenti, dovremmo effettuare quattro chiamate per determinare i metodi (*set_nome_ogg, set_tipo_ogg, set_colore_ogg* e *set_peso_ogg*) per ottenere la stessa cosa. Questo ci consente di utilizzare i metodi stabiliti per gli aggiornamenti necessari dopo aver impostato inizialmente l'oggetto (*$prova*).
Per determinare se gli aggiornamenti delle quattro proprietà hanno avuto esito positivo, dobbiamo recuperare i valori (TRUE, TRUE, TRUE, TRUE) dalla proprietà *$messaggio_err* nell'oggetto. Il metodo *__toString* nella classe *Oggetto* ci consente di farlo trattando *$prova* come se fosse una stringa. Questo ci consente di utilizzare il metodo *explode*:

```
list($nome_err, $tipo_err, $colore_err, $peso_err) = explode (',', $prova);
```

Questa riga di codice interromperà il contenuto di *$prova(TRUE, TRUE, TRUE, TRUE)* con le virgole e collocherà ciascuna parte nelle proprietà *$nome_err*, *$tipo_err*, *$colore_err* e *$peso_err*. Si spera che ciascuna di queste proprietà contenga la stringa 'TRUE'. Possiamo quindi valutare i messaggi per vedere se gli aggiornamenti hanno avuto successo in una tecnica molto simile che ha valutato i risultati dei metodi impostati:

```
print $nome_err == 'TRUE' ? 'Nome ok! <br/>' : 'Aggiornamento nome non riuscito<br/>';
```

Ci sono solo alcune piccole differenze tra questo formato e le dichiarazioni simili per valutare i risultati dei metodi impostati. Ognuna di queste righe di codice utilizza un messaggio di errore diverso per la

valutazione (*$nome_err*, *$tipo_err*, *$colore_err* e *$peso_err*). Avremmo anche potuto usare (*$messaggio_err*) per tutti i risultati dei metodi impostati, anche se, non sarebbe stato altrettanto efficiente.
Infine, ricordiamoci bene la sostanziale diversità tra la stringa '*TRUE*' rispetto alla costante *TRUE* (nel listato noteremo solo la presenza o meno delle virgolette).

Sessioni e verifica

Nessuna discussione sulla sicurezza sarebbe completa senza includere l'autenticazione tramite ID utente/password. L'attuale versione di PHP include molte tecniche per assistere gli sviluppatori nella convalida degli utenti. Questo capitolo esamina uno dei metodi più semplicistici.
In base alla natura della verifica immediata di credenziali per l'accesso, il processo di autenticazione accede direttamente all'origine dati per la convalida (non passa attraverso il livello delle regole aziendali). Pertanto, il processo di autenticazione è considerato un livello separato posizionato sopra l'applicazione per fornire l'accesso. Come vedrai, nei programmi di livello dell'interfaccia devono essere apportate solo modifiche minori per limitare l'accesso. La maggior parte della codifica necessaria è inserita nel livello di autenticazione.
Oltre all'autenticazione, durante il processo di accesso è possibile determinarne anche il livello. Non tutti gli utenti necessitano dell'accesso completo a un'applicazione, alcuni potrebbero essere limitati solo in lettura, altri avranno bisogno dell'accesso in scrittura solo sulle informazioni a loro pertinenti e alcuni (ad esempio gli amministratori) necessiteranno dell'accesso completo. Ciascuna parte dell'applicazione deve essere in grado di determinare il corretto livello di accesso senza richiedere informazioni aggiuntive all'utente (oltre al login originale all'applicazione).
Un processo di accesso deve consentire agli utenti di verificare tutte le parti dell'applicazione. Ciascuna parte dell'applicazione richiede l'ammissione a proprietà comuni (come ID utente e password) impostate dal livello di autenticazione per la verifica dell'ingresso e dei livelli di accesso validi. PHP offre la possibilità di memorizzare informazioni per un'applicazione nella memoria del server dichiarando una sessione. Si considera che una sessione includa la completa interazione dell'utente con l'applicazione (come i processi completi riguardanti trasferimenti di denaro da libretto a un conto corrente). Una sessione può essere stabilita non appena l'utente accede all'applicazione. La sessione può essere chiusa dopo che l'utente si è

disconnesso dal sistema (oppure l'applicazione è scaduta o è stata chiusa). Quando una sessione è chiusa, tutte le proprietà archiviate nella memoria del server sono rimosse dal Garbage Collector.
Mentre la sessione è attiva, le proprietà possono essere archiviate e condivise in tutta l'applicazione. Utilizzando questo processo, l'ID utente e la password possono essere archiviati nelle proprietà della sessione. Ciascuna parte dell'applicazione può quindi verificare che l'utente abbia effettuato l'accesso determinando se sono presenti valori nelle proprietà userid e password.
Prima di esaminare il processo "*autenticazione* e *accesso*", diamo un'occhiata a come determinare se un utente ha effettuato l'accesso al sistema. Gli esempi in questo capitolo non includono le verifiche dei livelli di accesso di sicurezza. Tuttavia, il processo per determinare questi livelli utilizzerà un codice simile come mostrato in questi esempi.
Concludendo, la chiamata al metodo *session_start* deve essere la prima istruzione all'inizio del codice. Non devono esserci spazi o codice tra il tag *<?php* e *session_start*. Questo metodo produce un'intestazione HTML che non sarebbe formata correttamente se esistesse del codice prima della sua chiamata.

```
   <?php
session_start();
if ((!isset($_SESSION['username'])) || (!isset($_SESSION
['password']))) {
echo "Effettuare il login per accedere alla scelta dell'auto";
echo "<p>";
echo  "<a  href='login.php'>Accedi</a>  |  <a  href='registrati.php'>
Crea un account</a>";
echo "</p>";

   }
else {
echo "<p>Bentornato, " . $_SESSION['nome utente'] . "</p>";
}
?>
```

Ciascun programma nel livello dell'interfaccia (a cui l'utente può accedere) includerebbe il codice simile all'esempio precedente. Il metodo *session_start*, in questo esempio, fa sapere al sistema operativo che questo

programma fa parte di una sessione esistente (che è dichiarata nel livello di autenticazione). Ogni sessione è identificata dal sistema utilizzando un ID generato in modo univoco. Finché l'utente (o il sistema) non ha chiuso la sessione, l'ID sessione sarà allegato a qualsiasi programma, chiamato dall'utente, che include il metodo *session_start*. Ciò consente al programma di accedere a tutte le proprietà concernenti la sessione corrente.

Il metodo PHP *isset* (in un'istruzione if) può determinare se esistono valori nelle proprietà di nome utente e password. Se i valori non esistono, indica che l'utente non è stato autenticato. Le proprietà della sessione sono recuperate (e impostate) utilizzando *$_SESSION*. Nell'esempio precedente, se una delle proprietà non è impostata, all'utente sono forniti i collegamenti alla pagina di accesso (*login.php*) o alla pagina di registrazione (*registrati.php*). Se sono impostate entrambe le proprietà, l'utente sarà accolto nel sistema. Proseguiamo:

```
   <?php
   session_start();
   if ((!isset($_SESSION['username'])) || (!isset($_SESSION
['password']))) {
   echo "Effettuare il login per accedere alla scelta dell'auto";
   echo "<p>";
   echo "<a href='login.php'>Accedi</a> | <a href='registrati.php'>
Crea un account</a>";
   echo "</p>";

   }
   else {
   echo "<p>Bentornato, " . $_SESSION['nome utente'] . "</p>";
   }
   ?>

   <!DOCTYPE html>
   <html lan="it">
   <head>
   <title>Scelta auto</title>
   </head>
   <body>
   <h1>Scegli la tua auto</h1>
```

```
.....
</body>
</html>

<?php
}
?>
```

Nell'esempio appena visto, il codice di verifica è posto nella parte superiore del programma. L'istruzione *else* deve includere tutto il codice da eseguire se l'utente è connesso al sistema. Poiché il linguaggio per questo programma è codice HTML (e CSS), l'istruzione *if* deve essere racchiusa attorno al codice esistente. Le parentesi di chiusura dell'istruzione *else* sono spostate nella parte inferiore del codice (dopo tutti i tag HTML).

