LE CLASSI

CLASSI

- La classe è l'elemento principale di un particolare stile di programmazione che cerca di impostare e risolvere i problemi cercando di imitare concettualmente la vita reale

OGGETTI

- Un oggetto del mondo reale, ad esempio un gatto, possiede delle *proprietà* (o stati), quali nome, età e colore, e presenta delle caratteristiche comportamentali, ad esempio dormire, mangiare e fare le fusa.
- Analogamente, nel mondo della programmazione orientata agli oggetti, gli oggetti presentano proprietà e comportamenti.
- Utilizzando le tecniche orientate agli oggetti significa affrontare la risoluzione di un problema affidandola ad un oggetto o attore che nel programma svolge la sua funzione specifica in maniera totalmente indipendente dal restodel programma.

ISTANZE E MEMBRI DI CLASSE

- La maggior parte delle classi definisce un modello per un tipo di oggetto sulla base del quale vengono costruiti gli oggetti concreti che utilizzerò nei miei programmi.
  - Tutte le caratteristiche e i comportamenti che appartengono a una classe sono detti *membri* di tale classe.
  - In particolare, le caratteristiche (nell'esempio del gatto, il nome, l'età e il colore) sono dette *proprietà* della classe e sono rappresentate da variabili, mentre i comportamenti (mangiare, dormire) sono detti *metodi* della classe e sono rappresentati da funzioni.

CREARE UN'ISTANZA DI UNA CLASSE

- Utilizzare questo tipo di classi significa:
  - creare un'istanza della classe stessa,
  - assegnare l'istanza creata ad una variabile e
  - usarne le proprietà e applicarne i metodi per ottenere i risultati desiderati.
- Per creare un'istanza di una classe devo applicare l'operatore ***new*** al ***costructor***, cioè alla funzione di costruzione della classe, una speciale funzione che ha lo stesso nome della classe.

ESEMPIO

```
// dichiaro “now” come variabile di tipo Date
var now:Date;
// assegno a now una istanza di Date che corrisponde alla
// data e all’ora corrente
now = new Date();
```

Dopo questa operazione now conterrà un’istanza della classe Date che rappresenta la data e l’ora corrente. A now si potranno applicare tutti i metodi e usare le proprietà della classe Date.

Da notare l’uso diverso di ***Date*** in ***var now:Date;*** e in ***now = new Date();***. Nel primo caso ***Date*** indica il ***tipo Date*** e il costrutto dice al compilatore che la variabile now potrà contenere solo oggetti di tipo Date. Nel secondo caso **Date()** (seguito da parentesi) indica la funzione di costruzione della classe Date e il costrutto ***now = new Date()*** crea una nuova istanza della classe Date() e la memorizza in now.

MEMBRI STATICI

- Ci sono delle classi che rappresentano un unico oggetto che il linguaggio crea automaticamente
- Si dice che queste classi hanno solo membri statici. Proprietà e comportamenti, cioè, che non si riferiscono a istanze della classe ma appunto ad un oggetto unico identificabile con la classe stessa.
- In questo caso proprietà e metodi si applicano direttamente alla classe senza bisogno di crearla.
- Questa classi in ActionScript sono: *Accessibility*, *Camera*, *ContextMenu*, *CustmActions*, *Key*, *Math*, *Microphone*, *Mouse*, *Selection*, *SharedObject*, *Stage*, *System*. In questi casi proprietà e metodi si applicano direttamente alla classe e vengono detti *statici*.

ESEMPIO

```
// ottengo le dimensioni della finestra in cui flash viene visualizzato
// utilizzando la classe Stage
var larghezza:Number;
var altezza:Number;
larghezza = Stage.width;
altezza = Stage.width;
```

Non è necessario creare un'istanza della classe ***Stage***. ***Stage*** è un oggetto già creato e posso accedere alle sue proprietà e ai suoi metodi senza nessuna operazione preliminare.

