lezione 9

Un confronto con la programmazione imperativa

Dal record alla classe

Nel passaggio dalla programmazione imperativa a quella a oggetti possiamo affermare che il concetto di **classe** può essere visto come un'estensione del concetto di **record**. In effetti, pur con le dovute cautele, una classe è un record con in più i metodi. Ovviamente, i metodi di una classe possono non essere presenti e i due concetti in questo caso si avvicinano ulteriormente. Ad esempio, in questa unità abbiamo definito la classe *Studente*. Se vogliamo definire uno studente, possiamo anche farlo con un record. Di seguito sono riportate entrambe le definizioni.

Programmazione imperativa	Programmazione a oggetti

TIPO Studente = RECORD
  Nome: STRINGA
  Cognome: STRINGA
  Voti : ARRAY(8) di INTERO
  Materie: ARRAY(8) di STRINGA
FINERECORD
	CLASSE Studente
	  Nome: STRINGA
	  Cognome: STRINGA
	  Voti: ARRAY(8) di INTERO
	  Materie: ARRAY(8) di STRINGA
	  ...
	FINE // Classe

Se vogliamo, ad esempio, creare lo studente *Stud1* di cognome "Rossi" e nome "Paolo" con i suoi voti nelle rispettive materie, scriveremo:

Programmazione imperativa	Programmazione a oggetti
Stud1: Studente Stud1.Cognome ← "Rossi" Stud1.Nome ← "Paolo"	Stud1: Studente Stud1 ← NUOVO Studente() Stud1.Cognome ← "Rossi" Stud1.Nome ← "Paolo"

Fin qui è tutto molto simile, ma le differenze esistono. Una prima differenza è il metodo **costruttore** che consente di inizializzare i valori del nome, del cognome, delle materie e dei voti di un nuovo studente e che non ha un analogo nella programmazione imperativa. Possiamo, infatti, scrivere il costruttore della classe *Studente* come:

```
Studente(pNome: STRINGA, pCognome: STRINGA)
INIZIO
    Materie(1) ← "Italiano"
    Materie(2) ← "Storia"
    ...
    Materie(8) ← "Calcolo"
    PER I ← 1 A 8 ESEGUI
        Voti(I) = 6
    FINEPER
    Nome ← pNome
    Cognome ← pCognome
FINE
```

e inizializzare un nuovo studente durante la sua creazione nel seguente modo:

```
Stud1 ← NUOVO Studente("Paolo", "Rossi")
```

Questo tipo di istruzione non è presente nella programmazione imperativa.
La funzione *setVoti()* nel codice con paradigma imperativo risulta visibile e utilizzabile da chiunque. Nella classe *Studente*, invece, come vedremo nella prossima unità, il metodo *setVoti()* può essere utilizzato solo all'interno della classe in cui è stato definito e può essere reso "invisibile" all'esterno (*information hiding*).

Dalla funzione al metodo

La differenza tra il paradigma imperativo e quello a oggetti non è solo formale o sintattica.

Consideriamo l'esempio del paragrafo precedente. Se vogliamo modificare i voti di uno studente, dobbiamo scrivere la funzione *setVoti()* per la programmazione imperativa e il metodo *setVoti()* per la programmazione a oggetti, come mostrato di seguito.

Programmazione imperativa	Programmazione a oggetti

TIPO Studente = RECORD
  Nome: STRINGA
  Cognome: STRINGA
  Voti: ARRAY(8) di INTERO
  Materie: ARRAY(8) di STRINGA
FINERECORD
 	
FUNZIONE setVoti(REF S: Studente)
INIZIO
  PER I ← 1 A 8 ESEGUI
  SCRIVI("Inserisci voto ",S.Materie(I),": ")
  LEGGI(S.Voti(I))
  FINEPER
FINE
	CLASSE Studente
	  Nome: STRINGA
	  Cognome: STRINGA
	  Voti: ARRAY(8) di INTERO
	  Materie: ARRAY(8) di STRINGA
	  ...
	  setVoti()
	  INIZIO
	  PER I ← 1 A 8 ESEGUI
	  SCRIVI("Inserisci voto ",Materie(I),": ")
	  LEGGI(Voti(I))
	  FINEPER
	  FINE
	  ...
	FINE // Classe