PHP ci consente di chiudere il codice tramite *?>* e di riaprirlo tramite *<? php*, tutte le volte che ne avremo bisogno. In questo esempio, il codice PHP è chiuso nella parte superiore del programma (nell'istruzione *else*) appena prima della parentesi di chiusura. Il codice PHP è quindi riaperto nella parte inferiore del listato per includere una singola parentesi di chiusura, che chiude l'istruzione *else*. Questo avvolge l'istruzione attorno a tutto il codice esistente. Gli utenti potranno accedere a questa sezione del codice solo se hanno effettuato l'accesso.

Ora diamo un'occhiata a come popolare le proprietà della sessione creando un programma di accesso. Per prima cosa, diamo un'occhiata all'HTML per richiedere informazioni agli utenti.

```
<form method="post" action="">
Nome utente: <input type="text" pattern=".{8,}" title="L'ID utente deve contenere otto o più caratteri." name="nomeutente" id="nomeutente" obbligatorio/><br />
Password: <input type="password" pattern="(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}" title="La password deve contenere almeno un numero, una lettera maiuscola e minuscola e almeno 8 caratteri totali." name="password" id="password" obbligatorio /> <br />
<input type="submit" value="Login">
</form>
```

HTML5 non include un parametro di lunghezza minima. Tuttavia, il parametro *pattern* può essere utilizzato (con espressioni regolari) per stabilire una dimensione minima. Nel tag *nome utente* nell'esempio precedente, il modello *".{8,}"* richiede almeno otto caratteri inseriti dagli utenti. Per la sicurezza della password, uno schema più complicato è necessario. Nell'esempio della password, oltre al requisito minimo di otto caratteri, almeno un numero *(?=.*\d),* una lettera maiuscola *(?=.*[A-Z])* e una lettera minuscola *(?=.* [a-z])* sono obbligatori.
Il filtro HTML fornito è utilizzato per informare gli utenti di eventuali errori di battitura che potrebbero essersi verificati. Nel processo di accesso, non stiamo memorizzando informazioni, ma solo confrontandole con ciò che è già stato memorizzato. Non dovremo preoccuparci di eventuali informazioni dannose trasmesse nelle caselle di testo. Qualsiasi cosa non corrisponderebbe alle informazioni valide memorizzate. L'utente riceverà un messaggio di ID utente/password non valido.

```
    // Processo di convalida non mostrato
$_SESSION['nomeutente'] = $_POST['nomeutente'];
$_SESSION['password'] = $_POST['password'];
//  Reindirizza  all'home  page  ("Location:  http://www.prova.it/
prova.php");

    }
?>
```

Supponendo di aver convalidato le informazioni rispetto a un elenco di ID utente e password validi (esamineremo presto questo processo), possiamo passare le informazioni sull'ID utente e sulla password valide nelle variabili di sessione. Quindi il metodo dell'intestazione PHP può essere utilizzato per reindirizzare l'applicazione al prossimo programma da eseguire (*prova.php*). Finché *prova.php* includerà *session_start* (come mostrato in precedenza), avrà accesso alle variabili di sessione.

```
    <?php
?>

    <?php
session_start();
```

```
if ((!isset($_POST['username'])) ||
(!isset($_POST['password'])))
{
<form method="post" action="">
Nome utente: <input type="text" pattern=".{8,}" title="L'ID utente deve contenere otto o più caratteri." name="nomeutente" id="nomeutente" obbligatorio/><br />
Password: <input type="password" pattern="(?=.*\d)(?=.*[a-z]) (?=.*[A-Z]).{8,}" title="La password deve contenere almeno un numero, una lettera maiuscola e minuscola e almeno 8 caratteri in totale." name="password" id="password" obbligatorio /><br />
<input type="submit" value="Login">
</form>

   } else {
// Processo di convalida non mostrato
$_SESSION['nomeutente'] = $_POST['nomeutente'];
$_SESSION['password'] = $_POST['password'];
//  Reindirizza  all'home  page  ("Location:  http://www.prova.it/prova.php");

   }
?>
```

Mettere insieme i pezzi richiede un'istruzione *if* per determinare se l'utente ha inserito l'ID utente e la password. Se non l'ha fatto, sarà visualizzato il codice HTML per richiederli. Se l'informazione è stata inserita (ed è valida usando le espressioni del pattern HTML5 mostrate), sarà eseguita la parte *else* dell'istruzione (memorizzazione dei valori nelle variabili di sessione e chiamata al programma *prova.php*). Questo ci fornisce la shell di base per accettare l'ID utente e la password, verificarne l'esistenza e chiamare il programma nel livello dell'interfaccia se esistono. Ovviamente, dobbiamo autenticare l'ID utente e la password prima di chiamare il programma.

```
   header('WWW-Authenticate:  Basic  realm="Scelta  Auto"');
header('HTTP/1.0 401 Non autorizzato');
```

È possibile creare messaggi d'intestazione non autorizzati se l'utente non ha immesso un ID utente/password valido. Ciò farà automaticamente in modo che il sistema richieda all'utente nuovamente di inserire un ID utente/ una password. Questa tecnica è piuttosto semplice. Tuttavia, ci sono state alcune segnalazioni da chi l'ha utilizzata in passato, di browser che non funzionavano correttamente. Inoltre, creare la nostra tecnica ci consente di progettare la schermata di accesso con lo stesso stile del resto del nostro sito Web.

```
   $useridpass_valide = array("accesso" => "funzionante");
$idut_validi = array_keys($useridpass_valide);
$idutente = $_SESSIONE['nomeutente'];
$password = $_SESSIONE['password'];
$valid = (in_array($idutente, $idut_validi)) && ($password == $useridpass_valide[$id_utente]);
If($valid) { header("Location: http://www.prova.it/prova.php");}
```

Esistono diversi modi per autenticare ID utente e password. Se si sta creando un sistema che non richiede la modifica di ID utente e password, è possibile utilizzare gli array (insiemi). Nell'esempio precedente, l'array associato *$useridpass_valide* contiene la combinazione di ID utente e password valide. Il metodo PHP *array_keys* inserisce tutte le chiavi (in questo esempio gli ID utente) in un array separato (*$idut_validi*). Dopo, le variabili di sessione sono state poste in *$idutente* e *$password*, l'*in_array* di PHP viene utilizzato per determinare se esiste la combinazione corretta di ID utente e password. *in_array* determina se l'ID utente esiste nell'array. Perciò l'ID utente è utilizzato come pedice per estrarre la password dall'array *useridpass_valide* e confrontarla al valore in *$password*. Se l'ID utente esiste e le password sono le stesse, allora tutto è ok. *$valid* conterrà TRUE. Se uno (o entrambi) non corrispondono *$valid* conterrà FALSE. Se *$valid* è TRUE, l'applicazione reindirizza al programma *prova.php*.
Tecnicamente le proprietà in una sessione sono protette da qualsiasi accesso esterno alla sessione. Tuttavia, in passato sono stati segnalati casi di programmi di hacker che violano questa sicurezza e accedono alle informazioni sulla sessione. Se l'ID utente e la password, in questo ad esempio, sono archiviate in variabili di sessione e passati attraverso Internet a un altro programma, gli hacker potrebbero accedere alle informazioni.
Se l'ID utente e la password sono memorizzate esternamente in un file o in

un database, le informazioni viaggeranno anche al di fuori del programma. Il programma non avrà più il controllo sulla sicurezza di questi elementi una volta che risiedono nel file o nel database. Ciò potrebbe consentire agli hacker di accedere alle informazioni. La sicurezza, come accennato, deve essere uno sforzo di squadra tra il programmatore, l'amministratore dei dati e l'amministratore di rete.
È pratica comune eseguire l'hashing della password per ridurre la possibilità che gli hacker scoprano le informazioni di autenticazione (o qualsiasi altra informazione sicura). A partire da PHP 5.5, il metodo *password_hash* può essere utilizzato per proteggerle.
Bisognerà prestare attenzione durante la memorizzazione della versione con hash della password. La dimensione dell'hash risultante aumenterà con le nuove versioni dell'hash. È probabile che una dimensione di 255 caratteri sia abbastanza grande per molti anni. Il numero di millisecondi necessari per questo hash aumenta con la dimensione e il tipo di crittografia. I programmatori avanzati potrebbero voler fare dei test sui loro server per i costi di tempo. Per eseguire un hash basterà lanciare il seguente codice:

```
echo password_hash("Mario", PASSWORD_DEFAULT);
```

Per ulteriori informazioni potrete dare un'occhiata alla pagina ufficiale: https://www.php.net/manual/en/function.password-hash.php .

