

Pubblico e privato

- Per capire il concetto di pubblico e privato è possibile pensare a diversi oggetti del mondo reale. Ad esempio: bancomat.
 - La tessera bancomat esegue operazioni sul conto corrente, ma in modo trasparente per l'utente (quindi offre un'interfaccia e maschera l'interno)
 - Inoltre fornisce protezione, perché solo tramite la carta è possibile accedere al conto corrente e: ottenere il saldo (get) o prelevare denaro (set).
 - Il saldo è come una variabile d'istanza (privata) di un oggetto della classe "ContoCorrente".

Pubblico e privato

- Nascondendo i dettagli all'esterno, possiamo assicurare all'utente che l'oggetto si trova sempre in uno stato consistente, cioè uno stato permesso dal progetto dell'oggetto.
 - Questo non garantisce che lo stato sia quello atteso dall'utente: se usa male l'interfaccia lo stato sarà consistente, ma non quello desiderato.
 - Esempio: invece di prelevare 100, prelevo 1000.

Pubblico e privato

- Dichiarando private le variabili d'istanza di una classe, le informazioni associate agli oggetti non sono direttamente accessibili:
incapsulamento dell'informazione.
- Le si può modificare solo con metodi **set**. Una classe è **immutabile** se è priva di metodo set.
 - ▣ Esempio: la classe **String**. In realtà le stringhe non vengono mai modificate, si crea un'altra stringa e si aggiorna il riferimento perché punti ad essa.

Pubblico e privato

- ❑ I metodi **get** tipicamente non hanno parametri e ritornano il campo come valore di ritorno.
 - ▣ Il valore di ritorno, come in C, si determina con la parola chiave **return** (che termina il metodo)
- ❑ I metodi **set** tipicamente hanno come tipo di ritorno void e passano per parametro il valore da settare.
- ❑ Entrambi devono essere **public** per poter essere usati da qualunque altra classe.

Pubblico e privato

```
class Date
```

```
{
```

```
//campi
```

```
private int giorno, mese, anno;
```

campi privati

```
//metodi
```

```
public int getGiorno()
```

```
{ return this.giorno;
```

```
};
```

```
public void setGiorno(int data)
```

```
{ this.giorno=data;
```

```
}
```

