Ricorsione

Emilio Di Giacomo e Walter Didimo

Ricorsione

- La ricorsione è una tecnica di progettazione del software che si basa sull'uso di metodi/funzioni ricorsivi/e
- Un metodo/funzione ricorsivo/a è un metodo/funzione che invoca se stesso/a

Ricorsione

- Introdurremo la ricorsione facendo riferimento soprattutto al linguaggio Java
- Quanto diremo vale anche per il C *mutatis mutandis*
- Vedremo i vari esempi anche in C:
  - per evitare possibili errori le slide specifiche per il C sono evidenziate dalla scritta **`versione C`**

Funzioni (matematiche) ricorsive

- Per introdurre la ricorsione cominciamo con un esempio basato sull'uso di funzioni ricorsive
- Una funzione ricorsiva è una funzione definita in termini di se stessa
- Ad esempio, il fattoriale di n può essere definito ricorsivamente come segue:

$$n! = \begin{cases} 1 & \text{se } n=0 \\ n(n-1)! & \text{se } n>0 \end{cases}$$

Funzioni ricorsive

- La definizione consiste di due casi:
  - il caso base che si ha quando $n = 0$
  - il caso induttivo che si ha quando $n > 0$
- Nel caso induttivo $n!$ è definito in termini di $(n-1)!$
- Ad esempio, immaginiamo $n=3$:
  - poiché $3>0$ siamo nel caso induttivo dobbiamo calcolare $2!$, che a sua volta richiede di calcolare $1!$ e così via...

    $$3!=3\cdot 2!=3\cdot(2\cdot 1!)=3\cdot(2\cdot(1\cdot 0!))$$

  - per il calcolo di $0!$ siamo nel caso base quindi:

    $$3\cdot(2\cdot(1\cdot 1))=3\cdot(2\cdot 1)=3\cdot 2=6$$

Il metodo fattoriale(...)

- Il seguente è un metodo ricorsivo per il calcolo del fattoriale

```
public static int fattoriale(int n){
   int f;
   if (n==0)
       f = 1;
   else
       f = n*fattoriale(n-1); // fatt, 5
   return f;
}
```

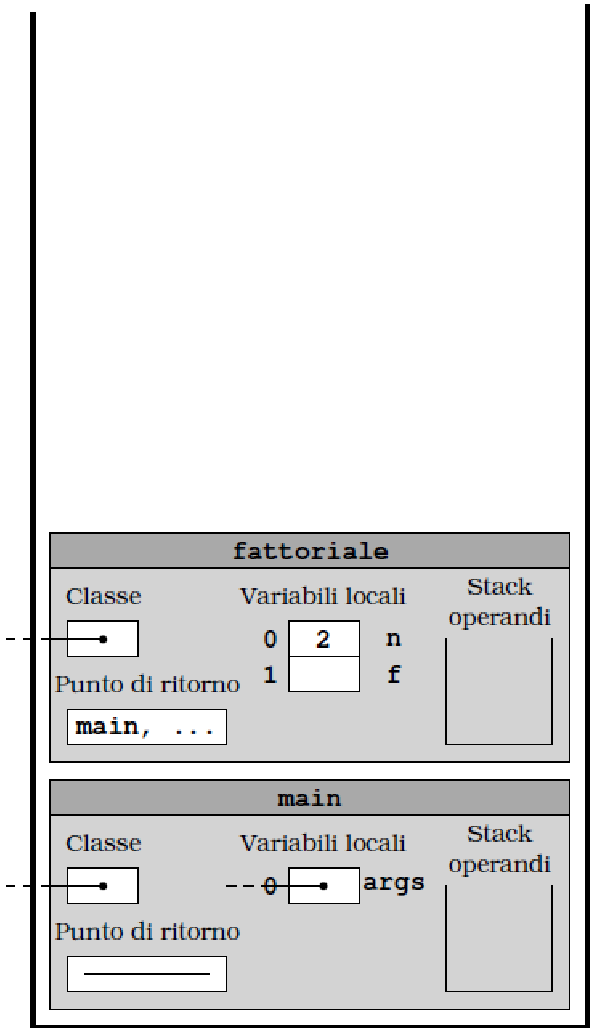
Metodi ricorsivi

- Il codice rispecchia la struttura della definizione ricorsiva
- L'istruzione *if-else* permette di stabilire se siamo nel caso base o nel caso induttivo
  - nel primo caso viene assegnato alla variabile "risultato" *f* il valore 1
  - nel secondo il metodo richiama se stesso per calcolare (n-1)!

Metodi ricorsivi

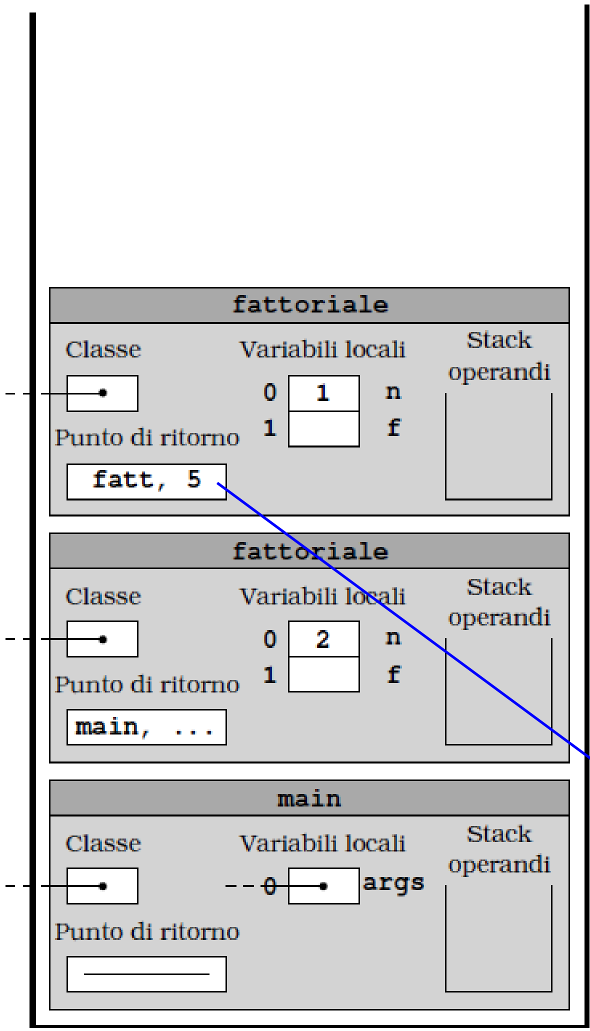
- Che cosa succede quando un metodo richiama se stesso?
- quando un metodo viene invocato:
  - la JVM crea un record di attivazione per esso
  - tale record viene posto in cima alla pila di attivazione e si avvia l'esecuzione del metodo
  - Il metodo chiamante rimane in attesa.
- Questo è esattamente ciò che succede anche nel caso dei metodi ricorsivi
- Consideriamo, ad esempio, l'invocazione *fattoriale(2)*