Benché siano formalmente molto simili, il metodo *setVoti()* fa parte della definizione della classe e, pertanto, è inserito al suo interno (incapsulamento), mentre la funzione *setVoti()* è esterna e scollegata dai dati su cui agisce (lo studente). Lo studente a cui bisogna assegnare i voti viene, infatti, passato come parametro.

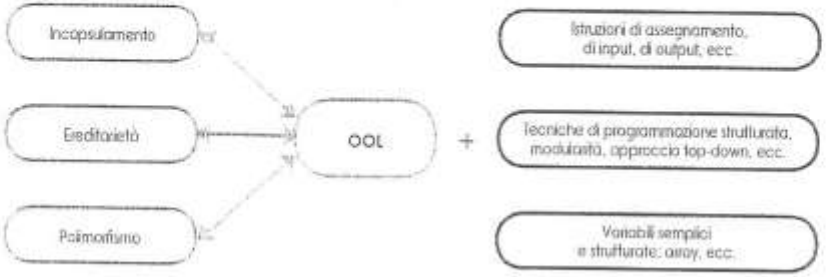
lezione 10

OOP come evoluzione della programmazione imperativa

Modularità, un concetto da recuperare pienamente

Come abbiamo accennato nella precedente unità, per programmare a oggetti non occorre ricominciare da zero. L'OOP è un'**evoluzione** che spesso integra le buone pratiche di programmazione apprese dal paradigma imperativo con alcuni concetti chiave come incapsulamento, polimorfismo, ereditarietà. Vanno mantenuti, quindi, i concetti di:

- variabili, costanti, tipi;
- istruzioni di assegnamento;
- istruzioni di input, di output,
- tecniche di programmazione strutturata: costrutto di selezione, iterazione;
- variabili strutturate: array,
- concetti chiave, come quello di **modularità**.



Un buon programmatore deve fare attenzione alla complessità di un sistema software. Per risolvere un problema complesso è bene suddividerlo in problemi più semplici e risolvere i singoli problemi. Questa strategia è alla base della modularità.

→ Il concetto di **modularità** consiste nello strutturare l'applicazione software in componenti (o **moduli**) il più possibile indipendenti ma cooperanti tra di loro. In questo modo tali componenti possono essere **riutilizzati** in altre applicazioni.

Il concetto di modularità è alla base della programmazione a oggetti.

Ogni mattoncino può essere pensato come un modulo, cioè una parte indipendente ma cooperante

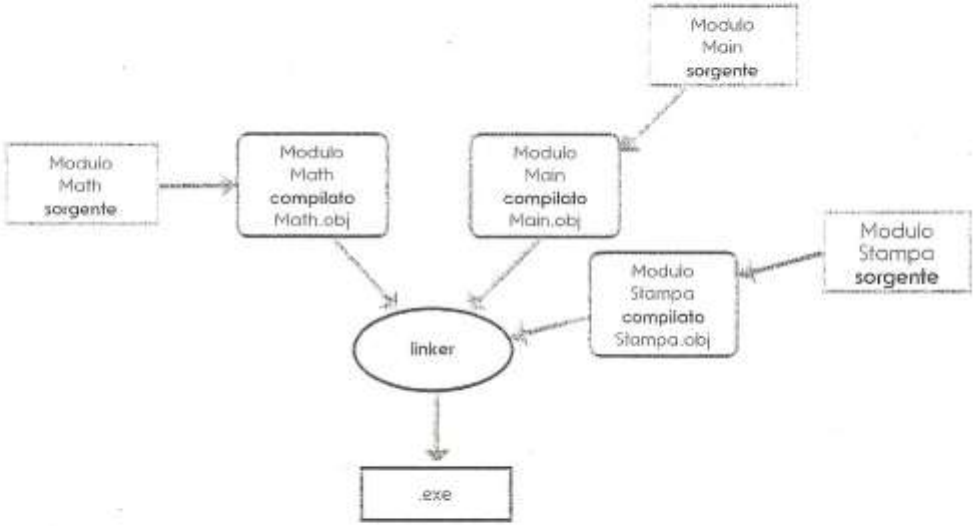
stop

In questo contesto useremo come sinonimi il concetto di componente e quello di modulo.