Ricordiamoci che non è possibile eseguire un semplice confronto tra la password con hash creata dal metodo *password_hash* di PHP. L'hash prodotto include il tipo di crittografia e la password. A partire da PHP 7.0.0, questo metodo utilizza un'impostazione *salt* predefinita e non consente altre opzioni *salt*, tradotto, significa che quest'opzione aumenta la sicurezza della password aggiungendogli una stringa prima dell'hash. Di conseguenza dobbiamo solo sostituire una riga di codice nell'esempio per verificare la password, quindi la seguente riga:

```
$valid = ((in_array($idutente, $idut_validi)) && ($password == $useridpass_valide[$idutente]));
```

La cambieremo con questa qui sotto:

```
$valid =((in_array($idutente, $idut_validi)) && (password_verify($password, $useridpass_valide[$userid]));
```

Se abbiamo inserito la password con hash nell'array *$useridpass_valide*, la tecnica di convalida non richiederebbe altre modifiche.
Il metodo PHP *password_verify* eseguirà l'hashing della password fornita dall'utente e la confronterà con la password esistente. Se corrispondono, restituirà *TRUE*:

```
   <utenti>
<utente>
<userid>Mario</userid>
<password>$2y$10$BCz7NRHbL6VJd97ptjU6buqFrmVwOCNOcVyViLBkAYHREcBYjbj
p2</password>
</utente>
<utente>
<userid>Rossi</userid>
<password>$2y$10$gELwuCzFh6JSJxj02jl/auPCS.viavY6fJk6ZZS5TvVWTssQI/M
RW</password>
</utente>
</utenti>
```

Questo è solo un esempio banale, per approfondire l'argomento avremmo bisogno di un altro libro dedicato, quindi passiamo oltre, parleremo del multithreading e nel prossimo capitolo approfondiremo alcuni argomenti importanti.

Il multithreading

Il multithreading in PHP può essere utilizzato beneficiando sia di *pthread* sia dell'estensione *parallel*. Di conseguenza, com'è ovvio, saremo in grado di svolgere più lavori contemporaneamente.
Potremo servirci dell'estensione *pthreads* fino alla versione 7.4 di PHP. Tuttavia, se stiamo usando una versione superiore, sarebbe fantastico avere l'estensione *parallel*. Entrambe possono essere scaricate dalla libreria "PECL" (https://pecl.php.net/), un utile "deposito" che ci abitueremo a sfruttare.
Il prerequisito per abilitare il multithreading riguardano una build PHP abilitata per ZTS (*Zend Thread Safety*) in anticipo, non potrà essere fatto dopo la compilazione.
Per abilitare il multithreading in PHP ci servirà il server XAMPP (https://www.apachefriends.org/it/index.html), vediamo i passaggi:

Scarichiamo l'ultima versione di *pthreads* o *parallel* dalla libreria di estensioni "PECL".
Copiamo e incolliamo il file "*pthreadsVC2.dll*" o "*parallelVC2*" nella cartella "PHP" che si trova all'interno della cartella "XAMPP".
Copiamo e incolliamo il file "*pthreads.dll*" o "*parallel.dll*" nella cartella "EXT" che si trova all'interno della cartella "PHP".
Apriamo il nostro file php.ini e scriviamo al suo interno il comando "*extension=php_pthreads.dll*" o "*extension=php_parallel.dll*".
Salviamo il nostro file *php.ini* e una volta riavviato il tutto, se abbiamo seguito correttamente i passaggi, il multithreading sarà abilitato e attivo.

Usiamo Pthread

Innanzitutto, dovremo creare una classe che estenda la classe *Thread*. Successivamente, definiremo un costruttore pubblico e una funzione pubblica "*run*". Successivamente, creeremo un array vuoto insieme all'inizializzazione della classe specificata più volte.
Infine, avvieremo i thread usando la funzione *start()* ed eseguiremo la funzione *join()* per attendere che il thread finisca la sua pausa.
Vediamo un esempio di script che utilizza *Pthread*:

```
   <?php
class Prova extends Threads{
public function __construct($id){
$this->$id = $id;
}
public function run(){
  $sleep = mt_rand(1, 10);
   echo “Partito: ” . date(“h:i:sa”) . ” : {$this->id} e sospeso
$sleep”;
  sleep($sleep);
  echo date(“h:i:sa”) . ” : {$this->id} Stop”;
}
}
$nuovo_array = array();
for($i=0; $i < 4; $i++){
$nuovo_array[$i] = new Prova($i);
```

```
   // Facciamo partire i threads
$nuovo_array[$i]->start();
}
foreach(range(0,4) as $i){
$nuovo_array[$i]->join();
}
?>
```

Iniziamo creando una classe che estende la classe Threads, dopodiché definiremo un costruttore con *public function __construct($id){*. Proseguiamo con una funzione pubblica *run()* e stabilendo il tempo di pausa del thread con *$sleep = mt_rand(1, 10);*. Infine ci serviremo di due cicli e di un insieme (array) vuoto per far girare il tutto.
Adesso vediamo un altro esempio dove, dopo aver esteso la classe Threads, la classe data accetterà un URL durante la creazione dei suoi oggetti. Se l'URL è passato al costruttore della classe, sarà recuperato il contenuto dell'URL specificato. Sarà invece stampato un messaggio informativo.
La classe indicata avrà quindi due proprietà pubbliche tra cui *url1* e *data1*. Inoltre, un costruttore sarà dichiarato e definito con un solo parametro "*url1*". Successivamente, la funzione *run()* assicurerà che l'URL sarà fornito per eseguire le attività di conseguenza.
Infine, creeremo un oggetto della classe data, calcoleremo l'ora di inizio e di fine della richiesta e stamperai i byte ricevuti del contenuto dell'URL.
Ecco lo script di codice che si allinea con l'esempio di multithreading PHP che è stato spiegato sopra:

```
   <?php
class Prova extends Thread{
public $url1;
public $data1;
public function __construct($url1){
$this->url1 = $url1;
}
public function run() {
  if (($url1 = $this->url1)) {
  $this->data1 = file_get_contents($url1);
  } else
          printf("Nessuna URL dal thread #%lu </br>", $this-
```

```
>getThreadId());
  }
}
$time = microtime(true);
$obj = new Prova(sprintf("http://www.google.it/?q=%s", mt_rand() *
10));

   // Qui facciamo partire la sincronizzazione
if($obj->start()){
  printf("La richiesta è partita dopo %f secondi", microtime(true) -
$time);
  while($obj->isRunning()){
  echo ".";
  usleep(100);
}
if($obj->join()){
printf(" e ci sono voluti %f secondi per ricevere %d bytes <br>",
microtime(true) - $time, strlen($obj->data1));
} else
printf(" e ci sono voluti %f secondi per finire. Richiesta non
andata a buon fine<br>", microtime(true) - $time);
}
?>
```