Esempio di esecuzione




```
public static int fattoriale(int n){
  int f;
  if (n==0)
    f=1;
  else
    f=n*fattoriale(n-1); // fatt, 5
  return f;
}
```

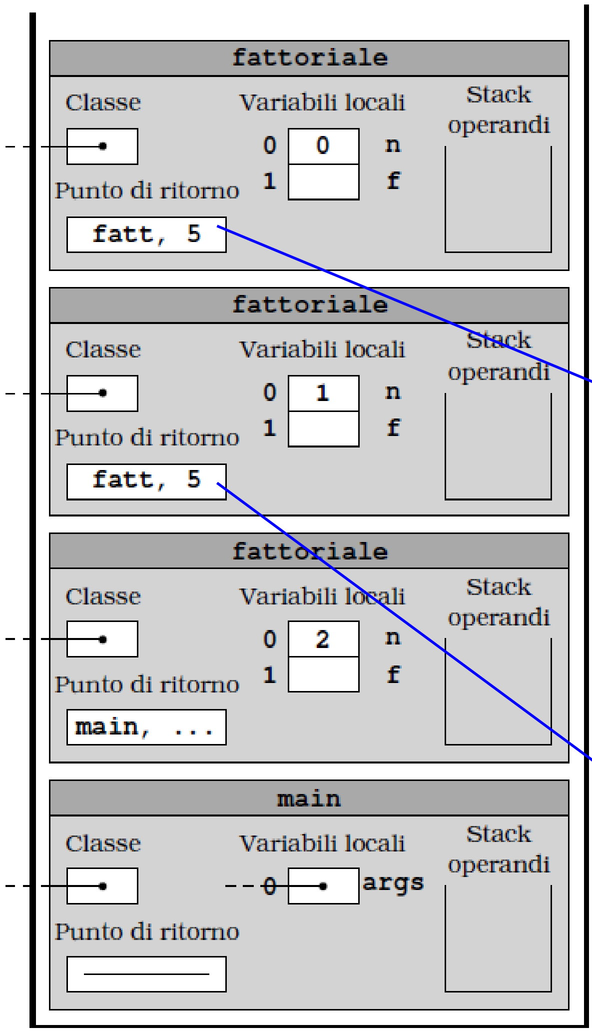
Esempio di esecuzione




```
public static int fattoriale(int n){
   int f;
   if (n==0)
      f=1;
   else
      f=n*fattoriale(n-1); // fatt, 5
   return f;
}
```

```
public static int fattoriale(int n){
   int f;
   if (n==0)
      f=1;
   else
      f=n*fattoriale(n-1); // fatt, 5
   return f;
}
```

Esempio di

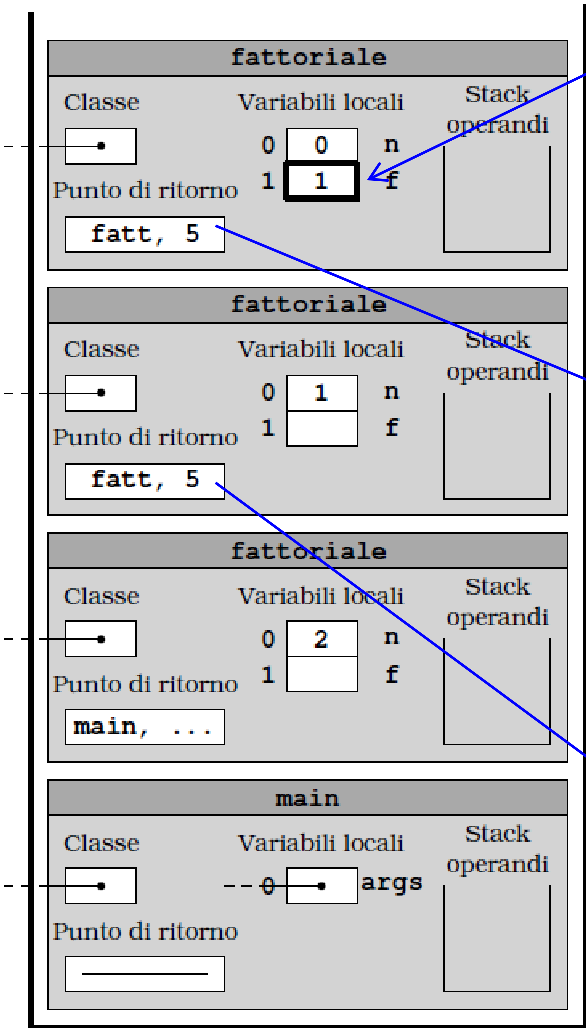



```
public static int fattoriale(int n){
  int f;
  if (n==0)
    f=1;
  else
    f=n*fattoriale(n-1); // fatt, 5
  return f;
}
```

```
public static int fattoriale(int n){
  int f;
  if (n==0)
    f=1;
  else
    f=n*fattoriale(n-1); // fatt, 5
  return f;
}
```

```
public static int fattoriale(int n){
  int f;
  if (n==0)
    f=1;
  else
    f=n*fattoriale(n-1); // fatt, 5
  return f;
}
```

