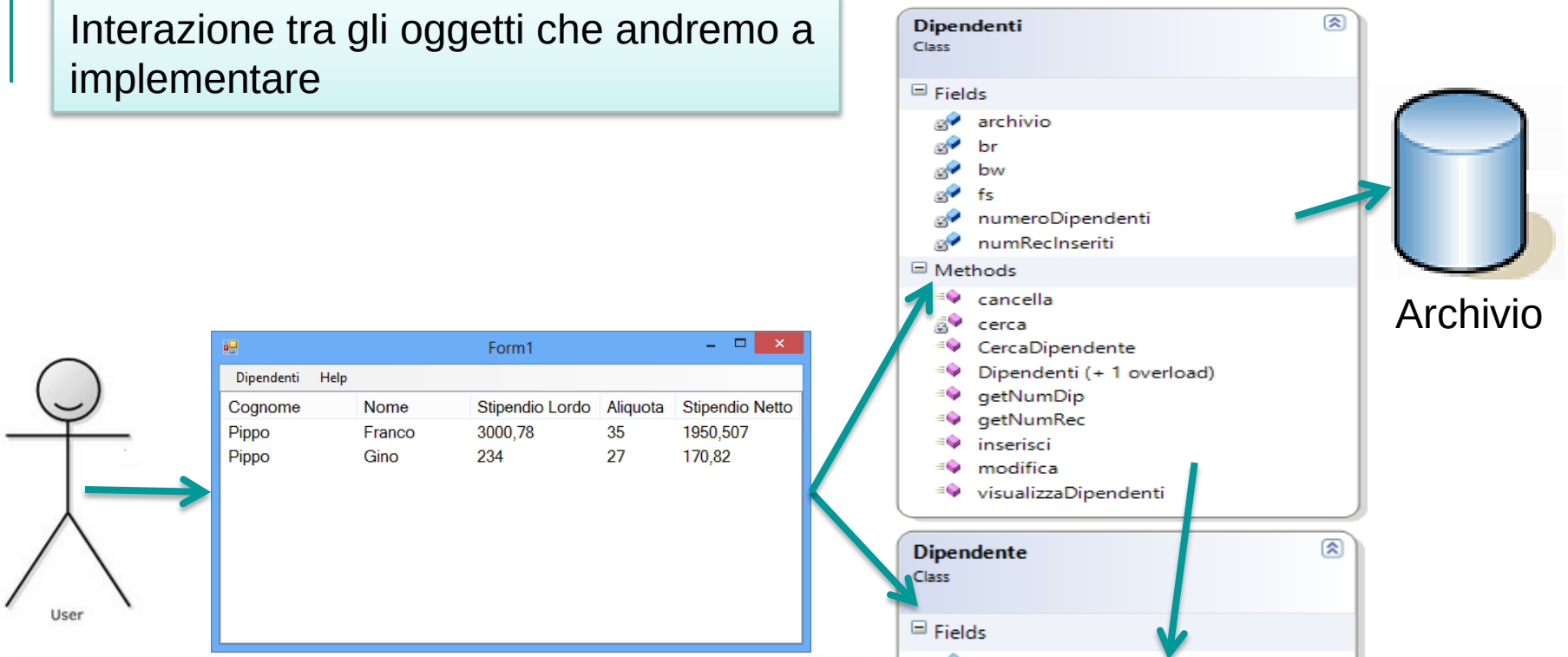
File ad accesso diretto funzioni Hash Parte1

Prof. Francesco Accarino
IIS Altiero Spinelli Sesto San Giovanni

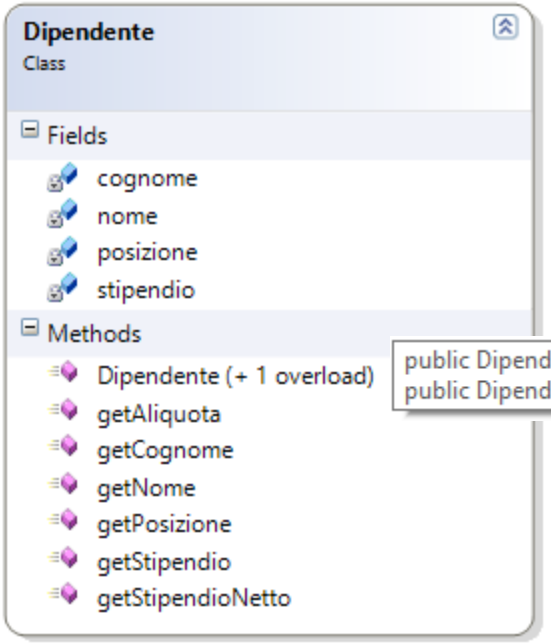
Esercitazione: Si vuole sviluppare una applicazione come quella mostrata nella figura in cui viene gestito un archivio di dipendenti contenete per ogni dipendente i campi mostrati. L'applicazioni permetterà di svolgere sull'archivio le operazioni mostrate nel menu della figura. Se l'archivio non è stato ancora creato nel menu sarà attivata la sola voce di menu Crea Archivio altrimenti saranno attivate tutte le altre