Usiamo Parallel

In questo caso dovremo utilizzare "*parallel/Runtime*" e "*parallel/Channel*" prima di scrivere il codice. Successivamente, definiremo una funzione e quindi inizializzeremo le classi *Runtime* e *Channel*. Infine, eseguiremo i thread, otterremo i dati da *Channel* e dopodiché lo chiuderemo.
Insieme con questo, non dimentichiamoci di racchiudere i passaggi che seguono la definizione della funzione nel blocco *try*. Inoltre, creeremo due blocchi *catch*, uno per rilevare l'errore e l'altro per rilevare l'eccezione.
Vediamo subito un esempio per capire bene tutti questi meccanismi:

```
<?php
use parallelRuntime;
use parallelChannel;
```

```
 $funzione_dei_thread = function(int $id, Channel $channel){
 $riposo = ($id == 2) ? 1 : 2;
 sleep($riposo);

 echo “Thread ID: $id e tempo di pausa: $riposo secondi.”;
 $channel->send($riposo);
 };
 try{
$uno = new Runtime();
$due = new Runtime();

$canale = new Channel();
$contenuto = array();
$contenuto[0] = null;
$contenuto[1] = $canale;
 // Facciamo partire i due thread
 $contenuto[0] = 1;
 $uno->run($funzione_dei_thread, $contenuto);
 $contenuto[0] = 2;
 $due->run($funzione_dei_thread, $contenuto);
 // Riceviamo il nostro dato da Channel
 $a = $canale->recv();
 $b = $canale->recv();
 $canale->close();
 // Vediamo il risultato su schermo
 echo “E’ stato inviato: $a e $b”;
 }
 // Se è presente un errore
 catch(Error $errore){
 echo “Error: “, $errore->getMessage();
 }
 // Se è presente un’eccezione
 catch (Exception $eccezione) {
 echo “Exception: “, $eccezione->getMessage();
 }
 ?>
```

Una volta compreso per bene il listato, dovremmo ricordarci i punti indicati sull'estensione *pthreads* durante la creazione di un programma multithread:

- Non è possibile utilizzare l'estensione *pthreads* in un ambiente server Web.
- È necessario specificare alcune limitazioni e restrizioni durante l'esecuzione dell'estensione *pthreads* per offrire un ambiente stabile.
- Le funzioni di debug *var_dump(), print_r()* e simili non includono la protezione dalla ricorsione (algoritmi espressi in termini di se stessi per semplificare il codice, soprattutto per operazioni matematiche) durante l'esecuzione.
- È necessario fare in modo che gli oggetti estendano la classe *Threaded* prima di usarli in aree multithread delle applicazioni.

In conclusione, per un'ulteriore chiarezza, è bene riassumere e definire alcuni termini inerenti al multithreading in PHP:

– *Thread* è una piccola unità d'istruzione. È eseguita da un processore ed è spesso definito come un'istruzione eseguibile.

– *Threaded* è una classe considerata un'unità derivata da quelle più semplici, cioè dai "*thread*". Troveremo il metodo *run()* all'interno della stessa classe. Il metodo specificato è responsabile dell'esecuzione dei thread.

– *Workers* è una classe utilizzata nel multithreading. La classe indicata è utilizzata per gestire i thread e sincronizzare i risultati.

– *Pools* è una classe responsabile della gestione di oggetti inclusi nella classe *workers*.

Quiz & esercizi

1) Correggere il seguente listato:

```
<?php
function sottrazione($primo_val, $sec_val) {
$risultato = primo_val * secondo_val; return $risultato;
}
print risultato( 5, 5);
?>
```

2) Per quale motivo in alcuni casi dovremo utilizzare lo *strict_types*?
3) A cosa serve il metodo *__toString*?
4) Quali sono le prime cose da controllare nel codice quando ci serviremo del metodo *session_start*?
5) Posso utilizzare il metodo *password_hash* per codificare le password?
6) Se dispongo dell'ultima versione di PHP, potrò servirmi dell'estensione *parallel*? Se sì, per cosa mi sarà utile?
7) Posso utilizzare l'estensione *pthreads* in un ambiente server Web?
8) Correggere le seguenti righe di codice:

```
catch (Exception eccezione) {
echo "Excep: ", eccezione->Message();
}
```

9) Con quale metodo eseguirò i threads in PHP?
10) Per quale motivo in PHP devo utilizzare il carattere $?

Riassunto

Un bel capitolo dedicato al PHP, partendo dalle classiche operazioni sui file. Poi abbiamo visto le funzioni, dove avremo notato il differente approccio rispetto ai linguaggi OOP. Si è parlato anche del costruttore, e poi siamo andati verso dei paragrafi più complessi, dedicati alla sicurezza e alle sessioni.

 # 7 – Ripasso generale

PHP e Python
C#
JAVA

Dopo solamente sei capitoli e, probabilmente qualche settimana di tempo impiegato, forse pochi giorni, siamo infine giunti a un livello decisamente impensabile all'inizio di questo percorso di studi. In teoria dovremmo aver acquisito le basi (e forse anche qualcosa di più) di ben quattro linguaggi di programmazione. Di conseguenza, in questo capitolo andremo proprio a testare questa nostra conoscenza appena acquisita. Esamineremo quindi diversi progetti di programmazione con PHP, Python, C# e Java leggermente più grandi e complessi in rispetto a quelli visti finora. Questi progetti sono pensati per tornare a ribadire alcuni dei concetti più rilevanti discussi in questo libro: variabili, loop, accesso ai file e threading. Se, alla fine dell'ultimo listato, saremo in grado di comprendere e decifrare il flusso del codice di questi progetti, potremo renderci conto e tranquillamente affermare di avere un livello di comprensione più che sufficiente dei linguaggi di programmazione affrontati.
E se a questo punto non saremo ancora sicuri delle nostre capacità di programmatore, il capitolo potrà allora aiutarci a chiarire alcuni concetti.

Perciò non indugiamo oltre e concludiamo l'ultimo passaggio che ci separa dal nostro attestato (virtuale) di programmatore.

Usare file XML in PHP

Nel prossimo listato vedremo come PHP possa, senza caricare alcuna libreria dedicata, gestire anche dei file XML e servirsi dei dati al suo interno. Il file XML in questione contiene le seguenti informazioni:

```
<?xml version="1.0" encoding="UTF-8"?>
<bonifico>
```

```
<banca>Banca ABC</banca>
<importo>1000</importo>
<data>10 Ottobre 2022</data>
<messaggio>Pagamento fattura 1</messaggio>
</bonifico>
```

Mentre il listato che gestirà il file XML (che noi nomineremo *prova.xml*) è questo che segue:

```
<?php
$parser=xml_parser_create();

function start($parser,$element_name,$element_attrs) {
switch($element_name) {
    case "BONIFICO":
    echo "-- Bonifico --<br>";
break;
    case "BANCA":
    echo "banca: ";
break;
    case "IMPORTO":
    echo "Importo: ";
break;
    case "DATA":
    echo "data: ";
break;
    case "MESSAGGIO":
    echo "Messaggio: ";
}
}

function ferma($parser,$element_name) {
echo "<br>";
}

function testo($parser,$data) {
echo $data;
}
```

```
 xml_set_element_handler($parser,"start","stop");
 xml_set_character_data_handler($parser,"char");

 // Apriamo il file XML
 $fp=fopen("prova.xml","r");

 // Leggiamo i dati
 while ($data=fread($fp,4096)) {
xml_parse($parser,$data,feof($fp)) or
die (sprintf("XML Error: %s at line %d",
xml_error_string(xml_get_error_code($parser)),
xml_get_current_line_number($parser)));
 }

 // Con questa funzione terminiamo il parser
 xml_parser_free($parser);
 ?>
```

Si comincia inizializzando il *parser XML* (per analizzare i dati), subito dopo creiamo le funzioni che useremo con i vari gestori degli eventi. La funzione *ferma* sarà adoperata alla fine di un elemento, la funzione *testo* quando si troveranno dei dati stringa.
Ci serviremo del gestore degli elementi, cioè *xml_set_element_handler* e dei dati, cioè *xml_set_character_data_handler*, due funzioni di PHP utili trattare dati XML.