metodi pubblici

```
}
```

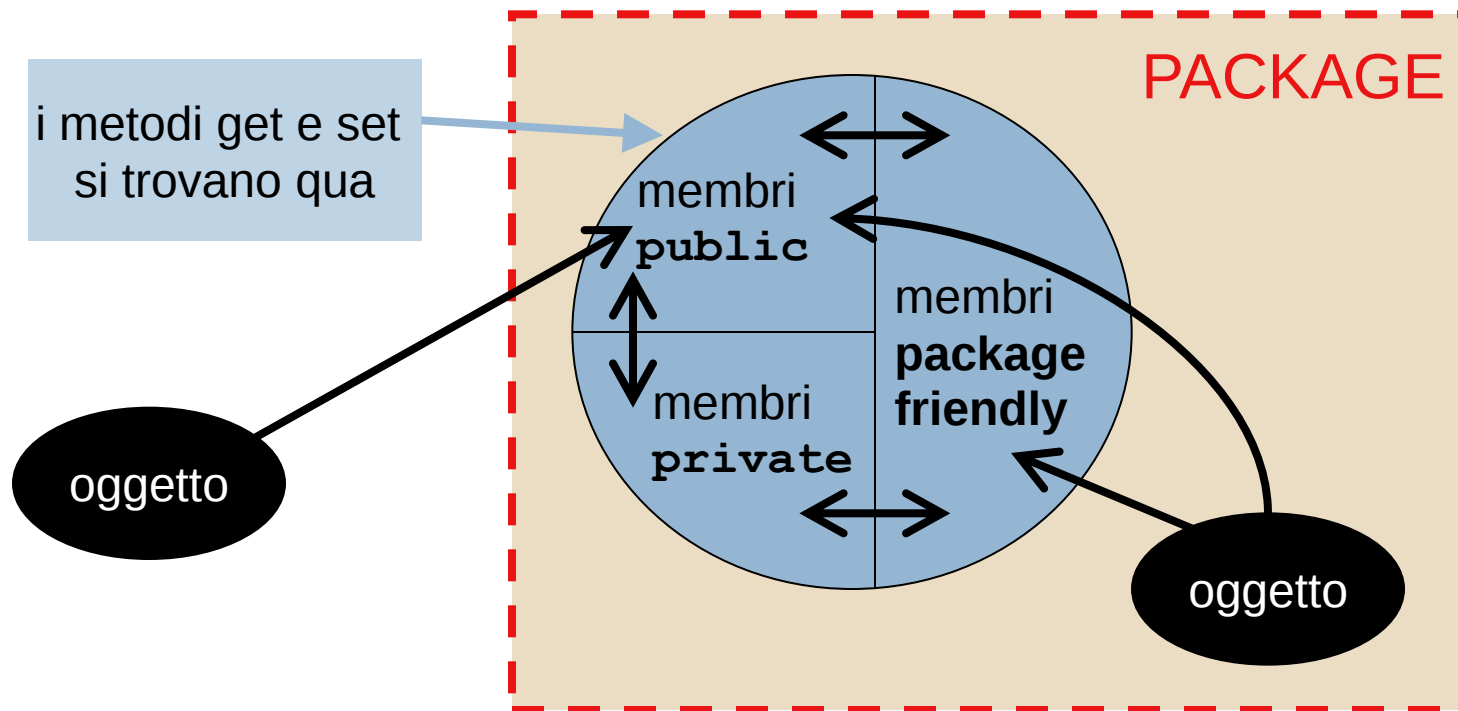
Pubblico e privato

- Java fornisce supporto per l'incapsulamento **a livello di linguaggio** con **public** e **private**.
- In effetti ci sono 3 livelli di incapsulamento:
 - ▣ i membri definiti **public** possono essere acceduti liberamente da qualunque classe;
 - ▣ i membri definiti **private** possono essere utilizzati solo da altri membri della stessa classe;
 - ▣ i membri non definiti sono accessibili a tutte le classi **dello stesso package**.

Pubblico e privato

- Si può anche dichiarare **public** una classe. Tuttavia questo non rende automaticamente pubblici i suoi membri: senza la presenza di altri modificatori tutti i membri hanno accessibilità di package.
- Nella documentazione non si dovrebbe trovare descrizione di metodi e campi privati perché sono membri che non si possono utilizzare fuori dalla classe (sono dettagli implementativi)

Pubblico e privato



Pubblico e privato

- Orientativamente conviene impostare:
 - ▣ le classi: **public**
 - ▣ i campi: **private**
 - ▣ i costruttori: **public**
 - ▣ i metodi get e set: **public**
 - ▣ gli altri metodi: dipende caso per caso a seconda del contratto della classe.

Pubblico e privato

- Una classe immutabile (cioè priva di metodi di modifica) presenta un grande vantaggio: i riferimenti ad oggetti di quella classe possono essere liberamente condivisi.
- Al contrario occorre fare attenzione quando si condividono oggetti modificabili.
 - In particolare è pericoloso un metodo di accesso che restituisce un riferimento a un campo che non sia di una classe immutabile.

Pubblico e privato

```
class Studente
{
    //campi
    private String nome;
    private int matric;
    private Date natoIl;

    //metodi
    public String getNome() {return this.nome};
    public int getMatric() {return this.matric};
    public Date getNatoIl() {return this.natoIl};
}
```

c'è qualcosa che non funziona ancora....