Interazione tra gli oggetti che andremo a implementare



Come si vede da questa figura l'utente interagisce con l'interfaccia grafica che a sua volta usa due classi. La prima Dipendente contiene gli attributi e i metodi per gestire le informazioni di un singolo dipendente. Mentre la classe Dipendenti gestisce tutte le operazioni sull'archivio

La classe Dipendente

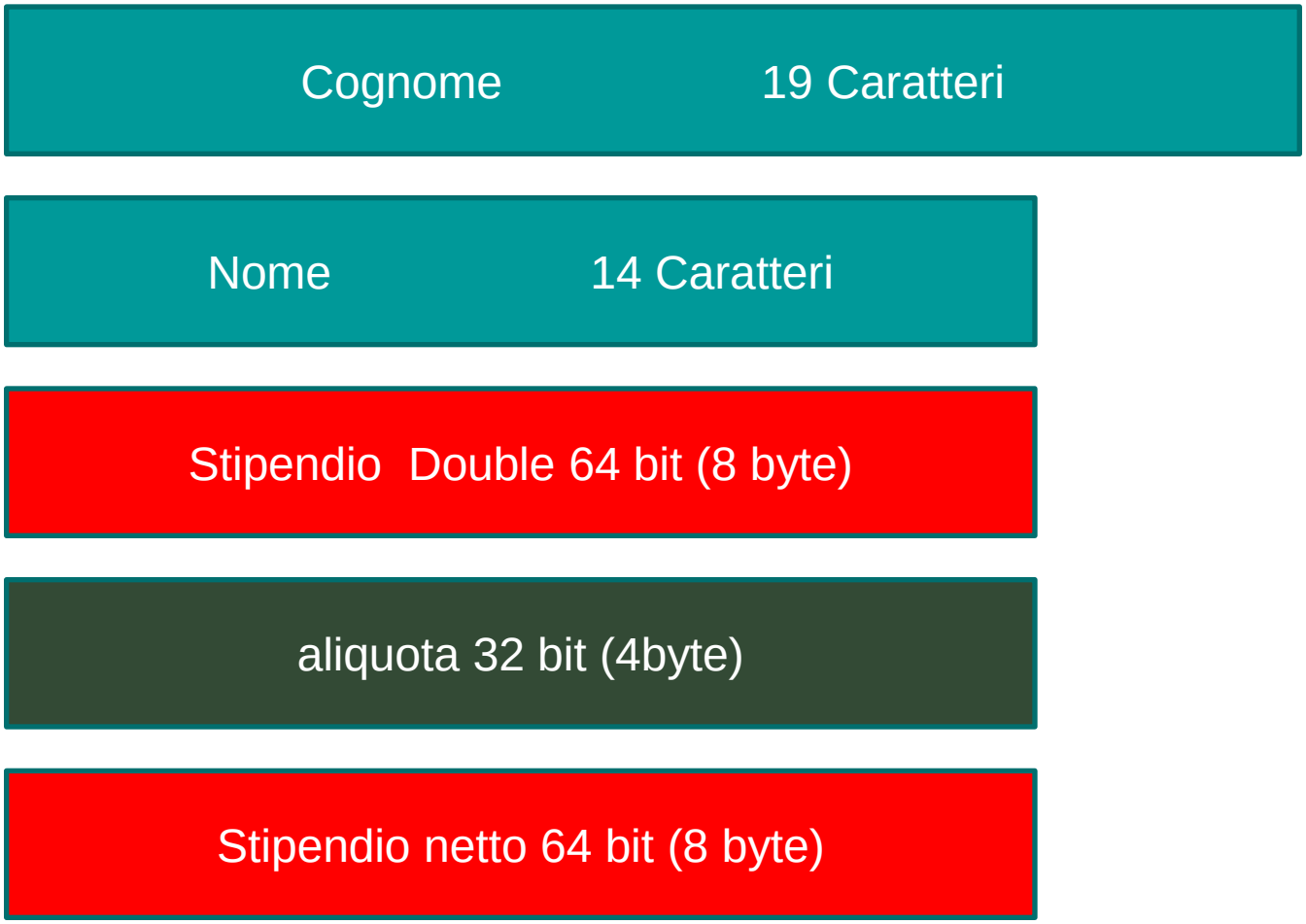


Come si vede dall'ADT della classe Dipendente osservando i suoi attributi e i due costruttori essa ha gli attributi Cognome nome e stipendio che caratterizzano un dipendente e un quarto attributo posizione che si riferisce alla posizione del record all'interno dell'archivio.

I campi aliquota e stipendio netto sono campi calcolati e come tali non andrebbero memorizzati nell'archivio ma calcolati appunto dai metodi dell'oggetto come mostrato nel frammento di codice seguente:

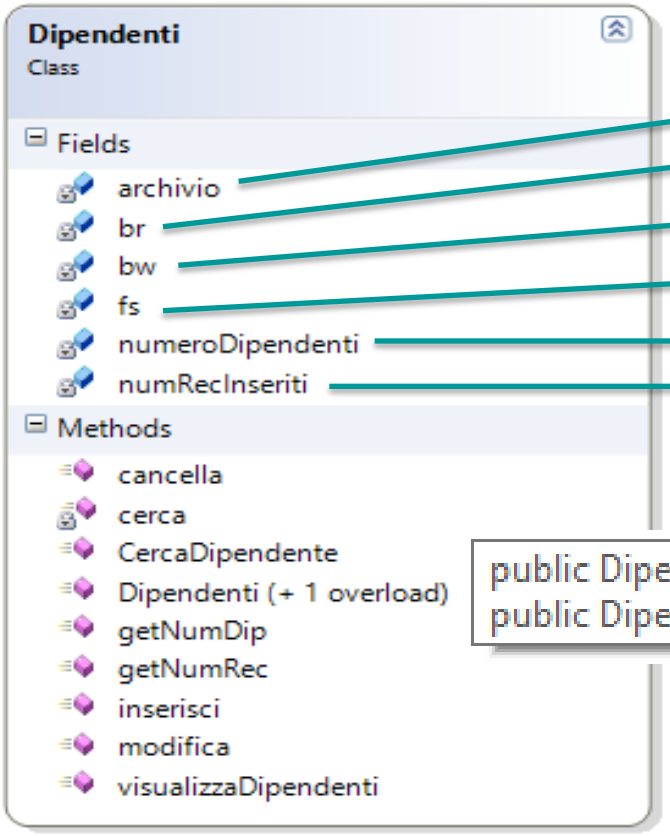
```
public int getAliquota()