Decodificare jSON in PHP

il prossimo listato servirà per vedere come utilizzare *cURL* al fine di ottenere dati JSON per decodificarli in PHP. L'istruzione *cURL* è l'acronimo di URL del *client*. È uno strumento da riga di comando per inviare e ricevere file utilizzando la sintassi degli URL. *cURL* consente di comunicare con altri server tramite HTTP, FTP, Telnet e altro.

Recupereremo i dati JSON da un sito Web, per farlo ci serviremo di un servizio totalmente gratuito, che fornisce questo tipo di dati per i test, nel listato sarà questo link: https://reqres.in/api/users?page=2 .

Per prima cosa, inizializziamo *curl* usando il metodo *curl_init()*. Poi inviamo la richiesta *GET* al server *reqres.in* utilizzando il metodo *curl_setopt()* con *CURLOPT_URL* per ottenere i dati jSON.

Dopodiché, ordineremo a *curl* di archiviare i dati in una variabile invece di visualizzarli sullo schermo. Questo viene fatto utilizzando il parametro *CURLOPT_RETURNTRANSFER* nella funzione *curl_setopt()*.

Infine eseguiremo *curl* usando il metodo *curl_exec(),* per poi chiuderlo, una volta terminato il compito, usando il metodo *curl_close()*.

Ecco un esempio dimostrativo di queste funzioni:

```
    <?php

    // Inizializziamo curl
    $curl = curl_init();

    // Sfruttiamo reqres.in per testare il programma
    curl_setopt($curl, CURLOPT_URL,
  "https://reqres.in/api/users?page=2");

    // I dati JSON verranno archiviati in una variabile
    curl_setopt($curl,
  CURLOPT_RETURNTRANSFER, true);

    // Eseguiamo curl
    $response = curl_exec($curl);

    // Controlliamo se ci sono errori
    if($e = curl_error($curl)) {
  echo $e;
} else {
    // Decodifichiamo I dati JSON
    $decodedData = json_decode($response, true);

  // E li visualizziamo
    var_dump($decodedData);
}

    curl_close($curl);
?>
```

Il simulatore di chat per Python

Dopo aver studiato le applicazioni PHP per decodificare XML o jSON

proveremo qualcosa di più leggero e divertente con Python, anche se il codice sarà sicuramente più lungo e complesso. Parliamo di un simulatore di chat. Con questo programma, tra le altre, ripasseremo le seguenti funzionalità: la formattazione delle stringhe, la definizione della funzione personalizzata, operazioni sui file, elenchi e iterazioni, enumerazione degli elementi, flusso del programma e cicli semplici.
Anche questo esempio sarà un'applicazione basata esclusivamente sulla console. I due elementi chiave del listato saranno un host di chat virtuale e il suo pubblico virtuale, quest'ultimo rappresentato da un soprannome condiviso di "chat". Entrambi questi attori virtuali genereranno del testo sullo schermo ogni pochi secondi, il cui contenuto sarà determinato da tre file di testo esterni, e quindi modificabili facilmente.
Nel prosieguo esamineremo il programma diviso in alcuni blocchi per essere più approfonditi. Innanzitutto, importeremo i moduli di codice richiesti, cioè: *datetime*, *time* e *random* inclusi con Python. Tuttavia, per il nostro progetto, dovremo anche acquisire un modulo aggiuntivo dal *PyPI* (cioè dal deposito ufficiale online per software di terze parti gratuito) utile a estendere le funzionalità di Python. Ci serviremo di *clrprint* in modo da poter aggiungere testo colorato nei nostri progetti con righe come questa ad esempio: *clrprint("Ciao da Mario!", clr='blue').* Per installare *clrprint* seguire le semplici istruzioni di questa pagina: https://pypi.org/project/clrprint/ .
Iniziamo con la prima parte del listato del nostro programma di chat, poi ne discuteremo i dettagli:

```
   import datetime
import time
import random
from clrprint import *
name = "Discutiamo di calcio"
interesse = 10
punteggio = messaggi = 0
risposte = giu = su = False
interessegiu = "arbitro"
interessesu = "goal"
orario = datetime.datetime.now()

   def scrivi(testo):
stile = random.choice([-1, 0, 1, 2])
```

```
if stile == 1:
testo = testo.capitalize()
elif stile == 2:
testo = testo.upper()
print("Chat: " + timer.strftime("<%A %H:%M> ") + testo)

   def apri(nomefile, elim):
try:
file = open(nomefile, 'r')
print("Caricamento file " + nomefile)
listat = file.readlines()
for i, element in enumerate(listat):
templist[i] = element.replace(elim, '')

   return listat # restituisce l'elenco enumerato
file.close()
except (FileNotFoundError, IOError):
print("Non è possibile leggere il file " +str(nomefile))
```

Dopo aver importato i vari moduli definiamo il nome dell'host della chat virtuale (*discutiamo di calcio*) e iniziamo a impostare il codice assegnando i valori iniziali alle variabili inerenti al punteggio del gioco. Con *datetime.now()* facciamo in modo di recuperare l'ora e la data corrente. Infine tramite la funzione *scrivi()* otterremo un po' di varietà casuale nei testi delle chat: con 1 avremo una lettera maiuscola all'inizio del testo mentre con 2 sarà tutto maiuscolo.
Successivamente, definiamo una funzione personalizzata chiamata *apri* che accetta due argomenti, *nomefile* ed *elim*. Il primo rappresenta semplicemente il nome del file che vogliamo aprire, mentre il secondo ci servirà nel loop seguente. Infatti, nella funzione, utilizziamo una struttura di elenco temporanea, *listat*. I dati saranno inseriti in modo che possano essere enumerati per rimuovere i caratteri della nuova riga. Il contenuto del file sarà letto utilizzando il metodo *readlines*.
Dopo la prima parte di codice, nel successivo blocco andremo a indicare tre liste da utilizzare poi con la nostra funzione *apri*; le quali saranno ognuna contenute in un file di testo (.txt) chiamati, per comodità, allo stesso modo: *interessenorm*, *interesseneg* e *interessechat*.

Una volta aperti avremo tre elenchi pieni di chiacchiere sul calcio pronti per essere sfruttati:

```
    clrprint("DISCUTIAMO DI CALCIO", clr='red')
interessnorm = interesseneg = interessechat = []
interessenorm = apri("interessenorm.txt", "\n")
interesseneg = apri("interesseneg.txt", "\n")
interessechat = apri("interessechat.txt", "\n")
```

Come abbiamo appena visto, in Python, possiamo inizializzare le variabili, incluse le liste, con la stessa variabile e sulla medesima riga usando l'operatore di assegnazione, ad es., *x = y = 3* o *Mario = alto = biondo = []*.

```
    print("Livello di interesse iniziale: " +str(interesse)+ "/100")

    try:
with open("punteggi.txt", "r+") as filepunteggi:
punteggi = filepunteggi.read()
print("Record del gioco: %s" % punteggi)

    # Se punteggi.txt non si trova, ne verrà creato uno nuovo
except (FileNotFoundError, IOError):
filepunteggi = open("punteggi.txt", "w")
filepunteggi.write("1") # scriviamo 1 su punteggi.txt
filepunteggi.close()
```