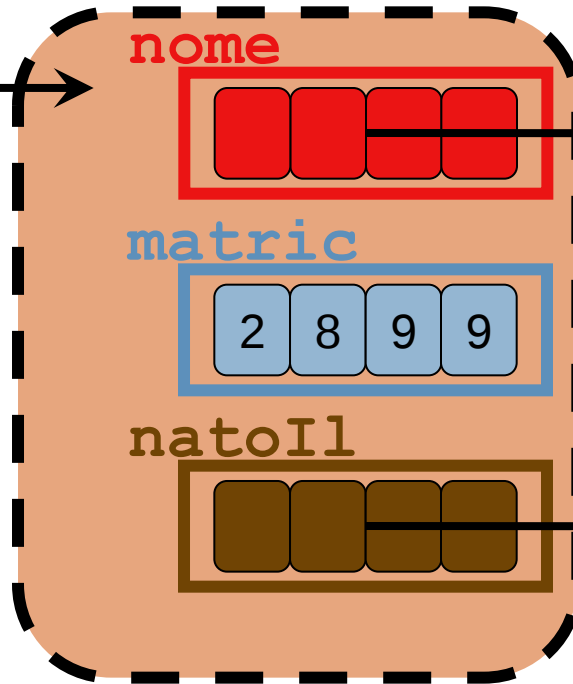
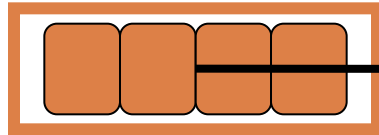
Pubblico e privato

- La classe `Date` (si vedano le slide prima) non è immutabile: esiste un metodo `setGiorno`, e magari anche `setMese` e `setAnno`
- Quindi `getNatoil()` viola l'incapsulamento!
 - ▣ Il riferimento restituito può essere modificato da chiunque (usando un metodo pubblico `set`) e cambiando lo stato dell'oggetto!

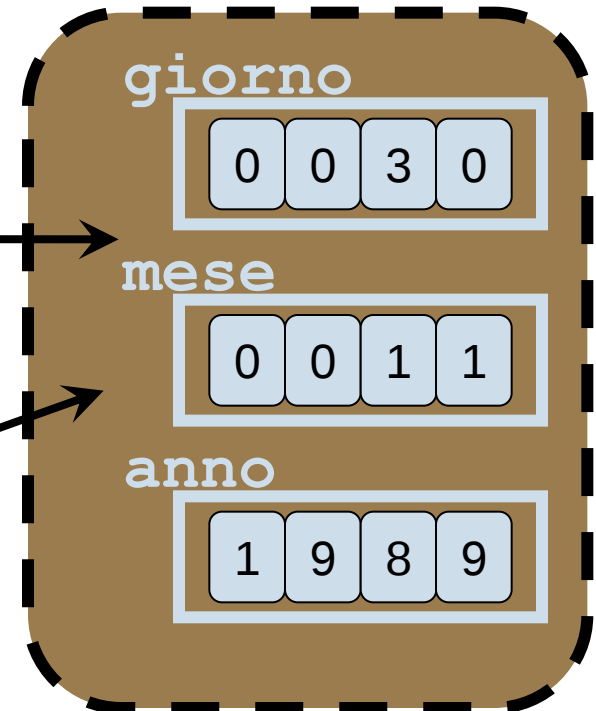
```
Studente paolo = ...;  
Date d = paolo.getNatoil();  
d.setGiorno(2); //cambia lo stato di paolo!
```

Pubblico e privato

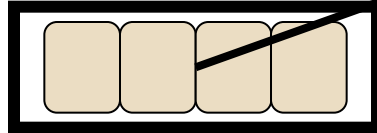
paolo



"Paolo Neri"



d



Pubblico e privato

- Come risultato, nonostante la protezione desiderata sul campo `natoI1` (che è **private**) siamo in grado di modificarlo.
- Bisogna cambiare il metodo **get**: occorre **clonare** l'oggetto prima di restituirlo. A tale scopo, usiamo il metodo `clone()`

```
public Date getNatoI1()  
{  
    return (Date) natoI1.clone();  
}
```

Pubblico e privato

- Il metodo `clone()` crea una copia dell'oggetto con gli stessi valori. Così non restituiamo un riferimento all'oggetto, ma a un suo clone.
 - ▣ Un metodo di accesso (`get`) non dovrebbe restituire un riferimento a un oggetto non immutabile: è opportuno clonare il campo.
- Il metodo `getName()` invece è sicuro perché il tipo `String` è immutabile.
- Per rendere immutabili i campi di una classe si può anche usare `final`.