Esempio di

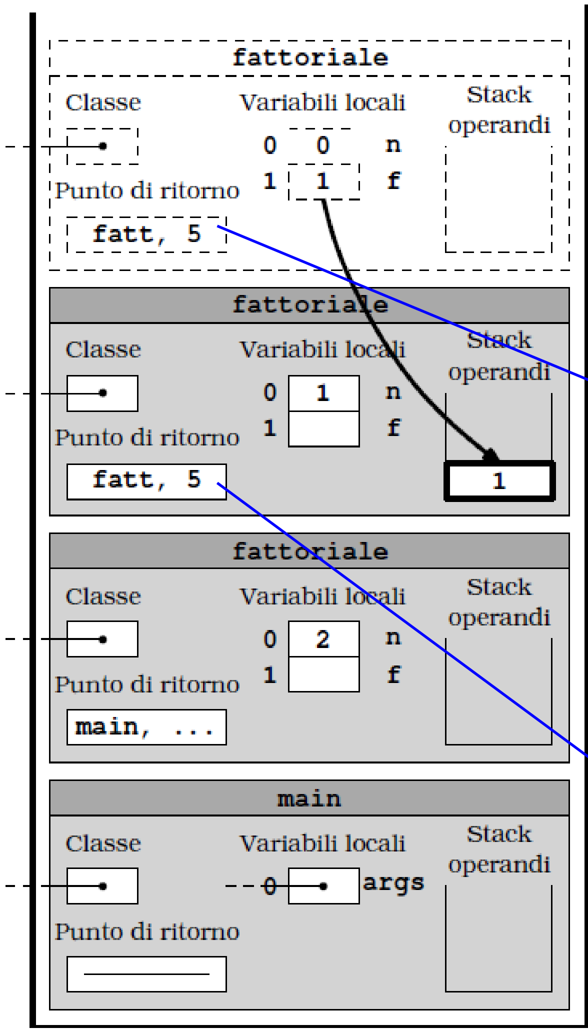



```
public static int fattoriale(int n){
   int f;
   if (n==0)
     f=1;
   else
     f=n*fattoriale(n-1); // fatt, 5
   return f;
}
```

```
public static int fattoriale(int n){
   int f;
   if (n==0)
     f=1;
   else
     f=n*fattoriale(n-1); // fatt, 5
   return f;
}
```

```
public static int fattoriale(int n){
   int f;
   if (n==0)
     f=1;
   else
     f=n*fattoriale(n-1); // fatt, 5
   return f;
}
```

Esempio di

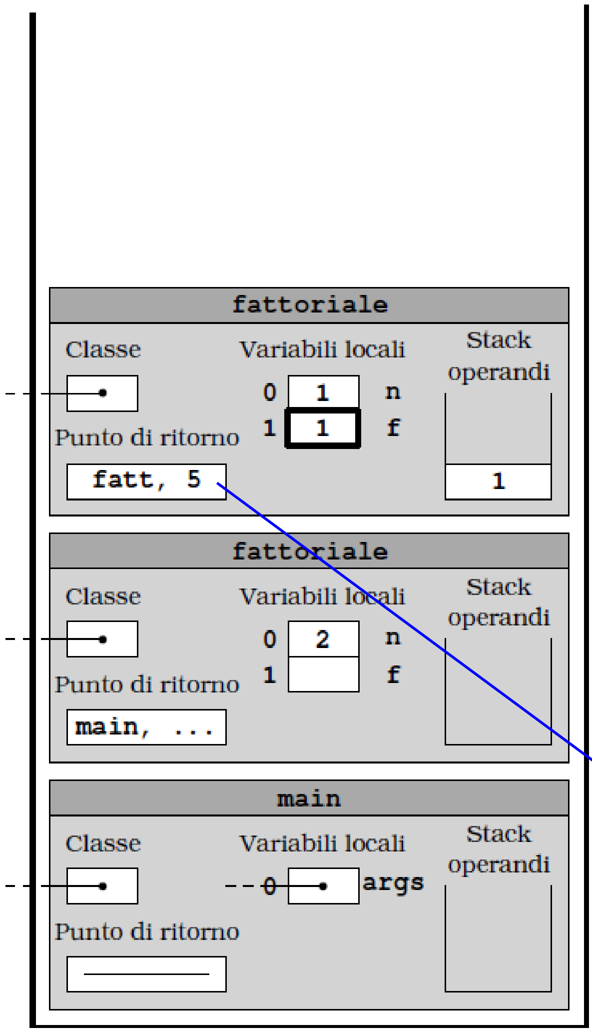



```
public static int fattoriale(int n){
  int f;
  if (n==0)
    f=1;
  else
    f=n*fattoriale(n-1); // fatt, 5
  return f;
}
```

```
public static int fattoriale(int n){
  int f;
  if (n==0)
    f=1;
  else
    f=n*fattoriale(n-1); // fatt, 5
  return f;
}
```

```
public static int fattoriale(int n){
  int f;
  if (n==0)
    f=1;
  else
    f=n*fattoriale(n-1); // fatt, 5
  return f;
}
```