Nella programmazione imperativa la modularità è strettamente legata all'approccio **top-down**, cioè alla possibilità di strutturare un programma in sottoprogrammi (procedure e funzioni). La strutturazione dell'applicazione in componenti indipendenti ma cooperanti offre numerosi **vantaggi**, tra i quali:

- **riduzione della complessità**: si risolvono problemi più semplici invece di uno solo più complesso.
- **riutilizzo del codice**: nei propri programmi o anche negli altri (rivendibilità del codice). La funzione di un programmatore, ad esempio, potrebbe essere acquistata da altri programmatori. Questi ultimi potrebbero non trovare conveniente sviluppare la funzione da zero per motivi di costi di sviluppo più elevati.
- **individuazione e correzione degli errori** (logici): è più facile seguire la logica nei singoli sottoprogrammi, poiché sono formati da molte meno righe di codice rispetto a un unico grosso programma.
- **leggibilità del codice**: è più facile scorrere le poche righe di codice di un sottoprogramma.
- **manutenzione del codice**: è più facile apportare modifiche alle poche righe di un sottoprogramma.
- **lavoro in team**: ogni programmatore può autonomamente sviluppare un sottoprogramma e poi assemblare il lavoro di tutti per ottenere l'intero programma.

Un modulo raggruppa al suo interno dati e funzioni in una singola unità sintattica. Generalmente viene compilato e reso eseguibile in modo da rendere ancor più inaccessibile il suo interno.



Purtroppo esiste un grosso inconveniente nell'utilizzo dei moduli. Un modulo **non ha dati propri**, ma applica la propria funzionalità a dati che risultano alla fine **globali** per il programma chiamante e, quindi, di uso "pubblico" secondo lo standard funzionale. Inoltre, utilizzando i moduli è difficile descrivere **oggetti troppo complessi** del mondo esterno. Infine, è ancora difficile poter **riutilizzare** il codice.
L'OOP ha superato questi limiti attraverso i concetti di interfacce, information hiding e incapsulamento.

lezione 14

Vantaggi dell'incapsulamento

Per spiegare quali sono i vantaggi dell'incapsulamento, facciamo un parallelo tra programmazione imperativa e programmazione a oggetti. Consideriamo una motocicletta. Nella programmazione imperativa, per descrivere una motocicletta il programmatore inizia a dichiarare le variabili che rappresentano le sue caratteristiche: una marca, un modello, una cilindrata e così via. Poi realizza le funzioni che descrivono il comportamento di una motocicletta: *AvviaMotore*( ), *FaiRifornimento*( ) e così via. Quando necessita di un comportamento per una motocicletta, invoca la funzione relativa a quel comportamento, passandogli come parametro la motocicletta su cui deve agire. Ad esempio, *AvviaMotore*(MotoDiLuca).

```
...

Marca: STRINGA
Colore: STRINGA
Cilindrata: INTERO
StatoMotore: BOOLEANO
Serbatoio: INTERO

PROCEDURA AvviaMotore(M: Motocicletta)
INIZIO
   ...
FINE

FUNZIONE Rifornimento(M: Motocicletta,
                      B: INTERO): INTERO
INIZIO
   ...
FINE

PROCEDURA MostraStato(M:Motocicletta)
INIZIO
   ...
FINE
```

```
CLASSE Motocicletta
INIZIO
   Marca: STRINGA
   Colore: STRINGA
   Cilindrata: INTERO
   StatoMotore: BOOLEANO
   Serbatoio: INTERO

   Motocicletta(pMarca: STRINGA, pColore: STRINGA,
                pCilindrata: INTERO)
   INIZIO
      Marca ← pMarca
      Colore ← pColore
      Cilindrata ← pCilindrata
      Serbatoio ← 0
      StatoMotore ← FALSO
   FINE

   AvviaMotore()
   INIZIO
      ...
   FINE

   Rifornimento(B: INTERO): INTERO
   INIZIO
      ...
   FINE

   MostraStato()
   INIZIO
      ...
   FINE
FINE   // Fine classe Motocicletta
```