Pubblico e privato

- Con `final` il valore del campo è una costante.
 - è consentito non inizializzarlo subito (*blank final*).

```
class Studente
{ //campi
  private final Date natoIl;
}
```

- Bisogna fare attenzione a non usare più di un assegnamento, per non ottenere errori.
 - Questo genere di errore è più propriamente un'eccezione (che Java è in grado di gestire).

Campi statici

- Talvolta è utile descrivere una proprietà di una classe che non fa riferimento a uno specifico oggetto istanza di quella classe.
 - ▣ Esempio: vogliamo che gli studenti abbiano un numero di matricola progressivo: 1 al primo studente, 2 al secondo e così via.
 - ▣ Potremmo richiedere un campo **nextMatric** dove memorizziamo il prossimo valore da assegnare al **matric** di un oggetto istanza

Campi statici

- Questo valore potrebbe essere memorizzato all'interno dei singoli oggetti: inefficiente
 - ▣ In effetti non è un'informazione di un oggetto, ma comune a tutta la classe
 - ▣ Meglio avere un campo comune a tutta la classe.
- Java consente di definire **campi statici**.
- Essi sono associati all'intera classe, non ai singoli oggetti.

Campi statici

- I campi statici sono anche detti variabili di classe (in contrapposizione alle variabili d'istanza, che sono cioè campi non-statici, perché sono associati a specifici oggetti).
- Quindi **4** casi per una variabile in un metodo:
 - ▣ variabili di classe
 - ▣ variabili d'istanza
 - ▣ parametri
 - ▣ variabili locali

Campi statici

- I campi statici appartengono all'intera classe
 - ▣ Anche se non è stato istanziato nessun oggetto della classe, i campi statici sono già inizializzati e sempre disponibili
 - ▣ Indipendentemente dal numero di oggetti-istanza creati per quella classe, esiste esattamente una copia di ogni variabile di classe.
 - ▣ Le variabili di classe vengono creati dalla JVM quando viene eseguita la prima istruzione che contiene un riferimento a quella classe.

Campi statici

- Come rivela il nome (dovuto a ragioni storiche), i campi statici si definiscono usando **static**:

```
class Studente
{
    //campi
    public static int nextMatric;
    private String nome;
    private int matric;
    private Date natoIl;

    //metodi
    ...
}
```

Campi statici

- Tutti i campi statici di una classe vengono inizializzati assieme al valore di default prima che un qualunque altro campo statico della classe venga utilizzato e prima che qualunque metodo della classe venga utilizzato.

Campi statici

- I campi statici possono essere tanto **public** quanto **private**.
- Se sono pubblici, possono essere acceduti fuori dal codice della classe usando il nome della classe come destinatario del messaggio.

`nome_classe.nome_campo_statico`

- ▣ Ad esempio:

`Studente.nextMatric = 500;`

Blocchi statici

- In effetti Java consente di definire diverse cose come “static” (termine mutuato dal C++)
 - ▣ campi statici
 - ▣ metodi statici
 - ▣ anche porzioni di codice statiche
- Le porzioni di codice statiche sono blocchi con il solo modificatore `static` (senza tipo di ritorno), usati tipicamente per l’inizializzazione.

Blocchi statici

- Si parla in questo caso di **blocchi di inizializzazione statica**. Sono usati quando non bastano le dichiarazioni dei campi per inizializzare la classe.

```
public class Crazy
{ public static double x;
  static
  { x = Math.log(5.0) ;
    for (int j=0;j<9;j++)
      { x = Math.log(x) ;
      }
  }
}
```

Diagram illustrating the static initialization block:

- The variable `x` is circled, indicating it is the target of the initialization.
- An arrow points from the text "voglio che parta da" (I want it to start from) to the `x` variable.
- The expression `log(log(log(.. (5) .. ) ) )` is shown, with a bracket underneath indicating it is repeated "10 volte" (10 times).

Blocchi statici

- Problema: cosa succede se un inizializzatore statico di una classe X invoca un metodo di una classe Y, e a sua volta l'inizializzatore statico di Y invoca un metodo della classe X?
 - ▣ Non è possibile assicurare che questa inizializzazione ciclica venga individuata durante la compilazione. Il programma potrebbe entrare in un circolo vizioso.