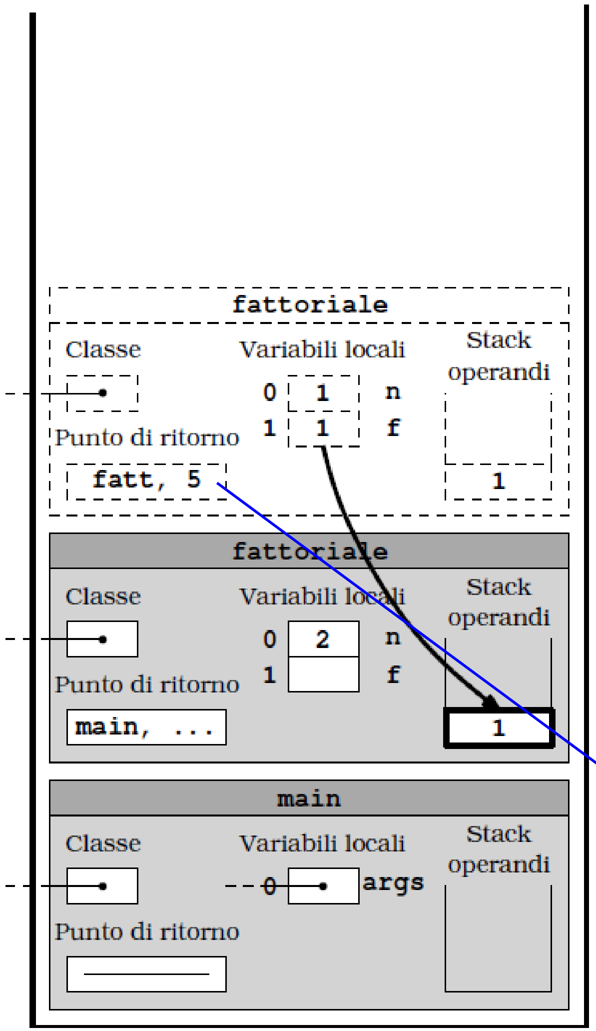
Esempio di esecuzione




```
public static int fattoriale(int n){
  int f;
  if (n==0)
    f=1;
  else
    f=n*fattoriale(n-1); // fatt, 5
  return f;
}
```

```
public static int fattoriale(int n){
  int f;
  if (n==0)
    f=1;
  else
    f=n*fattoriale(n-1); // fatt, 5
  return f;
}
```

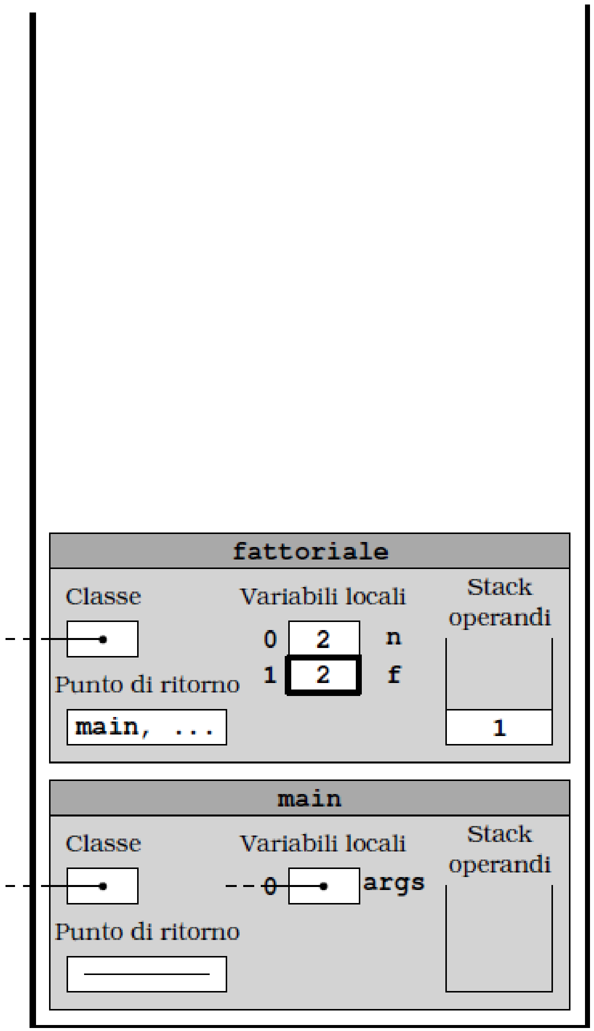
Esempio di esecuzione




```
public static int fattoriale(int n){
  int f;
  if (n==0)
    f=1;
  else
    f=n*fattoriale(n-1); // fatt, 5
  return f;
}
```

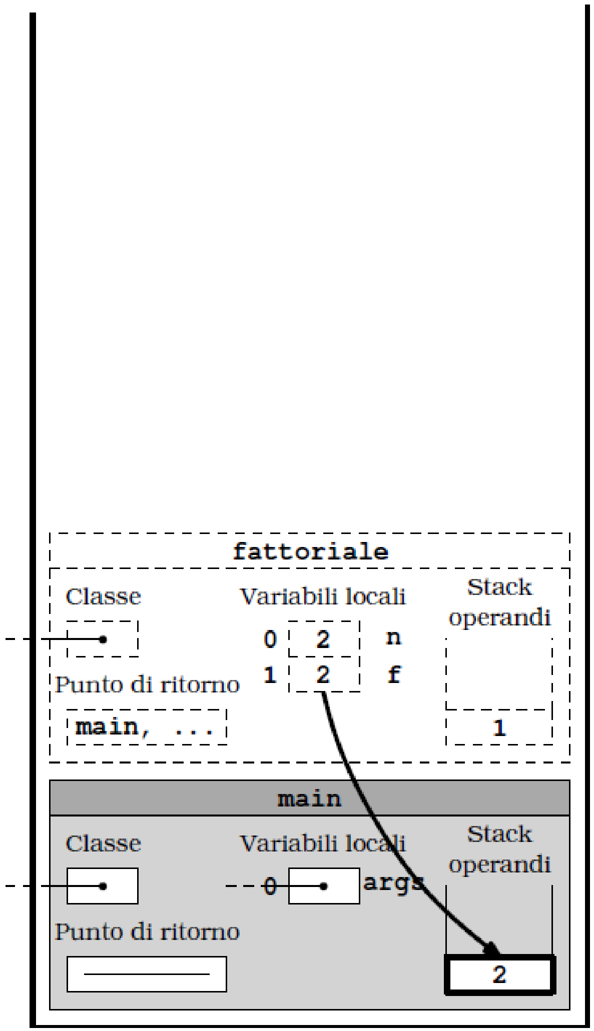
```
public static int fattoriale(int n){
  int f;
  if (n==0)
    f=1;
  else
    f=n*fattoriale(n-1); // fatt, 5
  return f;
}
```