In questa sezione mostriamo il nome del programma in rosso grazie al modulo *clrprint.* Dopo definiamo tre liste vuote (specificate dalla parentesi quadra) che saranno riempite con i dati dei file di testo che andremo a caricare nelle righe seguenti. Il punteggio è equivalente al nostro livello d'interesse, che sarà misurato nel corso della sessione di chat. Sarà quindi caricato il punteggio più alto ed effettuati alcuni test per essere sicuri che tutto sia a posto.
Nel prossimo blocco avremo a che fare con il motore dell'applicazione, dove avverrà la maggior parte dell'elaborazione. Il simulatore di chat lavorerà tramite il ciclo while 1 < interesse < 100 il quale essenzialmente mantiene attiva l'esecuzione finché la variabile *interesse* è sotto il valore cento. Se dovesse superarlo, il ciclo non sarà più eseguito. Inoltre, sarà chiuso

utilizzando *break* se scendesse sotto di uno.
Il cuore del programma in pratica funzionerà attraverso la variabile chiamata *parole*, dove un elemento sarà selezionato a caso dall'elenco di *interessechat* e quindi inserito in questa variabile per essere poi visualizzato più in basso. Tutto ciò è legato al punteggio della variabile *interesse,* la quale raggiungendo un valore più basso di 30, obbligherà il programma a utilizzare un diverso insieme di chat per il nostro host virtuale, ossia quelle più aspre nel file *interessegiu.txt*. Il valore di *interesse* sarà influenzato da parole chiave specifiche, cioè, come già detto le due variabili omonime *interessegiu* e *interessesu*. Se il programma rileverà una certa parola di attivazione (cioè *arbitro* e *goal*), *interesse* sarà ridotto di un numero compreso tra 1 e 5. Allo stesso modo, una parola di attivazione positiva comporta l'aggiunta dell'intero di un numero compreso tra 1 e 5. A tale scopo, utilizzeremo la riga *random.randint(1 , 5)* per generare un numero intero all'interno dell'intervallo desiderato, assegnandolo poi a un'altra variabile: *cambiointeresse*.
Abbiamo due variabili booleane che si attivano se è rilevata una parola specifica (che abbiamo segnalato), sono chiamate *giu* e *su*. Questi due booleani sono utilizzati per visualizzare messaggi di stato su eventuali cambiamenti dell'interesse nel corso del gioco.
Successivamente, per ottenere un po' di brio al testo, aggiungiamo la possibilità di cambiare casualmente lo stile nel testo della chat, che potrà essere: normale, con la prima lettera maiuscola o tutto maiuscolo. Ciò si ottiene esaminando il valore randomizzato della variabile *stile* (definita all'inizio del listato) una volta ogni ciclo principale. La quale riporterà quattro valori: -1, 0, 1 e 2. Sotto a 1, il testo non subirà variazioni, con 1 il programma richiamerà il metodo *capitalize()*, per mettere la maiuscola all'inizio della riga, mentre con 2 avremo il metodo *upper(),* per rendere tutta la riga maiuscola.
Infine, per rendere il tutto un po' più realistico è stato implementato un ritardo casuale attraverso il metodo *sleep(),* in modo che lo spettatore possa godersi appieno il flusso della discussione in chat:

```
   while 1 < interesse < 100:
orario1 = datetime.datetime.now()
parole = random.choice(interessechat) # Seleziona una riga casuale
```

```
   if interessegiu in parole:
# Trovando “rigore” in parole l’interesse scenderà
cambiointeresse =random.randint(1, 5)
interesse -= cambio interesse
giu = True
if interessesu in parole:
# Qui invece è stata trovata la parola goal
cambiointeresse =random.randint(1, 5)
interesse += cambio interesse
su = True

   scrivi(testo) # Funzione che esegue l'output della chat
messaggi = messaggi + 1

   if giu:
if interesse < 0:
interesse = 0
clrprint("Interesse -" +str(cambiointeresse)+ "! Il tuo livello di
interesse  attuale è " + str(interesse) + "/100", clr='red')
negative = False
if su:
clrprint("Interesse +" +str(cambiointeresse)+ "! Il tuo livello di
interesse attuale è " + str(interesse) + "/100", clr='green')
positive = False

   # Impostiamo un ritardo casuale tra un commento e l’altro
time.sleep(random.randint(1, 3))

   # imposta una probabilità del 50% per una risposta dal nostro
host di chat
discussioni = random.choice([True, False])
if discussioni:
if interesse > 30:
clrprint(name  +  ":  "  +  orario1.strftime("<%A  %H:%M>  ")  +
random.choice(intressenorm), clr='white')
   else:
clrprint(name  +  ":  "  +  orario1.strftime("<%A  %H:%M>  ")  +
random.choice(interesseneg), clr='white')
```

Dopo un ciclo principale piuttosto intenso, raggiungiamo finalmente l'ultimo segmento nel nostro simulatore di chat. Essenzialmente ci occuperemo di contare i punti, in base all'andamento del gioco.

Il punteggio nel programma si basa sulla quantità di messaggi di chat inviati dagli utenti virtuali, ogni messaggio vale tre punti. Abbiamo due finali in questo gioco. In quello negativo, allo spettatore è semplicemente detto che l'interesse dell'host della chat virtuale è sceso a zero. L'altro finale, notevolmente migliore, garantirà allo spettatore un bonus del valore di 500 punti:

```
    punteggio = messaggi * 3 # Ogni messaggio in chat vale 3 punti

    if interesse < 1: # In questo caso semplicemente finirà il gioco
clrprint("Il tuo interesse è sceso a 0/100!", clr='yellow'
    else: # In questo caso invece avremo preso il bonus
clrprint("Il tuo interesse ha raggiunto 100/100!", clr='red')
clrprint("Ottieni un bonus di 500 punti!", clr='giallo')
    punteggio = punteggio + 500

    tempogioco = orario1 - orario # Calcoliamo il tempo trascorso
print("Tempo trascorso nella chat: " +str(tempogioco))
print("Hai inviato " +str(messaggi)+ " messaggi")
print("PUNTEGGIO: " +str(punteggio))

    # Qui controlleremo e memorizzeremo il record del punteggio
    with open("punteggi.txt", "r+") as filepunti:
record = filepunti.read()
if not record: # Se il file è vuoto inseriamo uno
record = '1'
if punteggio >int(record):
print("Nuovo RECORD!")
filepunti.seek(1)
filepunti.write(punteggio) # Scriviamo il record
else:
print("Record: %s" % record)
```

Siamo giunti alla conclusione, speriamo che quest'applicazione abbia instillato l'interesse e l'ispirazione per il prosieguo dello studio con il fantastico Python, senza tralasciare gli altri tre linguaggi ovviamente!

La sequenza di Fibonacci in C#

Passiamo adesso al linguaggio della Microsoft, altrimenti rischiamo un'indigestione di Python.

Nel prossimo listato, ci occuperemo della sequenza di Fibonacci, dove, per chi non la conoscesse, una sequenza è generata dalla somma dei due numeri precedenti, quindi, partendo da zero avremo 0, 1, 1, 2, 3, 5, 8 e così via (0+1 = 1, 1+1= 2, 2+1= 3, 3+2=5, 5+3=8):

```
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;

    // Definiamo un ambito specifico per gestire i dati
    namespace ConsoleApp9
    {
  class Prova
  {
    public static int Fibonacci(int x)
    {
      int a = 0;
      int b = 1;
      for (int i = 0; i < x; i++)
      {
        int temp = a;
        a = b;
        b = temp + b;
        }
      return a;
    }

    static void Main()
    {
    Console.Write("Quanti numeri della sequenza vuoi?: ");
         int lunghezza = Convert.ToInt32(Console.ReadLine());
         Console.Write("\nLa sequenza di Fibonacci con {0} numeri :
", lunghezza);
```

```
          for (int i = 0; i < lunghezza; i++)
          {
           Console.Write("{0}  ", Fibonacci(i));
          }
          Console.ReadKey();
        }
   }

    }
```