Blocchi statici

- In generale, è bene limitare il più possibile tutto quanto è **static**.
 - ▣ I metodi statici vanno introdotti solo quando è necessario (tipicamente: il **main**, e metodi che agiscono su numeri, che non sono oggetti)
 - ▣ I campi statici vanno usati con cautela
 - ▣ I blocchi di inizializzazione statici possono creare problemi di loop infinito.

Costruttori

- I costruttori specificano come un oggetto di quella classe viene creato e inizializzato.
 - ▣ Potrebbero sembrare metodi, perché hanno le parentesi tonde nel nome. Tuttavia, in Java, i costruttori hanno caratteristiche speciali e sono considerati un caso separato dai metodi.
- Prima dell'intervento di un costruttore, ogni variabile istanza (tipo riferimento) di una classe viene inizializzata a **null** e non c'è area di memoria contenente i valori dei campi.

Costruttori

- Nel momento in cui un nuovo oggetto viene istanziato, il costruttore:
 - ▣ alloca la memoria necessaria all'oggetto
 - ▣ decide come inizializzarne i campi
 - ▣ prende riferimento della locazione di memoria dove l'oggetto è stato collocato perché possa essere collegato alla variabile riferimento con cui accedere l'oggetto (che già esiste, ma è `null`)

Costruttori

- Alla stregua dei metodi, i costruttori sono caratterizzati da:
 - ▣ un nome
 - ▣ un elenco di parametri
 - ▣ un modificatore d'accesso (tipicamente i costruttori sono **public**)
 - ▣ mentre non serve specificare un tipo di ritorno: è per forza un oggetto di quella classe! (differenza tra costruttori e metodi veri e propri)

Costruttori

- I costruttori hanno sempre lo stesso nome della classe (però hanno le parentesi)
 - ▣ **Date ()** è un costruttore della classe **Date**.
- Tuttavia, è possibile (e frequente) avere più di un costruttore: dato che il nome è per forza lo stesso, sarà diverso l'insieme dei parametri
 - ▣ Questa prassi è nota come *overloading* (sovraccarico) dei costruttori, e consente di usare diverse modalità di inizializzazione

Costruttori

- Ad esempio, la classe **Date** deve essere inizializzata per i campi **giorno**, **mese**, **anno**.
- Potrei decidere le seguenti modalità:
 - ▣ Se invoco il costruttore con tre parametri di tipo **int**, assegnali nell'ordine a **giorno**, **mese**, **anno**.
 - ▣ Se invoco il costruttore con un parametro solo di tipo **int**, lo assegno ad anno e setto sia giorno e mese a 1.
 - ▣ Se invoco il costruttore senza parametri, setto **giorno**, **mese**, **anno** a 1,1,1900.

Costruttori

```
class Date
{
    //campi
    private int giorno, mese, anno;

    //costruttori
    public Date(int x, int y, int z)
    { this.giorno = x; this.mese = y;
      this.anno = z;
    }
    public Date(int z)
    { this.giorno = 1; this.mese = 1;
      this.anno = z;
    }
    public Date()
    { this.giorno = 1; this.mese = 1;
      this.anno = 1900;
    }
}
```

campi privati

costruttori

Costruttori

```
class Date
{
    //campi
    private int giorno, mese, anno;

    //costruttori
    public Date(int x, int y, int z)
    { giorno = x; mese = y; anno = z;
    }
    public Date(int z)
    { giorno = 1; mese = 1; anno = z;
    }
    public Date()
    { giorno = 1; mese = 1; anno = 1900;
    }
}
```



si può omettere **this**,
perché non ci sono ambiguità

Costruttori

```
class Date
{
    //campi
    private int giorno, mese, anno;

    //costruttori
    public Date(int giorno, int mese, int anno)
    { this.giorno = giorno; this.mese = mese;
      this.anno = anno;
    }

    public Date(int anno)
    { giorno = 1; mese = 1; this.anno = anno;
    }

    public Date()
    { giorno = 1; mese = 1; anno = 1900;
    }
}
```

costrutto valido (e frequente)
può essere comodo, ma richiede
this per risolvere l'ambiguità

Costruttori

- In effetti, i costruttori possono anche invocarsi a vicenda. Per esempio, il costruttore senza parametri `Date ()` potrebbe semplicemente invocare il costruttore a tre parametri con modalità `Date (1 , 1 , 1900)` .
- Questa modalità è ammessa e detta cross-call tra costruttori. Tuttavia, l'istruzione di cross-call deve essere la prima riga di codice del costruttore chiamante.