Esempio di esecuzione




```
public static int fattoriale(int n){
  int f;
  if (n==0)
    f=1;
  else
    f=n*fattoriale(n-1); // fatt, 5
  return f;
}
```

Esempio di esecuzione




```
public static int fattoriale(int n){
   int f;
   if (n==0)
     f=1;
   else
     f=n*fattoriale(n-1); // fatt, 5
   return f;
}
```

Un secondo esempio

- Consideriamo la seguente funzione per il calcolo dell’n-esimo termine F(n) della successione di Fibonacci

$$F(n) = \begin{cases} 1 & n=1 \\ 1 & n=2 \\ F(n-1)+F(n-2) & n>2 \end{cases}$$

- ogni termine della successione è la somma dei due che lo precedono:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Il metodo fibonacci(...)

```
public static int fibonacci(int n){
   int f;
   if (n==1)
       f = 1;
   else if (n==2)
       f = 1;
   else
       f = fibonacci(n-1)+fibonacci(n-2);
   return f;
}
```

Esempi in C

- La versione C delle due funzioni *fattoriale* e *fibonacci* che abbiamo scritto è facilmente ottenibile

```
int fattoriale(int n){
  int f;
  if (n==0)
      f = 1;
  else
      f = n*fattoriale(n-1);
  return f;
}
```

versione C

Esempi in C

```
int fibonacci(int n){
  int f;
  if (n==1)
     f = 1;
  else if (n==2)
     f = 1;
  else
     f = fibonacci(n-1)+fibonacci(n-2);
  return f;
}
```

versione C

Commenti

- I due esempi visti ci permettono di discutere di alcune caratteristiche che un metodo ricorsivo deve avere affinché funzioni correttamente
- Un metodo ricorsivo deve avere uno o più casi base
  - se non ci fossero il metodo continuerebbe a chiamare se stesso indefinitamente
- I casi base potrebbero anche non comparire esplicitamente

Commenti

- Versione alternativa di fattoriale

```
public static int fattoriale(int n){
  int f = 1;
  if (n>0)
    f = n*fattoriale(n-1);
  return f;
}
```

- In questo caso non c'è un caso base esplicito
- ma se *n* è uguale a *0* l'*if* non viene eseguito e viene restituito il valore *1*: siamo nel caso base!!

Commenti

- La seconda condizione perché un metodo ricorsivo funzioni correttamente è che la sequenza di attivazioni ricorsive sia tale da ricadere, prima o poi, nel caso base

Commenti

- Il seguente metodo è errato perché ad ogni attivazione ricorsiva ci si allontana sempre di più dal caso base

```
public static int errato1(int n){
   int f = 1;
   if (n>0)
     f = n*errato1(n+1); // ERRORE !!!
   return f;
}
```

Commenti

- Anche il seguente metodo è errato
- Ad ogni attivazione ci si avvicina al caso base
  - ma per alcuni valori (ad es. 3) non si ricade mai nel caso base

```
public static int errato2(int n){
  int f;
  if (n==2)
    f = 1;
  else
    f = n*errato2(n/2); // ERRORE !!!
  return f;
}
```

Commenti

- Nota: nei due esempi precedenti abbiamo detto che il metodo ricorsivo continua ad invocare se stesso indefinitamente
  - questo è ciò che succede in linea di principio
  - in pratica le varie attivazioni ricorsive saturano la memoria riservata per la pila di attivazione
  - in Java viene generato un errore di tipo *StackOverflowException*
  - in C si ha una terminazione imprevista del programma

La ricorsione per risolvere problemi

- I metodi ricorsivi visti sono stati scritti partendo da funzioni definite in maniera induttiva
- Questo non è il solo caso in cui è possibile utilizzare metodi ricorsivi
- La ricorsione può essere utilizzata come una metodologia per risolvere un problema:
  - il problema viene scomposto in uno o più problemi della stessa natura ma di “dimensione” ridotta
  - i problemi individuati vengono risolti ricorsivamente
- La sequenza di attivazioni ricorsive termina quando si raggiunge un caso (*caso base*) in cui il problema è così “piccolo” da poter essere risolto direttamente

Esempio 1: binario

- Vogliamo scrivere un metodo ricorsivo che dato un numero non negativo n ci restituisca, sotto forma di stringa, la rappresentazione binaria di n

Esempio 1: binario

- Se *n=0* oppure *n=1*, si può restituire *n* sotto forma di stringa
- Se *n>1* la cifra meno significativa della rappresentazione di *n* è il resto *r* della divisione di *n* per *2*
- Le restanti cifre sono quelle della codifica binaria del quoziente *q* della divisione di *n* per *2*:
  - dobbiamo convertire *q*
- Il problema è lo stesso ma su un valore più basso (cioè *q*)
- Continuando a dividere si arriverà in uno dei casi base (*0* o *1*)

Esempio 1: binario

```
public static String binario(int n){
   String b;
   if (n<=1){
        b = ""+n;
   }else{
        b = binario(n/2)+(n%2);
   }
   return b;
}
```