Ed ecco invece il secondo esempio con una modalità differente:

```
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;

    namespace test
{
   class Prova
   {
       static long[] num;

       static long Fib(int n)
       {
           if (0 == num[n])
           {
               num[n] = Fib(n - 1) + Fib(n - 2);
           }
           return num[n];
       }

       static void Main()
       {
             Console.Write("Inserisci quale numero della sequenza di
Fibonacci vuoi vedere = ");
           int n = int.Parse(Console.ReadLine());
```

```
            num = new long[n + 2];
            num[1] = 1;
            num[2] = 1;
            long result = Fib(n);
               Console.WriteLine("Numero della sequenza scelto({0}) =
{1}", n, result);
            Console.ReadKey();
        }
    }
}
```

Adesso ripasseremo il threading semplice e l'accesso basato su thread a dati condivisi/variabili, l'utilizzo di oggetti di blocco e il metodo *Interlocked.Decrement()* utile a diminuire il valore di una variabile e poi memorizzarne il risultato:

```
    using System;
    using System.Threading;

    public class Esempio
    {
  const int GIU = 0;
  const int SU = 101;

  static Object blocco = new Object();
  static Random casuale = new Random();
  static CountdownEvent controllo;
  static int totNum = 0;
  static int totPun = 0;
  static int contaMP = 1000;

  public static void Main()
  {
    // Eseguiamo i tre thread
 controllo = new CountdownEvent(1);
    for (int ctr = 0; ctr <= 2; ctr++) {
      controllo.AddCount();
       Thread th = new Thread(GenerateNumbers);
```

```
        th.Name = "Thread" + ctr.ToString();
        th.Start();
     }
     controllo.Signal();
     controllo.Wait();
     Console.WriteLine();
           Console.WriteLine("Totale  valori  di  punto  intermedio:
 {0,10:N0} ({1:P3})", totPun, totPun/((double)totNum));
     Console.WriteLine("Totale numero casuali: {0,10:N0}", totNum);
  }

  private static void GenerateNumbers()
  {
     int midpoint = (SU - GIU) / 2;
     int value = 0;
     int total = 0;
     int midpt = 0;

    do {
        lock(blocco) {
           value = casuale.Next(GIU, SU);
        }
        if (value == midpoint) {
           Interlocked.Decrement(ref contaMP);
           midpt++;
        }
        total++;
     } while (Volatile.Read(ref contaMP) > 0);
     Interlocked.Add(ref totNum, total);
     Interlocked.Add(ref totPun, midpt);
                  string   s   =   String.Format("Thread   {0}:\n",
Thread.CurrentThread.Name) +
          String.Format("Numeri casuali: {0:N0}\n", total) +
                 String.Format("Valori  di  punto  intermedio:  {0:N0}
({1:P3})", midpt,((double) midpt)/total);
    Console.WriteLine(s);
    controllo.Signal();
```

```
  }
}
```

L'esempio mostrato determina la quantità di numeri casuali compresi tra 0 e 100 per poi generare altri 100 numeri casuali con un valore di punto medio (*midpoint*). Per tenere traccia di questi numeri abbiamo definito una variabile, *contaMP*, impostata su 1.000 che decrementerà ogni volta che il generatore di numeri casuali restituirà un valore di punto intermedio. Poiché tre thread genereranno i numeri casuali, il metodo *Decrement()* è chiamato per garantire che più thread non siano aggiornati da *contaMP* contemporaneamente. Si noti che è usato anche un blocco per proteggere il generatore di numeri casuali e che è usato un oggetto *CountdownEvent* per assicurarsi che il metodo *Main* non finisca l'esecuzione prima che lo facciano i tre thread addetti alle operazioni sui numeri.

Operazioni sui file e threading con C#

Nel paragrafo precedente ci siamo concentrati su alcune operazioni matematiche e sui thread, in questo invece ripasseremo e approfondiremo alcune importanti operazioni inerenti al flusso del programma e i cicli, la gestione dei file, la formattazione delle stringhe e l'interazione dell'utente. Quest'applicazione che andremo a osservare è basata sul classico e basico schema del quiz. Necessiterà quindi di due file separati, *domande.txt* e *risposte.txt*. Entrambi saranno elaborati dall'alto verso il basso con ogni riga nel file delle domande corrispondente alla stessa riga nel file delle risposte. Quest'approccio semplifica l'aggiunta di nuove domande e il cambio di quelle esistenti semplicemente modificando due file di testo.
L'utente dovrà digitare la propria risposta durante l'esecuzione del programma. Un oggetto *thread* è eseguito in background, indipendentemente dal quiz in corso, inducendo l'utente a sbrigarsi riproponendo a intervalli di cinque secondi la frase: "*Veloce! Non abbiamo tutto il giorno!"*. Ogni risposta corretta garantisce tre punti. Alla fine del quiz, all'utente è presentato il punteggio e la percentuale di risposte corrette. Iniziamo subito:

```
   using System;
using System.IO;
   using System.Threading;
   class Domande
```

```
{
  public static void Main()
  {

  int contatore=0, punteggio=0, percentuale=0;
   bool cronometro=true;
   Console.WriteLine("Benvenuto nel Quiz di Mario!\n");

  Thread ansia = new Thread(delegate() {
   Thread.Sleep(6500);
   while(true) {
      Console.WriteLine("Veloce! non abbiamo tutto il giorno!");
      Thread.Sleep(4900);
     if(!cronometro) break;
     }
  });

    var d = File.ReadAllLines("domande.txt");
var r = File.ReadAllLines("risposte.txt");
ansia.Start();

    foreach(string lines in d)
{
  Console.WriteLine("{0}\nInserisci risposta: ", lines);
  string risposta = Console.ReadLine().ToUpper();
  ++contatore;

    if(risposta.Equals(r[contatore-1])) {
Console.WriteLine("{0} è giusto. ", risposta); punteggio+=5;

   }else  if(risposta!="")  Console.WriteLine("{0}  è  sbagliato.  ",
risposta);
  else Console.WriteLine("Digita del testo per giocare!");
}
cronometro = false;
Console.Write("\nIl tuo punteggio: {0}. ", punteggio);
percentuale = punteggio*50/(contatore*2,5);
```

```
Console.WriteLine("Hai una percentuale del {0}% sulle {1} domande
presentate.", percentuale, contatore);

   if(percentuale < 25) Console.WriteLine("Fai un po' schifo.");
if(percentuale >= 45) Console.WriteLine("Ottimo risultato!");
}
}
```

Siamo partiti definendo tre variabili intere (*contatore, punteggio* e *percentuale*), una booleana (*cronometro*) e un thread che abbiamo chiamato *ansia*, il quale servirà proprio allo scopo di mettere un po' di fretta all'utente. Il thread continuerà a mettere ansia finché la variabile *cronometro* non sarà impostata su *false*. Dopo questo ciclo il programma aprirà i file di testo inerenti alle domande e le risposte che dovremo avere pronti nella cartella del progetto. Li potremo creare semplicemente con il blocco note.
In questo listato abbiamo anche utilizzato un tipo di variabile peculiare, cioè *var*. Questa è una definizione implicita in C# e significa semplicemente che il programma del compilatore può determinare il tipo di una variabile. Per i progetti più piccoli quest'approccio in genere funzionerà correttamente.
Una volta acquisiti questi file, convertiremo il testo della nostra risposta alla domanda in maiuscolo per ovviare a eventuali problemi di compatibilità del browser e lo confronteremo con le risposte nel file di testo, aggiungendo un punteggio in caso sia corretta. Adopereremo *else if* per verificare se l'utente ha semplicemente sbagliato o ha premuto invio senza scrivere nulla.
A questo punto la variabile *cronometro* sarà bloccata per uscire dal ciclo e si calcolerà il punteggio finale.