{
    int a;
    if (stipendio*13 < 30000) a = 27;
    else
        if ((stipendio *13)> 30000.00 && (stipendio*13) < 50000.00) a = 35;
        else a = 40;
    return a;
}
public double getStipendioNetto()
{
    double imposta;
    imposta = (stipendio * getAliquota()) / 100;
    return(stipendio - imposta);
}
```

Noi invece adotteremo il seguente tracciato record memorizzando in esso anche i campi calcolati a puro scopo esercitativo



Quindi ogni registrazione nell'archivio occuperà 55 byte

La classe Dipendenti



Stringa contenente il nome del file

BinaryReader

BinaryWriter

FileStream

Numero dipendenti che l'archivio può contenere

Numero di record inseriti

```
public Dipendenti(long record)
public Dipendenti()
```

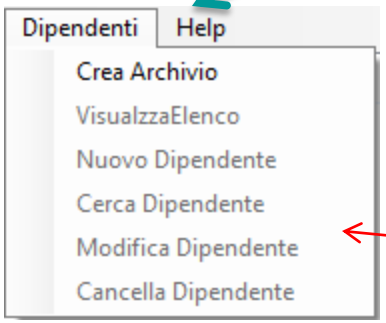
Utilizzato per creare l'archivio quando esso non è stato creato

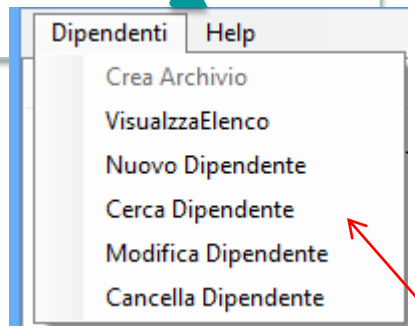
Utilizzato per aprire l'archivio quando esso già esiste