Esempio 2: mcd

- Vogliamo scrivere un metodo per il calcolo del Massimo Comun Divisore (MCD) di due numeri non negativi n ed m
- Il massimo comun divisore $MCD(n,m)$ di due numeri interi non negativi n ed m è il più grande intero che divide sia n che m
  - $MCD(n,0) = n$ per ogni $n>0$
  - poniamo convenzionalmente $MCD(0,0) = 0$

Esempio 2: mcd

- Algoritmo di Euclide per il calcolo del MCD
- Osservazione: se *n* e *m* hanno un divisore comune *d* allora *d* sarà un divisore anche di *n-m*
- Assumiamo $n \geq m$
  - se $m=0$, $MCD(n,m)=n$
  - altrimenti $MCD(n,m)=MCD(m,n-m)$
- Esempio $n=35$, $m=14$

$$
\begin{aligned}
MCD(35,14) &= MCD(21,14)\\
&= MCD(14,7)\\
&= MCD(7,7)\\
&= MCD(7,0) = 7
\end{aligned}
$$

Esempio 2: mcd

```
public static int mcd(int n, int m){
   int d;
   if (m<=0){
       d = n;
   }else{
       if (m>n-m)
             d = mcd(m,n-m);
       else
             d = mcd(n-m,m);
   }
   return d;
}
```

Esempio 3: maxArray

- Vogliamo scrivere un metodo ricorsivo per l’individuazione dell’elemento massimo in un array

Esempio 3: maxArray

- Se l’array contiene un solo elemento allora il massimo è l’elemento stesso
- altrimenti, il massimo è pari al più grande tra l’elemento in prima posizione e il massimo del sottoarray costituito dagli elementi dalla seconda posizione in poi

Esempio 3: maxArray

```
public static int maxArray(int[] a, int i){
   int max;
   if (i==a.length-1){
       max = a[i];
   }else{
       max = maxArray(a,i+1);
       if (a[i]>max)
              max = a[i];
   }
   return max;
}
```

Esempio 3: commenti

- In questo esempio è stato introdotto un parametro aggiuntivo rispetto a quello “naturale”
- Il parametro *a* è previsto dalla definizione del problema mentre *i* è stato aggiunto per controllare la ricorsione
  - in una versione non ricorsiva questo parametro sarebbe assente

Esempio 3: commenti

- L’aggiunta di parametri può anche essere vista come una variazione nella formulazione del problema
- Il metodo *maxArray(a,i)* cioè
  - non effettua la ricerca del massimo in un array *a*
  - ma la ricerca del massimo dell’array *a* tra gli elementi che si trovano nelle posizioni dalla *i* in poi
- In questi casi può essere utile definire un ulteriore metodo per l’avvio della ricorsione

```
public static int massimo(int[] a){
    return maxArray(a,0);
}
```

Esempi 1, 2 e 3 in C

- La versione C della funzione *binario* è più complessa della versione Java a causa della gestione delle stringhe
- Vengono usate le funzioni:
  - *malloc* per allocare dinamicamente le stringhe
  - *sprintf* per convertire un *int* in stringa
  - *strlen* per conoscere la lunghezza delle stringhe
  - *strcat* per concatenare due stringhe

versione C

Esempio 1 in C: binario

```
char* binario(int n){
   char * b;
   if (n<=1){
        b=malloc(2*sizeof(char));
        sprintf(b,"%d",n);
   }else{
        char * b1=binario(n/2);
        char * b2=malloc(2*sizeof(char));
        sprintf(b2,"%d",n%2);
        int n1=strlen(b1);
        int n2=strlen(b2);
        b=malloc((n1+n2+1)*sizeof(char));
        b=strcat(b1,b2);
   }
   return b;
}
```

versione C

Esempio 2 in C: mcd

```
int mcd(int n, int m){
   int d;
   if (m<=0){
        d = n;
   }else{
        if (m>n-m)
              d = mcd(m,n-m);
        else
              d = mcd(n-m,m);
   }
   return d;
}
```

versione C

Esempio 3 in C: maxArray

```
int maxArrayRic(int a[], int dim, int i){
   int max;
   if (i==dim-1){
       max = a[i];
   }else{
       max = maxArrayRic(a,dim,i+1);
       if (a[i]>max)
           max = a[i];
   }
   return max;
}

int maxArray(int a[], int dim){
   return maxArrayRic(a,dim,0);
}
```

versione C

Progettare un metodo ricorsivo

- Come si progetta un metodo ricorsivo?
- Si individuano uno o più casi in cui il problema può essere risolto facilmente
  - questi saranno i casi base della ricorsione
- Si stabilisce un modo per decomporre il problema in uno o più sottoproblemi della stessa natura
- Bisogna assicurarsi che applicando ripetutamente la decomposizione si ricada in uno dei casi base

Progettare un metodo ricorsivo

- Il metodo tipicamente conterrà un’istruzione condizionale per stabilire in quale dei casi individuati ci troviamo
- Il codice dei casi base viene scritto direttamente
- Per i casi induttivi si deve assumere di avere a disposizione un metodo in grado di risolvere i sottoproblemi...
- ...e si devono usare le soluzioni di questi per “costruire” la soluzione del problema di partenza

Esempio: stringa palindroma

- Vogliamo scrivere un metodo ricorsivo per determinare se una data stringa è palindroma
- Se la stringa consiste di un solo carattere è chiaramente palindroma
- Se la stringa ha più di un carattere, come possiamo ricondurre il problema ad un problema più semplice?
  - Confrontiamo il primo e l'ultimo carattere
  - se sono uguali verifichiamo se risulta palindroma la stringa ottenuta rimuovendoli
- Il problema di verificare se una stringa di *n* caratteri è palindroma si riconduce al problema di stabilire se una stringa di *n-2* caratteri è palindroma
- Abbiamo ridotto la dimensione del problema

