



JAVA: Definizione classi

- **Definizione di classi, metodi**
- **Creazione di un oggetto**
- **Creazione delle classe Name**

Definizione di una classe

- Sintassi più elementare

```
SpecificatoreAccesso class NomeClasse {  
    // Definizione attributi e metodi  
}
```

- Contiene la definizione degli attributi
- Contiene i suoi metodi
 - Descrivono il comportamento comune alle istanze della classe
- Le parentesi graffe aperte e chiuse fungono da delimitatori del contenuto di una classe

Definizione di un metodo

- **Definizione del body**
 - Prototipo e corpo
- **Le parentesi graffe aperte e chiuse fungono da delimitatori del corpo del metodo**

```
SpecificatoreAccesso TipoDiRitorno nomeDelMetodo {  
    // Corpo del metodo  
}
```

Invocazione di un metodo

■ Forma generale

```
RiferimentoOggetto.nomeMetodo(argomenti);
```

■ Esempio

```
String x = "Rocco";  
System.out.println(x.length());
```

■ Due tipologie di argomenti

- Argomenti **impliciti**: Riportati nella dichiarazione
- Argomenti **espliciti**: Ricevente del messaggio

[Specificatori di accesso]

- Applicabili a classi, metodi e attributi
- Tre tipi di specificatori
 - **public**
 - Specificatore di default
 - Tutti hanno accesso
 - **private**
 - Accesso consentito solo all'interno della classe
 - **protected**
 - Accesso consentito solo all'interno del package

[L'istruzione return]

■ Problema

- Metodi che devono restituire qualcosa al mittente

■ Istruzione return

- **return value**
 - Permette ad un metodo di restituire un valore
 - Valore costante o variabile
- Una volta eseguita l'istruzione **return** il metodo termina e il mittente riprende l'esecuzione

[Metodi getter e setter]

- **Utilizzati per ottenere e settare il valore degli attributi di un oggetto**
 - Incapsulamento
 - Protezione degli attributi della classe
- **Vantaggi**
 - Modificare l'implementazione interna comporta solo la modifica dei metodi della classe
 - Controllo degli errori
 - Vietato accesso incondizionato ai dati
- **Attenzione ai riferimenti...**

[Metodi particolari: i costruttori - 1]

■ Costruttori

- Metodi speciali presenti in ogni classe
- Ogni classe ha uno o più costruttori
- Hanno lo stesso nome della classe
- Sono invocati per creare un nuovo oggetto
- Restituiscono un riferimento al nuovo oggetto creato
- Possibile l'overloading

```
SpecificatoreAccesso NomeDellaClasse (argomenti) {  
    // Corpo  
}
```

[Metodi particolari: i costruttori - 2]

■ Per fissare le idee...

- La creazione di un oggetto della classe A comporta l'invocazione di uno dei costruttori di A

■ Problema

- Per invocare un metodo è necessario inviare un messaggio ad un oggetto
- Non esiste nessun oggetto cui inviare il messaggio di invocazione del costruttore

[L'operatore new]

- Utilizzato per creare nuovi oggetti
- Forma generale

```
new NomeDelCostruttore (argomenti);
```

- Esempio

```
String s = new String("Hello world");
```

- Viene ritornato un riferimento al nuovo oggetto **String**

[Metodi per creare un oggetto]

- **Esistono due modi per creare un oggetto**
 - Attraverso l'uso dell'operatore **new**
 - Attraverso l'invocazione di un metodo che ha un valore di restituzione che è un riferimento ad un nuovo oggetto
- **E' facile osservare che il secondo sfrutta il primo**
 - All'interno del metodo si crea un nuovo oggetto con **new**
 - Viene restituito l'oggetto appena creato

Inizializzazione degli attributi

- **Overloading dei costruttori**
 - Inizializzazione diversa degli attributi
- **E' fortemente consigliato inizializzare gli attributi**
 - Inizializzazione automatica
 - Non è un buon modo di programmare
- **Inizializzazione costante di un attributo**
 - Tutti i costruttori lo inizializzano allo stesso modo

```
public class FirstSample {  
    int field = 0;  
    // Definizione metodi  
}
```

Costruttori predefiniti

■ Costruttore senza argomenti

- Se una classe non ha nessun costruttore, JAVA crea un costruttore predefinito
 - Inizializzazione automatica degli attributi
 - Programmatori C++
 - In JAVA non esistono gli elenchi di inizializzazione

```
Customer :: Customer(String n)
    : name(n),
      accountNumber(Account::getNewNumber()) {
}
```