I dati della motocicletta e i suoi comportamenti non si trovano insieme in una "unità sintattica-semantica". Chi ha esperienza di programmazione sa bene che, dopo un certo periodo di tempo di utilizzo del software, modificando, migliorando, manutenendo il codice, le variabili si confondono tra di loro e si aggiungono altre funzioni che nulla hanno a che vedere con la motocicletta, come possiamo vedere nello pseudocodice seguente.


```
...

Marca: STRINGA
Colore: STRINGA
Cilindrata: INTERO
UnaVariabile: INTERO
StatoMotore: BOOLEANO
UnaAltraVariabile: BOOLEANO
Serbatoio: INTERO

FUNZIONE AvviaMotore(M: Motocicletta)
INIZIO
   ...
FINE

PROCEDURA UnaNuovaProcedura()
INIZIO
   ...
FINE

FUNZIONE Rifornimento(M: Motocicletta,
                      B: INTERO): INTERO
INIZIO
   ...
FINE

PROCEDURA UnaAltraProcedura(X: INTERO)
INIZIO
   ...
FINE

PROCEDURA MostraStato(M:Motocicletta)
INIZIO
   ...
FINE
```

```
CLASSE Motocicletta
INIZIO
   Marca: STRINGA
   Colore: STRINGA
   Cilindrata: INTERO
   StatoMotore: BOOLEANO
   Serbatoio: INTERO

   Motocicletta(pMarca: STRINGA, pColore: STRINGA,
                pCilindrata: INTERO)
   INIZIO
      Marca ← pMarca
      Colore ← pColore
      Cilindrata ← pCilindrata
      Serbatoio ← 0
      StatoMotore ← FALSO
   FINE

   AvviaMotore()
   INIZIO
      ...
   FINE

   Rifornimento(B: INTERO): INTERO
   INIZIO
      ...
   FINE

   MostraStato()
   INIZIO
      ...
   FINE

FINE   // Fine classe Motocicletta
```

Col tempo diventa molto difficile distinguere ciò che ha a che fare con la motocicletta da tutto ciò che non ha nulla a che vedere con essa. Una conseguenza è che, oltre all'aspetto "visivo" di confusione, risulta molto più facile commettere errori durante i miglioramenti e le modifiche che si apporteranno al software e, quindi, sarà molto più difficile riutilizzare il software.
La definizione di una motocicletta con un linguaggio object-oriented permette invece di sfruttare la classe come contenitore che incapsula tutto ciò che serve. Non è quindi possibile utilizzare una funzione che non è della motocicletta o chiedere a una funzione di una motocicletta di lavorare con variabili che non sono di una motocicletta. Nella programmazione imperativa, invece, ciò è possibile.
Nella programmazione non a oggetti, il buon funzionamento del programma è quindi legato all'ordine, alla precisione, alla memoria, alla pulizia, al buon senso del programmatore.

Nella programmazione a oggetti è il compilatore che può eseguire buona parte dei controlli per mantenere coerente l'utilizzo del codice. Questo paradigma di programmazione, pertanto, è molto più ordinato e quindi più sicuro.

lezione 30

Un altro confronto con la programmazione imperativa

Esaminiamo i vantaggi dell'ereditarietà e dei concetti trattati in questa unità riconsiderando l'esempio della classe (e del record) *Studente* trattato alla fine della prima unità di questo blocco tematico.
Dopo aver definito le caratteristiche e i comportamenti di uno *studente*, supponiamo ora di dover definire le caratteristiche e i comportamenti di uno *studente lavoratore*.
Di seguito è riportato come potremo procedere nei due paradigmi di programmazione: imperativo e a oggetti.