Esempio: stringa palindroma

- Abbiamo coperto tutto i casi?
  - Se il numero di caratteri *n* della stringa è dispari, rimuovendo due caratteri alla volta si arriva ad una stringa con un solo carattere
    - "RADAR" → "ADA" → "D"
  - Se *n* è pari non si ricade mai nel caso base; infatti rimuovendo due caratteri alla volta si arriva alla stringa vuota
    - "ANNA" → "NN" → ""
- Dobbiamo rivedere le nostre scelte
- Aggiungiamo come caso base il caso della stringa vuota

Esempio: stringa palindroma

- Scriviamo il codice

```
public static boolean palindroma(String s){
  int n = s.length();  // lunghezza di s
  boolean p; // risultato
  if (n<=1){
     // casi base
  }else{
     // caso induttivo
  }
  return p;
}
```

Esempio: stringa palindroma

- Scriviamo il codice

```
public static boolean palindroma(String s){
  int n = s.length();  // lunghezza di s
  boolean p; // risultato
  if (n<=1){
     p = true;
  }else{
     // caso induttivo
  }
  return p;
}
```

Esempio: stringa palindroma

- Come si scrive il caso induttivo?
- Assumiamo di avere un metodo *x(...)* che, data la sottostringa di *s* ottenuta rimuovendo il primo e l'ultimo carattere, ci dica se questa è palindroma o meno
- Come possiamo sfruttare il metodo *x(...)* per stabilire se *s* è palindroma?
  - Basta prendere il risultato restituito da *x(...)* e metterlo in AND logico con la condizione *s.charAt(0)==s.charAt(n-1)*

Esempio: stringa palindroma

- Scriviamo il codice

```
public static boolean palindroma(String s){
  int n = s.length();  // lunghezza di s
  boolean p; // risultato
  if (n<=1){
     p = true;
  }else{
     p = (s.charAt(0)==s.charAt(n-1));
     p = p && x(s.substring(1, n-1));
  }
  return p;
}
```

Esempio: stringa palindroma

- Chi ci dà il metodo *x(...)*?
- È un metodo che data una stringa ci dice se questa è palindroma
- Ma questo è proprio ciò che fa il metodo *palindroma(...)* che stiamo scrivendo!

Esempio: stringa palindroma

- Codice finale

```
public static boolean palindroma(String s){
   int n = s.length();  // lunghezza di s
   boolean p; // risultato
   if (n<=1){
       p = true;
   }else{
       p = (s.charAt(0)==s.charAt(n-1));
       p = p && palindroma(s.substring(1, n-1));
   }
   return p;
}
```

Commenti

- Ovviamente non c’è bisogno di ricorrere al metodo *x(...)*
  - possiamo scrivere direttamente la versione ricorsiva del metodo
- La soluzione vista è corretta ma l’uso del metodo *substring(...)* per accorciare la stringa causa un certo spreco di memoria
  - ad ogni invocazione viene creato un nuovo oggetto *String*
- Il codice può essere modificato come segue

Esempio: stringa palindroma

- Codice modificato

```
private static boolean palindroma(String s,int i,int j){
   boolean p;
   if (i>=j)
       p=true;
   else{
       p=(s.charAt(i)==s.charAt(j))&&palindroma(s,i+1,j-1);
   }
   return p;
}

public static boolean palindroma(String s){
   return palindroma(s, 0, s.length()-1);
}
```

Palindroma in C

```
int palindromaRic(char * s, int i, int j){
   int p;
   if (i>=j)
       p=1;
   else{
       p=(s[i]==s[j])&&palindromaRic(s,i+1,j-1);
   }
   return p;
}

int palindroma(char * s){
   return palindromaRic(s, 0, strlen(s)-1);
}
```

versione C

Tipi ricorsivi

- La ricorsione si presta particolarmente bene a risolvere problemi definiti su tipi che hanno un'intrinseca natura ricorsiva
- Un tipo ricorsivo è un tipo che può essere definito in termini di se stesso
- Ad esempio immaginiamo di voler scrivere una classe per rappresentare delle espressioni matematiche costituite da numeri, lettere (che rappresentano variabili) e operatori aritmetici +, -, $\times$, /

Tipi ricorsivi: Espressione

- Il tipo *espressione* può essere definito come segue:
  - Un numero è un’espressione
  - Una lettera è un’espressione
  - Se α e β sono espressioni, allora lo sono anche $(\alpha+\beta)$, $(\alpha-\beta)$, $(\alpha\times\beta)$ e (α/β)

Tipi ricorsivi

- La definizione precedente è ricorsiva in quanto un'espressione che non sia un numero o una lettera è definita a partire da due sottoespressioni
- Anche in una definizione di un tipo ricorsivo deve esistere un caso base
  - Nel nostro esempio i casi base si hanno quando l'espressione consiste di un numero o di una lettera
- Scriviamo una classe *Espressione* strutturata sulla base della definizione precedente

La classe Espressione

```
public class Espressione {
   private String element;
   private Espressione exp1;
   private Espressione exp2;

   public Espressione(String element, Espressione exp1,
                                               Espressione exp2){
      this.element = element;
      this.exp1 = exp1;
      this.exp2 = exp2;
   }

   ...
}
```

La classe Espressione

```
public class Espressione {
   ...

   public String getElement(){
       return element;
   }

   public Espressione getExp1(){
       return exp1;
   }

   public Espressione getExp2(){
       return exp2;
   }

}
```

La classe Espressione

- Nei casi base la variabile di istanza *element* memorizza il numero o la lettera
- Nel caso induttivo memorizza invece l'operatore
- Le due variabili *exp1* ed *exp2* rappresentano le due sottoespressioni:
  - tali campi saranno nulli nel caso base

Stampa di espressione

- Immaginiamo di voler scrivere un metodo che riceve in ingresso un oggetto *Espressione* e lo stampa
- Vista la natura ricorsiva di un oggetto *Espressione* il metodo sarà un metodo ricorsivo

