

# Classi, campi e metodi



# Classi, campi e metodi

- Template da cui vengono creati gli oggetti
- Definiscono dati e funzionalità di un oggetto
- Definiscono un tipo



# Classi, campi e metodi

- **Campi:**
  - dati e informazioni
  - rappresentano lo stato dell'oggetto
- **Metodi:**
  - funzionalità
  - possono modificare lo stato dell'oggetto



# Campi, variabili e parametri


- **Campi:** dichiarati all'interno della classe, sono salvati nell'heap e quindi contenuti da ogni oggetto. Si accedono dereferenziando l'oggetto
- **Variabili:** definite nel corpo di un metodo e allocate nel suo stack. Sono rimosse alla fine della sua esecuzione
- **Parametri:** dichiarati nella firma di un metodo e utilizzati per passargli informazioni

# Esecuzione di un programma

- Non esiste un entrypoint singolo, ogni classe può contenere un main
- Bisogna specificare la classe da cui eseguire il codice

Argomenti opzionali che possono essere passati dalla linea di comando quando si esegue il programma

```
public class Classe{  
    ...  
    public static void main(String[] args){  
        ...  
    }  
}
```

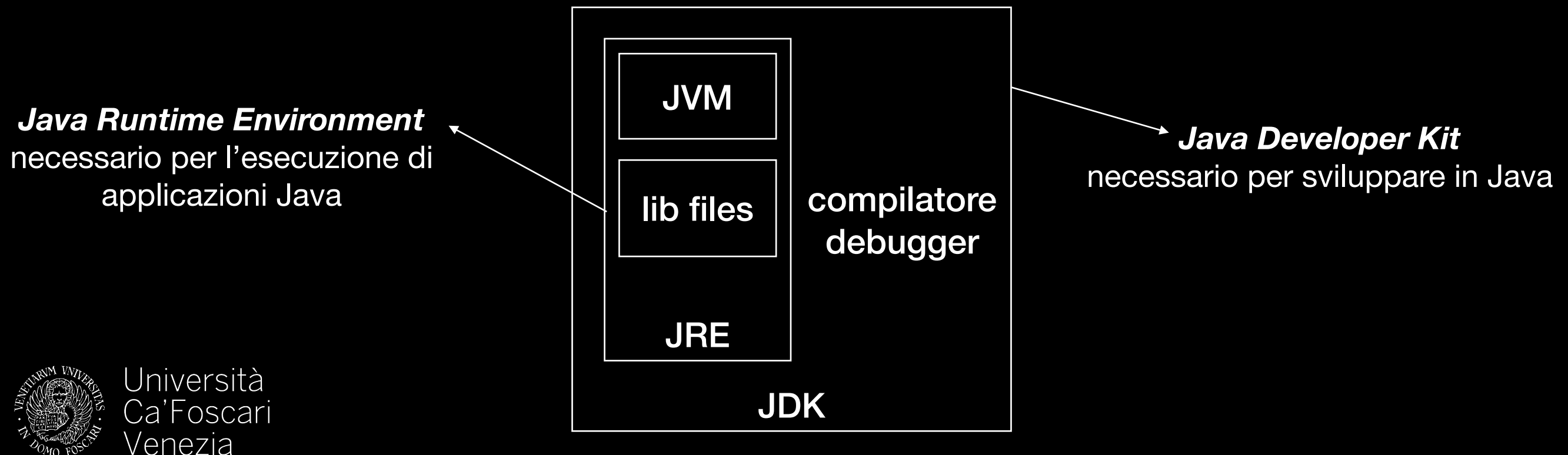


*All'interno della cartella in cui si trova la classe*  
`java Classe argomento1 argomento2 ...`



# Esecuzione di un programma

- Il codice è compilato in bytecode (linguaggio simile al linguaggio macchina, ma indipendente dal SO)
- Il risultato sono file `.class` o `.jar`
- La JVM, specifica per il SO, converte il bytecode in codice macchina ed esegue il programma