L'accesso e la gestione dei file in Java

Torniamo al nostro Java per osservare un listato focalizzato sull'accesso e sullo sfruttamento del contenuto di file contenenti testo. In pratica visualizzeremo delle liste con promettenti calciatori, inseriti in file che dovremo ovviamente preoccuparci di fornire nella cartella del progetto:

```
    import java.io.File;
    import java.io.FileNotFoundException;
import java.util.Scanner;

    public class MiglioriCalciatori{
    static void Carica(String percorso, String nome) {
    try{
   File oggetto = new File(percorso + nome);
   Scanner risposta = new Scanner(oggetto);
   while (risposta.hasNextLine()) {
      String testo = risposta.nextLine();
       System.out.println(testo);
      }
   risposta.close();
}catch(FileNotFoundException e) {
   System.out.println("File" + nome + "non trovato!");
   }
    }

    static void Scelta() {
   System.out.println("Migliori calciatori U21\nScegli: ");
   System.out.println("[1] Portieri");
   System.out.println("[2] Difensori");
   System.out.println("[3] Centrocampisti");
 System.out.println("[4] Attacanti");
 System.out.println("[5] Esci)");
}

    public static void main(String[] args) {

   // Visualizzamo il menu principale
   Scelta();
```

```
  Scanner leggi = new Scanner(System.in);
  while(true) {
  String scelta = leggi.nextLine();


  if(scelta.equals("1")) Carica("","portieri.txt");
  if(scelta.equals("2")) Carica("","difensori.txt");
  if(scelta.equals("3")) Carica("","centrocampisti.txt");
 if(scelta.equals("4")) Carica("","attaccanti.txt");
 if(scelta.equals("5")) break;


  // Ritorniamo al menu se l'utente preme invio
  if(scelta.equals("")) Scelta();

  } System.out.println("Uscita dal prgramma eseguita.");
}
   }
```

Abbiamo appena visto che inizialmente creiamo un metodo, *Carica*, che accetta due argomenti, cioè il percorso e il nome del file. Si procede quindi al caricamento del file di testo specificato visualizzando il contenuto, riga per riga, sullo schermo. Abbiamo anche un altro metodo, *Scelta*, che è semplicemente utilizzato per visualizzare le opzioni disponibili per l'utente. Entrambi i metodi sono definiti *void*, il che significa che devono restituire un'informazione.

C'è un blocco *try catch,* dove avremo la gestione della scelta del menù principale attraverso l'oggetto Scanner *risposta*, e che genererà un'eccezione, se non saranno trovati i file di testo con i calciatori. Un ciclo *while* infinito mantiene il programma in esecuzione fino a quando l'utente non inserisce "5" sulla tastiera, nel qual caso sarà eseguita la parola chiave *break* che porterà a termine il listato.

Un cronometro per Java

Il programma che andremo a vedere tra poco funge da semplice cronometro con una visualizzazione grafica basica. Questo programma consiste di un unico file principale. Vediamo il listato:

```
 import java.awt.*;
 import java.awt.event.*;
 import javax.swing.*;

 public class Cronometro
 {
private JFrame frame;
private JLabel labelTime;
private JPanel panelButtons;
private JButton buttonStart;
private JButton buttonStop;
private Timer timer;
private long startTime;

 public Cronometro()
 {
frame = new JFrame("Cronometro");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize (220, 140);

labelTime = new JLabel("0:00:00.0");
labelTime.setFont(new Font ("SansSerif", Font.BOLD, 30));
labelTime.setHorizontalAlignment(JLabel.CENTER);

buttonStart = new JButton("START");
buttonStop = new JButton("STOP");
buttonStop.setEnabled(false);

panelButtons = new JPanel(new GridLayout (1, 2));
panelButtons.add(buttonStart);
panelButtons.add(buttonStop);

frame.add(labelTime, BorderLayout.CENTER);
```

```
  frame.add(panelButtons, BorderLayout.SOUTH);

  timer = new Timer (50, new ActionListener () {
  public void actionPerformed (ActionEvent e) {
  long diffTime = System.currentTimeMillis() - startTime;

  int decimi = (int) (diffTime % 1000 / 100);
  int secondi = (int) (diffTime / 1000 % 60);
  int minuti = (int) (diffTime / 60000 % 60);
  int ore = (int) (diffTime / 3600000);

  String s = String.format("%d:%02d:%02d.%d", ore, minuti,
   secondi, decimi);

  labelTime.setText (s);
}
   });

   buttonStart.addActionListener(new ActionListener () {
   public void actionPerformed(ActionEvent e) {
  startTime = System.currentTimeMillis ();
  timer.start ();
  buttonStart.setEnabled(false);
  buttonStop.setEnabled(true);
}
});

   buttonStop.addActionListener(new ActionListener () {
public void actionPerformed(ActionEvent e) {
  timer.stop();
  buttonStart.setEnabled(true);
  buttonStop.setEnabled(false);
}
});

   frame.setVisible(true);
}
```

```
    public static void main(String[] args)
{
  SwingUtilities.invokeLater(new Runnable () {
  public void run() {
  new Cronometro();
  }
});
}
}
```

E con questo siamo giunti alla fine dell'ultimo listato del libro, dove abbiamo avuto a che fare con un po' di grafica a livello molto basico e anche con ascoltatori di eventi legati al click sui pulsanti del cronometro. Tutte funzionalità che ci apriranno un mondo sulla programmazione di applicazioni desktop. Mi auguro, dunque, che questo testo abbia contribuito a creare un nuovo programmatore.

Quiz & esercizi

1) Creare un listato PHP per importare una fattura elettronica all'interno di una pagina Web.
2) In PHP posso inviare file con l'istruzione cURL?
3) In Python potrò assegnare un valore a una variabile e a un array nella stessa riga?
4) Correggere l'errore nella riga di codice Python:

```
x = 22 = true = [] = basso = int = y
```

5) Correggere gli errori e le mancanze nel listato Python:

```
    import time
name = "Prova"
a = 10
b = 0
orario = datetime.datetime.now()
testo = random.choice([0, 1, 2])
print(random, ora, a, b)
```

6) Nel linguaggio C# esiste un metodo per diminuire il valore di una variabile e allo stesso tempo memorizzarne il risultato?
7) Che differenza c'è tra le variabili e le costanti?
8) In Java, quando imposteremo un metodo *void*, cosa dovremmo ottenere?
9) E invece, sempre in Java, impostando un metodo *static*, cosa accadrà?
10) Rileggi e primi esercizi e controlla quanto sei migliorato!

Riassunto

Alla fine di quest'ultimo capitolo, avremo rinfrescato la memoria sui seguenti aspetti di PHP, Python, C# e Java: le strutture di dati, il flusso del programma e le operazioni di base sui file, le basi sulle tecniche di threading, la gestione delle eccezioni tramite il blocco *try catch*, e molto altro. Spero vivamente che questi capitoli siano stati interessanti ma soprattutto utili.

Ti ringrazio per aver acquistato questo libro, spero sia stato utile e ti sia piaciuto.

Se vorrai lasciare una recensione su Amazon sarà molto gradita, grazie mille, ti voglio bene.

Tony Chan.

Bibliografia

I sottoelencati siti internet hanno fornito ispirazione e strumenti per diversi argomenti trattati nel libro:

https://it.wikipedia.org

https://www.codingcreativo.it/

https://www.html.it/

https://www.w3schools.com/

https://stackoverflow.com/

http://www.w3bai.com/it

http://www.codewigs.com/

http://www.xml.com/

http://www.luigisabbetti.it/

http://www.html5italia.com/

https://giordanoblog.altervista.org/

https://www.andreaminini.com/

http://gallocchiolorenzo.altervista.org/

https://www.marcoalbasini.com/

https://pythonitalia.github.io/

https://www.yocker.com

https://onlinephp.io/

https://www.programiz.com/csharp-programming/online-compiler/

https://www.programiz.com/python-programming/online-compiler/

https://www.jdoodle.com/online-java-compiler/

E anche i testi: ***Javascript, la guida definitiva*** *di Tony Chan*
HTML5 & CSS3, la guida completa *di Tony Chan*