La parola chiave **this**

- **Permette di accedere all'oggetto stesso**
 - Aumentare la leggibilità del codice
 - Simile al puntatore **this** di C++
- **Utilizzato anche all'interno dei costruttori**
 - Combina il codice comune tra i costruttori

```
public class FirstSample {  
    public FirstSample(argomento1) {  
        this(argomento1, argomento2);  
        .....  
    }  
    public FirstSample(argomento1, argomento2) {  
        .....  
    }  
}
```

[La parola chiave null]

- Molti metodi restituiscono un riferimento ad un oggetto
- Occasionalmente vorremmo restituire un'indicazione che specifichi che non è possibile restituire alcun oggetto.
- In tale situazione usiamo la keyword null
 - Riferimento che non fa riferimento ad alcun oggetto
 - Assegnato a variabili di riferimento di ogni classe
 - E' anche possibile verificare che una variabile contenga null
 - `s == null`

Blocchi di inizializzazione

- **Per inizializzare gli attributi**
 - Impostiamo il valore nel costruttore
 - Assegniamo un valore all'atto della dichiarazione
- **Blocco di inizializzazione**
 - Eseguito ogni volta che si crea un oggetto
 - Utile per inizializzare le variabili statiche...

```
public class FirstSample {  
    .....  
    {  
        field = 0;  
    }  
    .....  
}
```

[Variabili e oggetti: ciclo di vita]

■ Periodo di vita di una variabile

- Parametri e variabili locali vivono solo durante l'esecuzione di un metodo
- Le variabili di istanza hanno lo stesso periodo di vita dell'oggetto cui appartengono

■ Periodo di vita di un oggetto

- Esiste fino a quando c'è una variabile di riferimento che si riferisce ad esso
- La distruzione è automatica
 - Garbage collection

[Distruzione di oggetti]

- **Distuggere gli oggetti**
 - Ripulire la memoria da loro occupata
- **In JAVA esiste la garbage collection**
 - Un oggetto viene distrutto quando non è referenziato da alcuna variabile
- **Utilizzo del metodo finalize**
 - Eseguito prima di distruggere l'oggetto
 - Libera eventuali risorse occupate dall'oggetto
 - Non si sa precisamente quando viene invocato
 - E' conveniente definire un metodo di rilascio esplicito

Attributi e metodi statici - 1

- **I campi statici non cambiano da un'istanza all'altra**
 - Appartengono alla classe
 - Utilizzati per creare costanti di classi
 - Esempio
 - Attributo **out** della classe **System**
 - Attributo **PI** della classe **Math**
- **I metodi statici non agiscono su istanze di classe**
 - Non è possibile accedere agli attributi/metodi non statici
 - Esempio
 - Metodo **pow** della classe **Math**
 - Metodo **main**

Attributi e metodi statici - 2

■ Utilizzo degli attributi e dei metodi statici

```
// Accesso ad un metodo statico
NomeClasse.metodoStatico(parametri);

// Accesso ad un campo statico
NomeClasse.nomeCampoStatico;
```

■ Inizializzazione degli attributi statici

- Indicazione del valore iniziale
- Blocco di inizializzazione statico
 - Necessario quando l'inizializzazione non rientra in un'espressione

[Un approccio metodologico]

■ I fase: Specifica della classe

- Decidere il comportamento che la classe dovrà fornire
 - Identificare i metodi da fornire
- Stabilire in che modo la classe verrà usata
 - Definire i prototipi dei metodi
- Scrivere un programma di esempio che utilizza la classe
- Scrivere lo scheletro della classe
 - Prototipi e corpi vuoti

■ II fase: Implementazione della classe

[La classe Name - 1]

■ Obiettivo

- Scrivere una classe, **Name**, che modelli il nome di una persona

■ Comportamento

- Aggiungere o sostituire un titolo
 - Dott., Dott.ssa, Sig.na, Sig., Sig.ra
- Ottenere le iniziali come oggetto String
- Ottenere il nome come oggetto String
 - Formato: *cognome, nome*
- Ottenere il nome come oggetto String
 - Formato: *titolo nome cognome*

[La classe Name - 2]

■ Costruttore

- Name(String first, String last)

■ Metodi

- String getInitials()
 - Iniziali del nome
- String getLastFirst()
 - *cognome, nome*
- String getFirstLast()
 - *titolo nome cognome*
- void setTitle(String title)

[La classe Name - 3]

■ Scheletro della classe

```
public class Name {  
    public Name(String pFirst, String pLast) {  
    }  
  
    public String getInitials() {  
    }  
  
    public String getLastFirst() {  
    }  
  
    public String getFirstLast() {  
    }  
  
    public void setTitle(String newTitle) {  
    }  
}
```