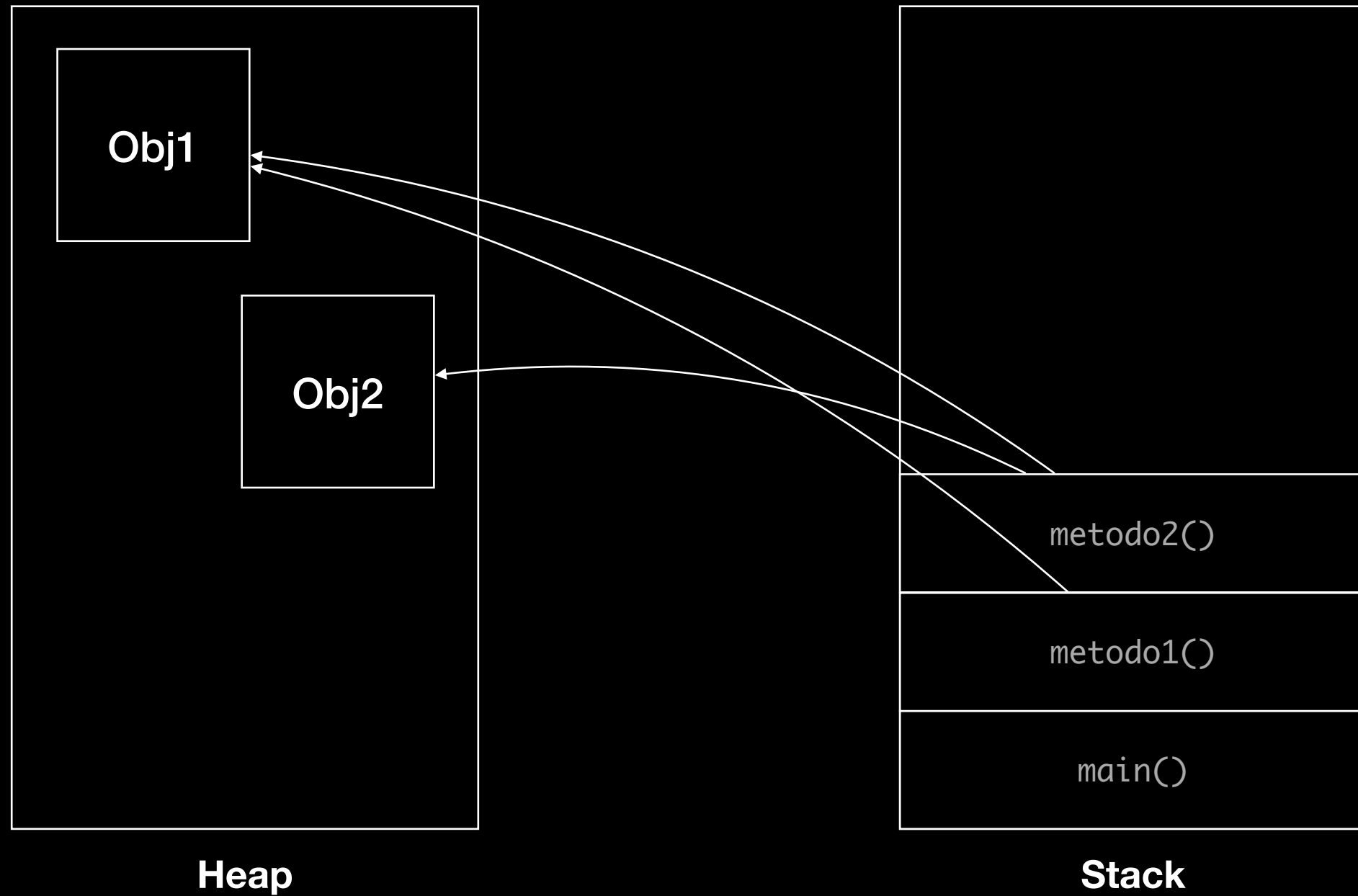
# Bytecode

- Linguaggio intermedio tra Assembly e Java, indipendente dalla macchina
- Lo stato di esecuzione è composto da uno stack di frame (per metodo) e la memoria che contiene gli oggetti
- Ogni frame contiene le variabili locali del metodo e uno stack degli operandi



```
void method(int y, int z){  
→ int x = y + z  
  System.out.println(x)  
}  
...  
method(10, 11)
```