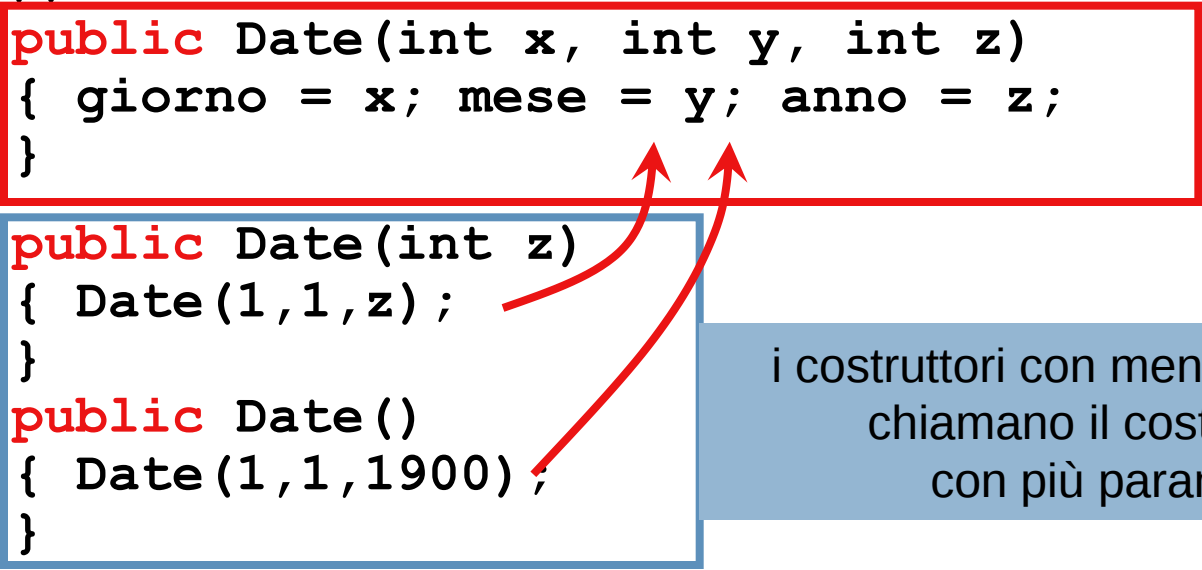
Costruttori

```
class Date
{
    //campi
    private int giorno, mese, anno;

    //costruttori
    public Date(int x, int y, int z)
    { giorno = x; mese = y; anno = z;
    }

    public Date(int z)
    { Date(1,1,z);
    }

    public Date()
    { Date(1,1,1900);
    }
}
```



i costruttori con meno parametri
chiamano il costruttore
con più parametri

Costruttori

```
class Date
{
    //campi
    private int giorno, mese, anno;

    //costruttori
    public Date(int x, int y, int z)
    { giorno = x; mese = y; anno = z;
    };

    public Date(int z)
    { this(1,1,z);
    };
    public Date()
    { this(1,1,1900);
    };
}
```

notazione abbreviata:
this vale Date

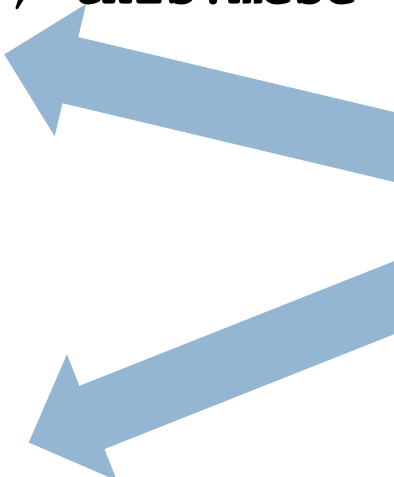
Costruttori

```
class Date
{
    //campi
    private int giorno, mese, anno;

    //costruttori
    public Date(int giorno, int mese, int anno)
    { this.giorno = giorno; this.mese = mese;
      this.anno = anno;
    }

    public Date(int anno)
    { this(1,1,anno);
    }

    public Date()
    { this(1,1,1900);
    }
}
```