[La classe Name - 4]

■ Implementazione

- Nome e cognome sono passati come parametri al costruttore
- I metodi `getFirstLast` e `getLastFirst` hanno entrambi bisogno di accedere a nome e cognome
- Il titolo deve avere un ciclo di vita più lungo dell'esecuzione del metodo `setTitle`
 - `getFirstLast` accede anche al titolo
- E' necessario introdurre tre variabili di istanze
 - `firstName`
 - `lastName`
 - `title`

[La classe Name - 5]

- Il costruttore e il metodo setTitle...

```
public Name(String first, String last) {
    this.firstName = first;
    this.lastName = last;
    this.title = "";
}

public void setTitle(String newTitle) {
    this.title = newTitle;
}
```

[La classe Name - 6]

- Il metodo getInitials...

```
public String getInitials() {  
    String s;  
    s = firstName.substring(0,1);  
    s = s.concat(". ");  
    s = s.concat(lastName.substring(0,1));  
    s = s.concat(".");  
    return s;  
}
```

[La classe Name - 7]

- I metodi get...

```
public String getLastFirst() {  
    return lastName.concat(", ").concat(firstName);  
}  
  
public String getFirstLast() {  
    return title.concat(" ").concat(firstName)  
        .concat(" ").concat(lastName);  
}
```

[Stato e comportamento]

■ Comportamento

- Definito dai metodi, è comune a tutti gli oggetti della classe

■ Stato

- Definito dalle variabili di istanza, è proprio dell'oggetto

■ Esempio

```
Name n1, n2;  
n1 = new Name("Tom", "Petty");  
n2 = new Name("Benny", "Roger");
```

- Gli oggetti **name1** e **name2** hanno lo stesso comportamento ma stati diversi
 - I valori delle variabili di istanza di ogni oggetto sono distinti

[Comportamento e responsabilità]

- **Determinare le responsabilità di una classe è un ottimo punto di partenza per definirne il comportamento**
 - Le classi migliori sono quelle che si rendono responsabili dei task che le coinvolgono
- **Esempio**
 - Estrarre una sottostringa da una stringa
- **Esempio classico**
 - Ogni classe dovrebbe essere responsabile di leggere e scrivere i propri oggetti

[Suggerimenti - 1]

- **Impostare sempre dati privati**
 - Non violare mai l'incapsulamento
 - La rappresentazione varia più spesso dell'impiego
 - Modifiche alla rappresentazione non devono impattare sulle attività dell'utente della classe
- **Inizializzare sempre i dati**
 - Inizializzazione automatica degli oggetti e non delle variabili locali
 - Inizializzare esplicitamente le variabili

[Suggerimenti - 2]

- **Non utilizzare troppi tipi essenziali in una classe**
 - Creazione di una classe **Address** invece di campi singoli
- **Non tutti i campi necessitano di singoli metodi di accesso e di modifica**
 - Variabili gestite solo dall'oggetto
 - Esempio
 - Contatori degli accessi in un sito
- **Suddividere le classi con troppe responsabilità**
- **Assegnare alle classi e ai metodi nomi che ne riflettono le responsabilità**

[Suggerimenti - 3]

- **Utilizzare una forma standard per le definizioni delle classi**
 - Divisione in sezioni
 - Metodi pubblici
 - Metodi privati
 - Ordinare il contenuto delle sezioni
 - Costanti
 - Costruttori,
 - Metodi e metodi statici
 - Variabili e variabile statiche
- **Enfatizzare i metodi accessibili dall'esterno**

[Organizzare le classi: i package]

- **Necessario organizzare le classi correlate**

- Creazione di librerie di codice

```
package nomePackage;  
  
class .....
```

- **Possibilità di annidare i package**

- Utilizzo del "." per creare package annidati
 - Struttura simile alle directory
- Garantita l'unicità dei nomi dei package
 - E' prassi comune utilizzare il nome del proprio dominio al contrario

Utilizzo dei package

■ Accesso alle classi di un package

- Nome completo del package

```
nomePackage.nomeSottoPackage.NomeClasse nomeOggetto;  
  
nomeOggetto = new nomePackage.nomeSottoPackage.NomeClasse();
```

- Importare il package

- Possibile utilizzare il carattere jolly “*”

```
import nomePackage.nomeSottoPackage.NomeClasse;  
// import nomePackage.nomeSottoPackage.*;  
class NewClasse {  
    .....  
    NomeClasse nomeOggetto = new NomeClasse();  
    .....  
}
```