# Memoria

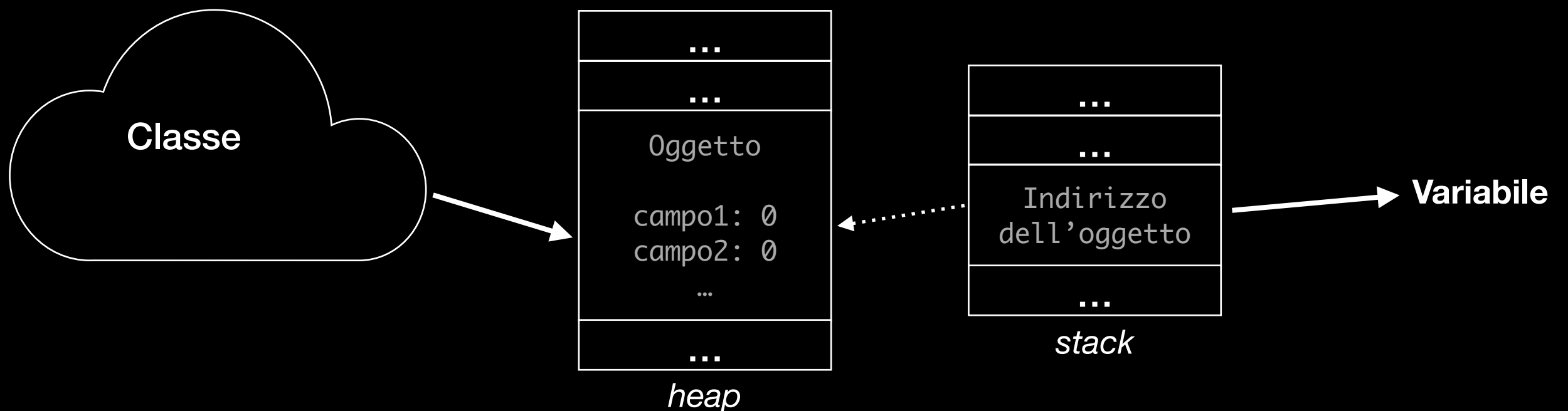




# Keyword new

Tipo Variabile = new Classe()

1. Alloca la memoria all'interno dell'heap
2. Inizializza i campi ai loro valori di default (*0, null*)
3. Ritorna il puntatore all'oggetto creato