Infatti all'avvio l'applicazione, nel codice associato all'evento Load andremo a testare se l'archivio è già esistente, in questo caso sarà chiamato il costruttore senza parametri e si abiliteranno le altre voci di menu altrimenti si abilita solo la voce crea, ed in risposta al click su di essa si chiamerà il costruttore con numero di record

Il codice da inserire nel gestore dell'evento Load del form è il seguente. Mediante la proprietà getNumDip sappiamo se l'archivio esiste o deve essere ancora creato agendo di conseguenza sulle voci di menù

```
private void Form1_Load(object sender, EventArgs e)
{
    mioArchivio = new Dipendenti();

    if (mioArchivio.getNumDip()>0)
        enableMenuOperation();

    else

        disableMenuOPeration();
 }
```





La variabile mioArchivio è dichiarata a livello di classe nel Form1 e conterrà il reference all'oggetto dipendenti attraverso il quale svolgere le varie operazioni sull'archivio.

I due metodi enableMenuOperation e disbleMenuOperation li creeremo noi in Form1. Scrivendo tante: nomeVoce.Enabled = true; nomeVoce.Enabled = false; In base a cosa esideriamo

```
public class Dipendenti
 {
  long numeroDipendenti, numRecInseriti = 0;
  FileStream fs;
  BinaryReader br;
  BinaryWriter bw;
  string archivio ="dipendenti.dat";
```

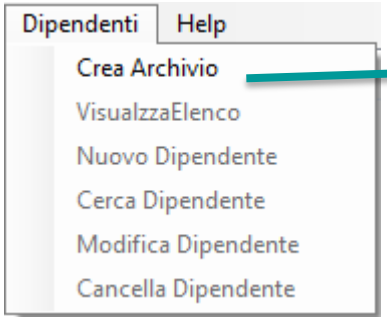
Un estratto della classe dipendenti in cui si vede come opera il costruttore della classe. Se il file esiste inizializza la variabile numero Dipendenti e conta in esso i numeri di record realmente inseriti per controllare in seguito gli inserimenti.

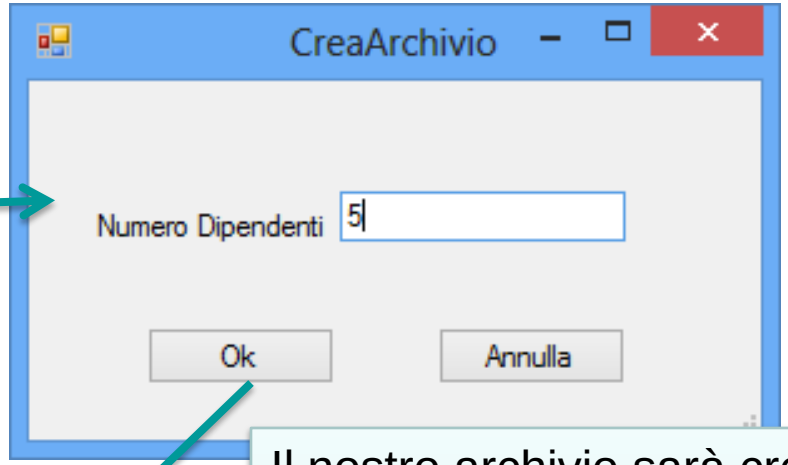
```
 public Dipendenti()
 {
     int i;
     if (File.Exists(archivio))
     {
         fs = File.Open(archivio, FileMode.Open);
         numeroDipendenti = fs.Length / 55;
         fs.Seek(0, SeekOrigin.Begin);
         for (i = 0; i < numeroDipendenti; i++)
         {
             if (Convert.ToByte(fs.ReadByte()) != '-')
                 numRecInseriti++;
             fs.Seek(54, SeekOrigin.Current);
         }
         fs.Close();
     }
 }
```

Osservando il codice si nota come se il file esiste si scorre l'archivio spostandosi di record in record semplicemente spostando il cursore di 55 byte in avanti.

A questo proposito è opportuno sottolineare che in fase di creazione dell'archivio esso viene riempito di trattini. Quindi se un record inizia con un carattere diverso da trattino contiene dei dati.

Ecco quindi cosa accadrà all'inizio.