Stampa di espressione

```
public static String stampaEspressione(Espressione e){
   String s;
   if (e.getExp1()==null){
        s = e.getElement();
   }else{
        String s1 = stampaEspressione(e.getExp1());
        String s2 = stampaEspressione(e.getExp2());
        s = "("+s1+" "+e.getElement()+" "+s2+")";
   }
   return s;
}
```

Espressione in C

- Vediamo l'uso del tipo ricorsivo *espressione* in C
- Possiamo definire la seguente *struct*

```
struct espressione{
     char * element;
     struct espressione * exp1;
     struct espressione * exp2;
};
```

versione C

Espressione in C

- Scriviamo quindi un metodo ricorsivo per stampare un'*espressione*

```
void stampaEspressione(struct espressione e){
      if(e.exp1==NULL)
            printf("%s",e.element);
      else{
            printf("(");
            stampaEspressione(*e.exp1);
            printf("%s",e.element);
            stampaEspressione(*e.exp2);
            printf(")");
      }
}
```

versione C

Ricorsione vs. iterazione

- In tutti gli esempi visti abbiamo fatto uso della ricorsione
- Per quasi tutti i codici visti è immediato trovare un equivalente che non fa uso della ricorsione ma usa le istruzioni iterative
- Ci sono casi in cui la ricorsione è necessaria?
- Si può dimostrare che per ogni metodo ricorsivo ne esiste uno “equivalente” iterativo

Ricorsione vs. iterazione

- La conversione di un metodo ricorsivo è particolarmente semplice nel caso della cosiddetta ricorsione di coda (tail recursion)
- Si parla di ricorsione di coda quando dopo la chiamata ricorsiva il metodo non fa nulla se non, eventualmente, restituire un valore

Ricorsione di coda

- Struttura di un metodo che usa la ricorsione di coda

```
<tipo> m_ric(<tipo> x){
  <tipo> r // risultato
  if (x soddisfa la proprietà p)
      istruzioni caso base
  else{
      istruzioni caso induttivo
      y = g(x) //riduz. della "dimens." del problema
      r = m_ric(y) //invocazione ricorsiva
  }
  return r // può essere assente se il metodo è void
}
```

Ricorsione di coda

- Versione iterativa equivalente

```
<tipo> m_iter(<tipo> x){
  <tipo> r // risultato
  while (x NON soddisfa la proprietà p){
     istruzioni caso induttivo
     x = g(x) //riduz. della "dimens." del problema
  }
  istruzioni caso base
  return r // può essere assente se il metodo è void
}
```

Esempio: mcd

- Versione ricorsiva

```
public static int mcd(int n, int m){
   int d;
   if (m<=0){
       d = n;
   }else{
       if (m>n-m)
             d = mcd(m,n-m);
       else
             d = mcd(n-m,m);
   }
   return d;
}
```

Esempio: mcd

- Versione iterativa

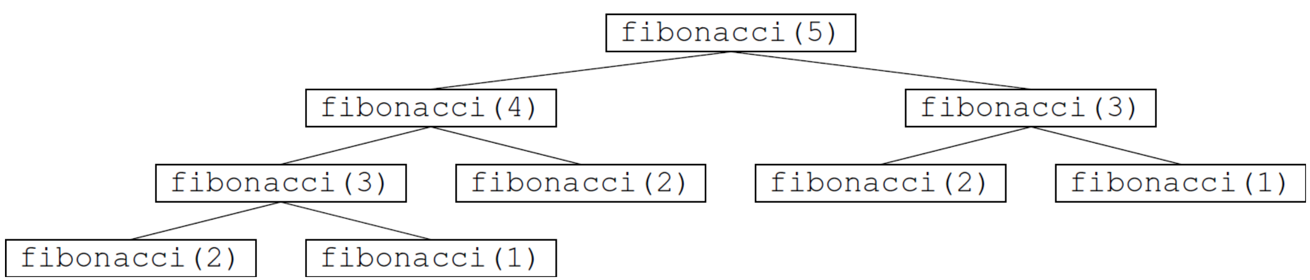
```
public static int mcd_i(int n, int m){
   int d;
   while (m>0){
        if (m>n-m){
               int a = n-m;
               n = m;
               m = a;
        }else{
               n = n-m;
        }
   }
   d = n;
   return d;
}
```

Ricorsione vs. iterazione

- Visto che un metodo ricorsivo può sempre essere trasformato in un metodo iterativo, quando è opportuno usare la ricorsione?
- Non esistono regole, possiamo però dare alcune indicazioni

Ricorsione vs. iterazione

- L'iterazione è di solito più efficiente:
  - sia rispetto al tempo: l'invocazione di un metodo richiede un certo lavoro per gestire il corrispondente record di attivazione
  - sia rispetto alla memoria: per ognuna delle attivazioni ricorsive che rimangono in sospeso, esiste un record di attivazione diverso
- La ricorsione inoltre è spesso inefficiente perché ripete più volte gli stessi calcoli



Ricorsione vs. iterazione

- La ricorsione permette spesso di scrivere codici più compatti e semplici:
  - soprattutto quando si affrontano problemi intrinsecamente ricorsivi
- Ciò deriva dal fatto che, nei casi più complessi, la versione iterativa di un programma ricorsivo deve gestire esplicitamente delle strutture dati per memorizzare i risultati intermedi di calcolo
- In un metodo ricorsivo ciò viene gestito automaticamente dalla pila di attivazione

Ricorsione vs. iterazione

- In conclusione:
  - Una soluzione iterativa è preferibile se non risulta particolarmente complessa o se le prestazioni rappresentano un aspetto critico
  - La ricorsione è invece preferibile quando una soluzione iterativa risulta più complessa da scrivere o più difficile da individuare