Programmazione imperativa

```
TIPO Studente = RECORD
        Cognome: STRINGA(25)
        Nome: STRINGA(25)
        NumTelefono: STRINGA(20)
FINERECORD

TIPO StudenteLavoratore = RECORD
        St: Studente
        Stipendio: INTERO
        Ditta: STRINGA
FINERECORD

PROCEDURA ModificaTelefonoStud(REF S: Studente, VAL NuovoNumero: STRINGA(20))
INIZIO
    S.NumTelefono ← NuovoNumero
FINE

PROCEDURA ModificaTelefonoStudLav(REF SL: Studentelavoratore,
                                  VAL NuovoNumero: STRINGA(20))
INIZIO
    ModificaTelefonoStud(SL.St, NuovoNumero)
FINE

PROCEDURA ModificaStipendioStudLav(REF SL: Studentelavoratore,
                                   VAL NuovoStipendio: INTERO)
INIZIO
    SL.Stipendio ← NuovoStipendio
FINE
```


Programmazione a oggetti

```
CLASSE Studente
    Cognome: STRINGA(25)
    Nome: STRINGA(25)
    NumTelefono: STRINGA(20)

    Studente(pCognome: STRINGA(25), pNome:STRINGA(25),
                        pNumTel: STRINGA(20))
    INIZIO
        Nome ← pNome
        NumTelefono ← pNumTel
    FINE            // Costruttore Studente

    ModificaTelefono(NuovoNumero: INTERO)
    INIZIO
        NumTelefono ← NuovoTelefono
    FINE            // Metodo ModificaTelefono
FINE                // Classe Studente

CLASSE StudenteLavoratore EREDITA Studente
    Stipendio: INTERO
    Ditta: STRINGA
    StudenteLavoratore(pCognome: STRINGA(25), pNome:STRINGA(25),
          pNumTel: STRINGA(20), pStipendio:INTERO,pDitta: STRINGA)
    INIZIO
        SUPER(pCognome, pNome, pNumTel)
                    // Richiama esplicitamente il costruttore di Studente
        Nome ← pNome
        NumTelefono ← pNumTel
    FINE            // Costruttore StudenteLavoratore
    ModificaStipendio(NuovoStipendio: INTERO)
    INIZIO
        Stipendio ← NuovoStipendio
    FINE            // Metodo ModificaStipendio
FINE                // Classe StudenteLavoratore
```

Notiamo quanto segue.

- Nello pseudocodice a oggetti la classe *StudenteLavoratore*, poiché eredita dalla classe *Studente*, è composta solo dagli attributi e dai metodi specifici per la gestione di uno studente lavoratore (metodi estesi). Gli attributi *Nome*, *Cognome*, *NumTelefono* sono ereditati e pertanto non vanno riscritti. Il metodo *ModificaTelefono()* è ereditato e quindi non va riscritto.
- Nello pseudocodice imperativo, invece, dobbiamo riscrivere la funzione *ModificaTelefono()* anche per uno studente lavoratore. Dobbiamo, quindi, conservare due funzioni: *ModificaTelefonoStud()* e *ModificaTelefonoStudLav()* l'una per uno studente e l'altra per uno studente lavoratore.

Osservando i due pseudocodici risulta evidente:

- la maggior **fluidità** e **comprensione** della logica del paradigma a oggetti. Nel paradigma imperativo notiamo, invece, una evidente complicazione e intreccio di funzioni. Non si capisce con immediatezza su quali dati esse agiscano e che differenza esista tra funzioni apparentemente molto simili tra loro;
- l'elevata **riutilizzabilità** del codice a oggetti.