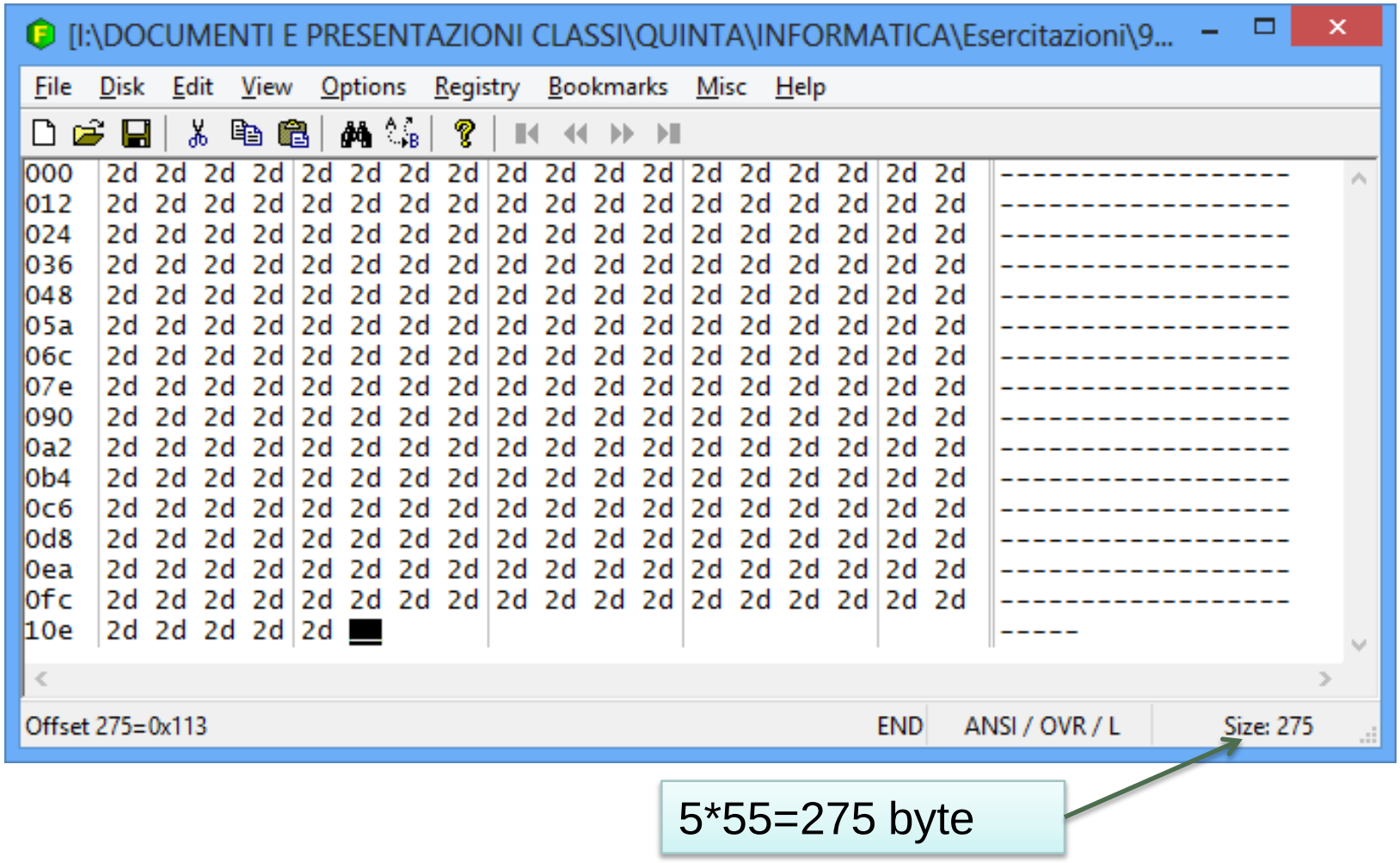





```
public class Dipendenti
 {
  long numeroDipendenti, numRecInseriti = 0;
  FileStream fs;
  BinaryReader br;
  BinaryWriter bw;
  string archivio ="dipendenti.dat";

 public Dipendenti(long record)
 {
     int i;
     fs = new FileStream(archivio, FileMode.Create, FileAccess.Write);
     numeroDipendenti = record;
     for (i = 0; i < record * 55; i++) fs.WriteByte(Convert.ToByte('-'));
     fs.Close();
 }
```

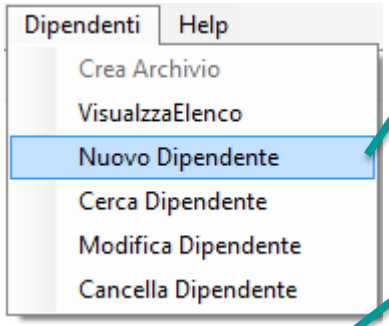
Il nostro archivio sarà creato inserendo 55 – per ogni record. Il carattere – lo utilizzeremo anche per cancellare logicamente un record e per verificare se un record è vuoto. Ovviamente un cognome inizia con un carattere diverso da –

Ed ecco come si presenta il nostro archivio appena creato quando è vuoto utilizzando il visualizzatore binario di file Fhred .