# Costruttore

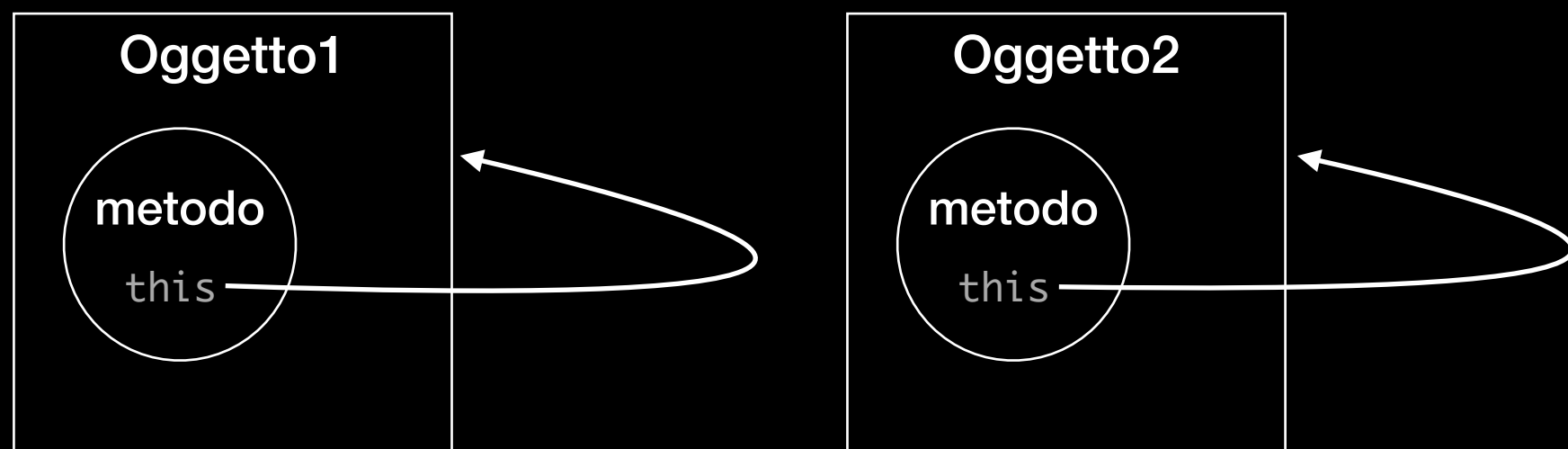
- Metodo eseguito dopo l'inizializzazione dei campi quando si istanzia un nuovo oggetto
- Se non definito, il compilatore ne aggiunge uno di default (Costruttore implicito)

```
class Classe{  
    Classe(paramList1){  
        ...  
    }  
  
    Classe(paramList2){  
        ...  
    }  
}
```

*La firma dei costruttori deve essere diversa  
(la lista ordinata dei tipi dei parametri deve essere unica)*

# Keyword `this`

- È un puntatore all'oggetto corrente
- Utilizzata per:
  - accederne ai campi e i metodi (buona practice usarla sempre)
  - passarne la reference ad un altro metodo
  - invocarne il costruttore da un altro costruttore (solo come prima istruzione)



# Modificatori

- Specificano comportamenti aggiuntivi di campi, metodi e classi

	Classe	Campo	Metodo
<i>accesso</i>	✓	✓	✓
<i>static</i>	✗	✓	✓
<i>final</i>	✓	✓	✓
<i>abstract</i>	✓	✗	✓

**Modificatori di accesso:** `public`, `protected`, `private`, `<default>`

# static

- Precede i campi il cui valore è condiviso tra tutte le istanze della classe
- I metodi statici possono accedere solamente a campi e altri metodi statici (in quanto non appartenenti ad un'istanza specifica)
- Il costruttore statico può inizializzare i campi statici ed è invocato all'inizio dell'esecuzione del programma
- Acceduti facendo:

`Classe.campo`

`Classe.metodo(params)`

*Si può accedere anche tramite `this` o un'istanza,  
ma è una bad practice*



# Campi final

- Non possono essere modificati dopo l'inizializzazione
- Possono essere assegnati dal costruttore se non è specificato un valore nella dichiarazione

```
classe Classe{  
    final int campo = 0;  
}
```



```
classe Classe{  
    final int campo;  
  
    Classe(int a){  
        campo = a;  
    }  
}
```



```
classe Classe{  
    final int campo = 0;  
  
    Classe(int a){  
        campo = a;  
    }  
}
```



# Tipi di tipi

- **Valore:**
  - contengono il valore concreto
  - *int, long, float, double, boolean, char*
- **Riferimento**
  - contengono un puntatore
  - *array e oggetti*

**Aliasing:** poiché gli oggetti sono salvati per riferimento, nel caso lo stesso oggetto fosse salvato in due variabili differenti (nomi), modificarlo utilizzando un nome rifletterà tale modifica anche sull'altro



# Garbage collection

- Si può accedere all'heap solamente dereferenziando oggetti (a differenza di C)
- JRE tiene traccia di cosa è raggiungibile, e nel caso negativo, il **garbage collector** dealloca tale parte di memoria
- A causa del GC, la velocità di esecuzione può variare molto in programmi che allocano molta memoria