da notare l'uso
di **this** con due
significati del
tutto differenti

Costruttori

- Quando il nome del costruttore è sovraccarico, il compilatore decide quale usare sulla base dei parametri. Ad esempio:
 - ▣ se chiamo **Date** (5 , 5 , 2005) viene usato il costruttore con tre parametri di tipo **int**
 - ▣ se chiamo **Date** () viene usato il costruttore senza parametri
 - ▣ se chiamo **Date** ("oggi") viene restituito errore, perché non esiste nessun costruttore che ha un solo parametro di tipo **String**.

Costruttori

- Java fornisce anche un costruttore di default.
- Non fa nulla e ha la lista dei parametri vuota.
 - ▣ in pratica, è come fosse:
 - ▣ è public solo se lo è la sua classe
- Questo consente di avere un costruttore, se il programmatore si dimentica di scriverne uno
- Se però viene scritto un costruttore qualsiasi, questo sovrascrive quello di default.

```
public Date()  
{  
}
```

Costruttori

- Tuttavia, dato che in Java i costruttori non sono considerati metodi, non possono essere invocati su oggetti esistenti.
 - ▣ Per esempio questo codice non è ammesso.

```
public class Date
{
    private int giorno, mese, anno;
    public Date(int x, int y, int z) ...
}

Date anniversario;
anniversario.Date(12, 10, 2003); //Errore!
```

Costruttori

- I costruttori vanno invece invocati facendoli precedere dall'operatore **new**.
 - ▣ In questo modo si distingue meglio l'occorrenza del nome come identificativo della classe e come identificativo del costruttore
 - ▣ Il ruolo di **new** è quello di riservare memoria per l'oggetto (mentre il costruttore poi lo inizializza). Il valore di **new** è quello del riferimento creato.

Costruttori

- ❑ È corretto quindi scrivere:
`new Date (12,10,2003) ;`
- ❑ Solo, in questo modo alloco l'oggetto ma mi perdo il suo riferimento. Conviene quindi salvare il riferimento appena creato da **new** in una variabile riferimento.

```
Date anniversario = new Date (12,10,2003) ;
```



tipico modo di usare un costruttore

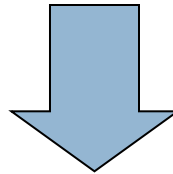
Costruttori

- Di solito conviene che il costruttore sia dichiarato come **public**, in modo che chiunque possa creare un'istanza della classe.
 - ▣ Ad esempio, si può settare che **Anniversario** è il 12/10/2003 in ogni punto del codice.
- Tuttavia, ci sono casi in cui i costruttori sono essere dichiarati privati, per restringere le possibilità di creazione degli oggetti.
 - ▣ esistono anche costruttori **protected** (possono essere usati solo dalle sottoclassi)

Costruttori

- È facile dimenticarsi di invocare il costruttore

```
Date anniversario;  
anniversario.setGiorno(12);  
//ERRORE: anniversario non è inizializzato
```



```
Date anniversario = new Date();  
anniversario.setGiorno(12);
```

Costruttori

- ❑ Non si possono invocare costruttori su oggetti esistenti:

```
Date mondiali = new Date(2010);  
mondiali.Date(2014);  
//ERRORE: non si può ricostruire
```



- ❑ Però si possono costruire oggetti nuovi che sovrascrivano quelli esistenti.

```
Date mondiali = new Date(2010);  
mondiali = new Date(2014); //AMMESSO
```



Costruttori

- Non è detto che il costruttore usato insieme a **new** salvi poi il risultato in una variabile.

- È possibile anche passare il valore risultante come parametro, senza memorizzarlo:

```
luca.setNatoIl(new Date(12,3,1989));
```

imposta il campo `natoIl` alla data richiesta, senza creare una variabile allo scopo. È come

```
Date temp = new Date(12,3,1989);
```

```
luca.setNatoIl(temp);
```