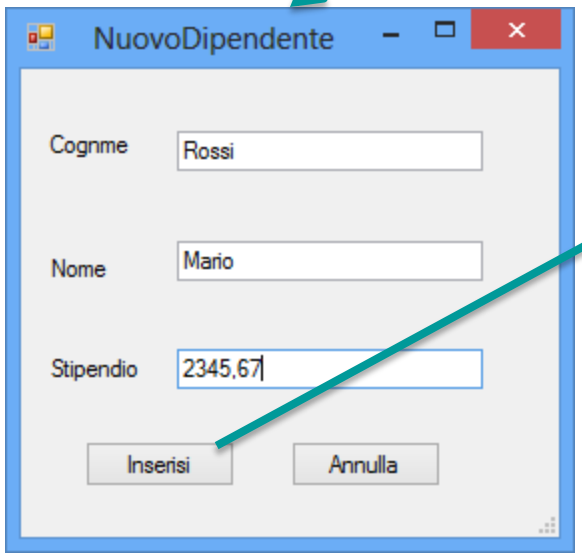


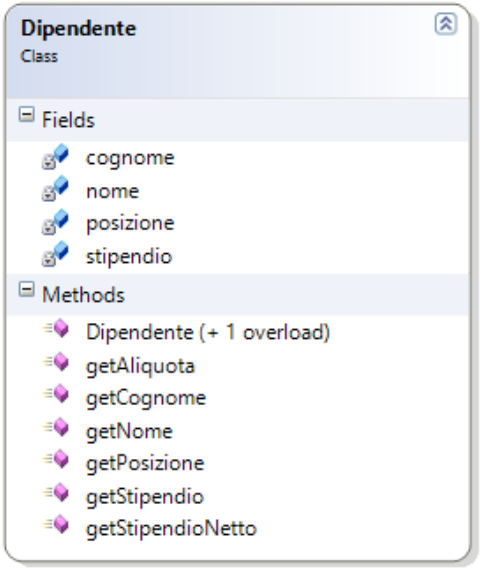
5*55=275 byte

Vediamo ora come implementare le altre voci di menu iniziando dalla funzione Nuovo Dipendente




```
private void nuovoToolStripMenuItem_Click(object sender, EventArgs e)
{
    NuovoDipendente nd;
    if (mioArchivio.getNumRec() < mioArchivio.getNumDip())
        nd = new NuovoDipendente(mioArchivio);
    else MessageBox.Show("L'archivio è Pieno!!", "errore",
                         MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

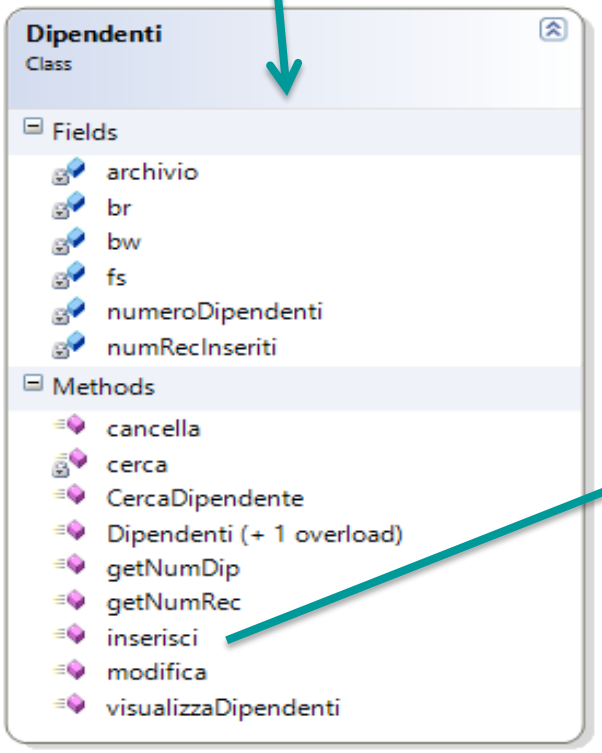




Creo un dipendente con i dati, calcolo mediante le funzioni di Hash la posizione relativa del record utilizzando il cognome inserito, invoco il metodo inserisci di mioArchivio passandogli la posizione e il dipendente

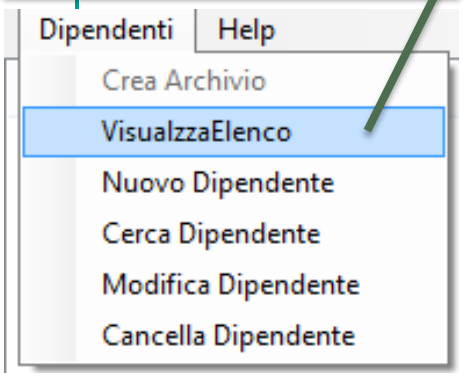
```
private void inserisciBtn_Click(object sender, EventArgs e)
{
    CharEnumerator car;//oggetto per enumerare  singoli caratteri di una stringa

    long hashCode,posizioneRelativa;//hashCode valorenumerico ottenuto come somma dei valori ascii dei singoli
                                    //caratteri del cognome da inserire
                                    //posizione relativa resto intero della divisione tra hashCode e numero di record

    if (cognome.Text == "" || nome.Text == "" || stipendio.Text == "")
    {
        MessageBox.Show("Devi Compilare tutti i Campi!", "errore", MessageBoxButtons.OK, MessageBoxIcon.Error);
        this.Close();
    }//se non sono stati inseriti i dati lo segnalo e esco

    else{//inizio operazioni di inserimnto

    //creo un nuovo dipendete con i valori inseriti
    Dipendente dipendente =new Dipendente(cognome.Text,nome.Text,Convert.ToDouble(stipendio.Text));

    //calcolo l'indice relativo del record nel file mediante la funzione di hash
    hashCode = 0;
    car = dipendente.getCognome().GetEnumerator();

    while (car.MoveNext() == true)
    {
      hashCode = hashCode + car.Current; //somma dei valori ascii dei caratteri del cognome
    }
    hashCode = hashCode * dipendente.getCognome().Length;//moltiplico la somma dei singoli caratteri per la lunghezza del cognome
    posizioneRelativa = (hashCode % archivio.getNumDip()); //calcolo il resto intero della divisione
    posizioneRelativa = posizioneRelativa * 55; //moltiplico indice record trovato per la lunghezza del record
    archivio.inserisci(posizioneRelativa, dipendente);//chiamo il metodo di archivio per inserire il nuovo dipendente

    }

    this.Close();
}
```

```
archivio.inserisci(posizioneRelativa, dipendente);//chiamo il metodo di archivio per inserire il nuovo dipendente
```

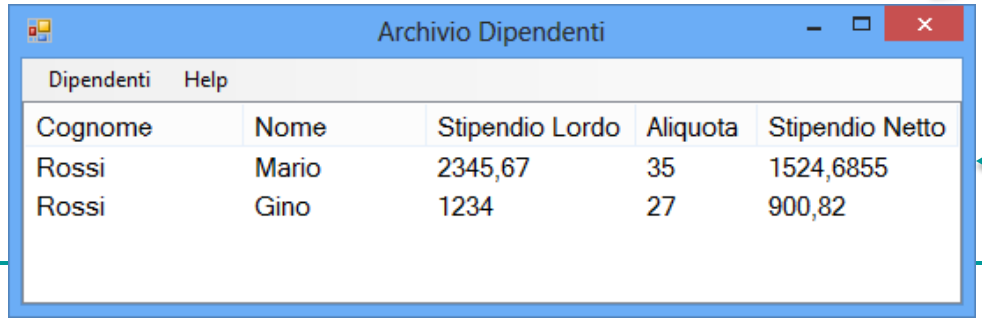



```
public void inserisci(long p, Dipendente dip)
   {

   //apro un Filestream sul file e mi posiziono sulla posizione corispondente al record
           fs = new FileStream(archivio, FileMode.Open);
           fs.Seek(p, SeekOrigin.Begin);
           //mentre trovo una collisione mi sposto sul prossimo e
           //se raggiungo la fine ricomincio dall'inizio
           while (fs.ReadByte()!=Convert.ToByte('-'))
           {
              MessageBox.Show("collisione cerco nuova posizione");
               if (fs.Position<(numeroDipendenti*55)- 54)
                   fs.Seek(54, SeekOrigin.Current);
               else
                   fs.Position = 0;
           }
           //trovata la posizione finalmente inserisco il record
           numRecInseriti++;
           fs.Seek(-1, SeekOrigin.Current);
           BinaryWriter bw = new BinaryWriter(fs);
           bw.Write(dip.getCognome().PadRight(19,' '));
           bw.Write(dip.getNome().PadRight(14,' '));
           bw.Write(dip.getStipendio());
           bw.Write(dip.getAliquota());
           bw.Write(dip.getStipendioNetto());
           bw.Close();
           fs.Close();
   }
```

Visualizzazione dell'archivio



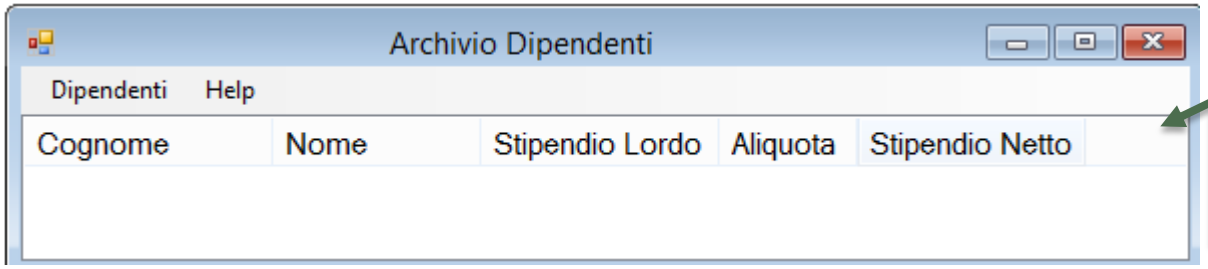

```
public void visualizzaElencoToolStripMenuItem_Click(object sender, EventArgs e)
{
    ListViewItem riga; //oggetto della listview per creare le righedella listview
    List<Dipendente> elenco; //arraylist di oggetti di tipo dipendente

    if (mioArchivio.getNumRec() == 0)//se l'archivio è vuoto messaggio
        MessageBox.Show("L'archivio è vuoto");
    else
    {//altrimenti chiamo la funzione visualizzaDipendenti che mi restituisce la lista
        elenco = mioArchivio.visualizzaDipendenti();
        resettaListaView();//funzione con la quale resettiamo la listview

        foreach (Dipendente trovato in elenco)
        { //ciclo iterativo ad oggetti. in pratica per ogni oggetto di tipo dipendente
          //contenuto nell'arraylist elenco si assegna il suo indirizzo a trovato e con esso
          // si aggiorna la listview
            riga = new ListViewItem(trovato.getCognome());
            riepilogo.Items.Add(riga);
            riga.SubItems.Add(trovato.getNome());
            riga.SubItems.Add(Convert.ToString(trovato.getStipendio()));
            riga.SubItems.Add(Convert.ToString(trovato.getAliquota()));
            riga.SubItems.Add(Convert.ToString(trovato.getStipendioNetto()));
            riga.SubItems.Add(Convert.ToString(trovato.getPosizione()));
        }
    }

}
```



Diamo un'occhiata alla funzione resettaListaView

```
private void resettaListaView() {
    riepilogo.Clear();
    riepilogo.Columns.Add("Cognome");
    riepilogo.Columns.Add("Nome");
    riepilogo.Columns.Add("Stipendio Lordo");
    riepilogo.Columns.Add("Aliquota");
    riepilogo.Columns.Add("Stipendio Netto");
    riepilogo.Columns.Add("");
    riepilogo.Columns[0].Width = 124;
    riepilogo.Columns[1].Width = 104;
    riepilogo.Columns[2].Width = 119;
    riepilogo.Columns[3].Width = 69;
    riepilogo.Columns[4].Width = 114;
    riepilogo.Columns[5].Width = 0;
    riepilogo.View = View.Details;
}
```

Se guardiamo il codice notiamo che alla listview è stato aggiunta una colonna nascosta nella quale sarà memorizzata la posizione di ciascun record contenuto nell'archivio. Questo sotterfugio ci semplifica di molto le operazioni di cancellazione e modifica perché come abbiamo già visto nelle precedenti esperienze l'utente selezionerà un' intera riga e quindi grazie al valore del campo nascosto contenuto in essa possiamo trovare automaticamente la posizione del record nell'archivio.



Campo nascosto