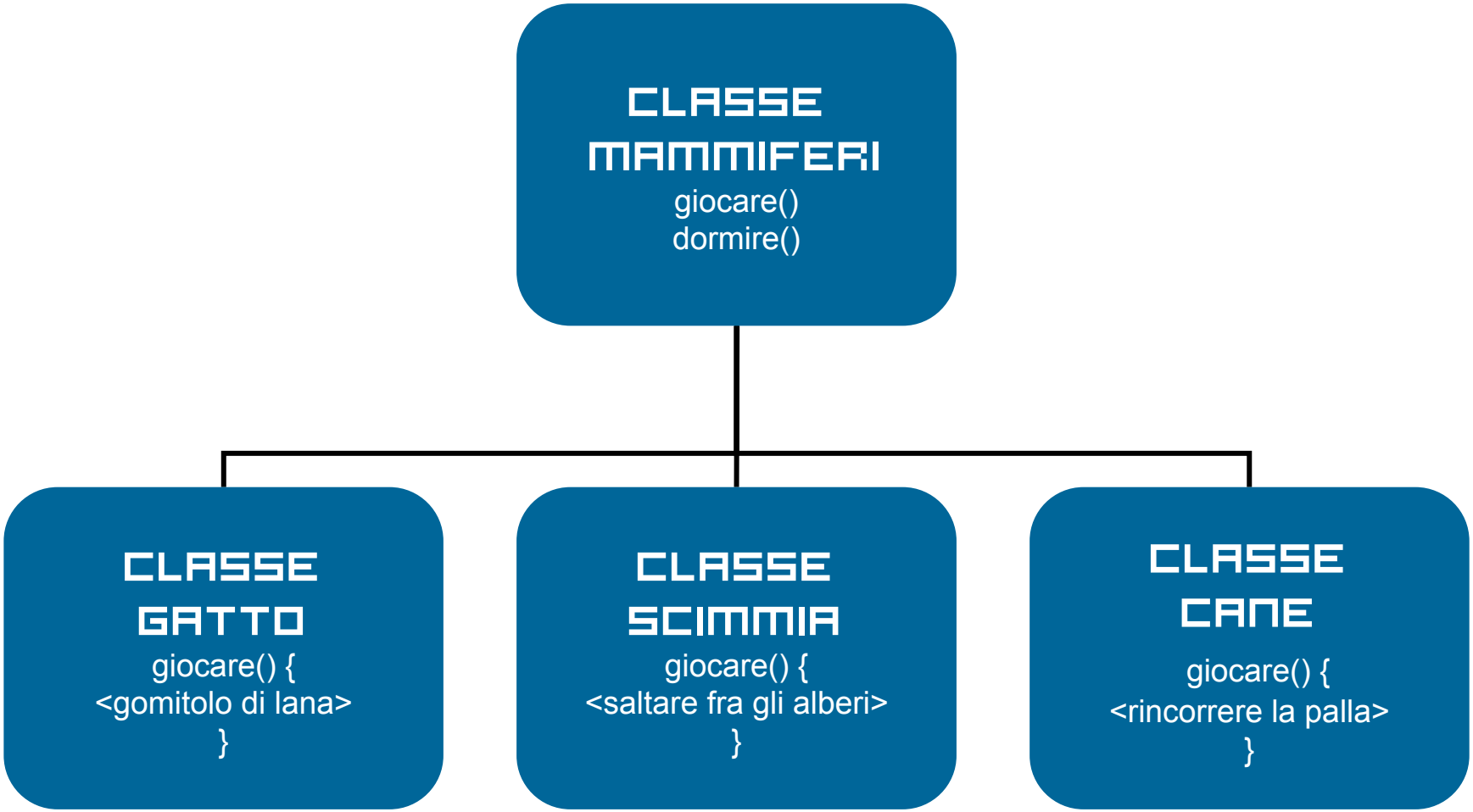
EREDITARIETÀ

- Uno dei vantaggi principali della programmazione orientata agli oggetti è rappresentato dalla creazione di **sottoclassi** (tramite estensione di una classe); una **sottoclasse** eredita tutte le proprietà e i metodi della rispettiva classe. La sottoclasse definisce generalmente ulteriori metodi e proprietà o sostituisce metodi o proprietà definiti nella superclasse. Le sottoclassi possono inoltre sostituire i metodi definiti in una superclasse, ovvero fornire definizioni proprie per tali metodi.

POLIMORFISMO

- La programmazione orientata agli oggetti permette di esprimere differenze tra le singole classi con l'ausilio di una tecnica detta ***polimorfismo***, grazie alla quale le classi possono sostituire i metodi delle relative superclassi e definire implementazioni specializzate di tali metodi. .

POLIMORFISMO



LE CLASSI INCORPORATE

- *ActionScript* comprende circa ***65*** classi incorporate che servono a gestire diversi tipi di elementi: tipi di dati di base quali Array, Boolean, Date, gestione degli errori, gestione degli eventi, caricamento di contenuto esterno (XML, immagini, dati binari originari, ecc.).

OBJECT

- Il tipo di dati ***Object*** rappresenta l'astrazione dell'idea di oggetto. Sta alla radice della gerarchia delle classi: tutte le classi sono estensioni di ***Object***.
- La classe *Object* mi consente di creare oggetti vuoti, senza proprietà né metodi.
- Creando un'istanza di *Object* creo un oggetto personalizzato, a cui posso aggiungere le proprietà che voglio e che mi può servire per organizzare le informazioni nell'applicazione Flash.

COME CREO UN'ISTANZA DI OBJECT

```
var user:Object = new Object();
user.name = "Irving";
user.age = 32;
user.phone = "555-1234";
```

1. Viene creato un nuovo oggetto denominato user utilizzando l'operatore ***new***. L'oggetto ha tre proprietà: name, age e phone che sono tipi di dati String e Numeric.
2. Lo stesso oggetto può essere creato anche assegnando alla variabile il letterale di tipo *Object* corrispondente.

```
var user:Object;
user = {name:”Irving”,age:32,phone:”555-1234”};
```

ARRAY ASSOCIATIVI

- Un array associativo è composto da chiavi e valori non ordinati e utilizza le chiavi alfanumeriche al posto degli indici numerici per organizzare i valori.
- Ogni chiave è una stringa univoca e viene utilizzata per accedere al valore a cui è associata. Il valore può essere un tipo di dati Number, Array, Object e così via.
- Array associativi e Object rappresentano due modi diversi per rappresentare gli stessi dati e sono intercambiabili.

```
// Definisce l'oggetto da utilizzare come array associativo
var someObj:Object = new Object();
// Definisce una serie di proprietà
someObj.myShape = "Rectangle";
someObj.myW = 480;
someObj.myH = 360;
someObj.myX = 100;
someObj.myY = 200;
someObj.myAlpha = 72;
someObj.myColor = 0xDFDFDF;
// Visualizza una proprietà utilizzando l'operatore punto e
// la sintassi di accesso agli array
trace(someObj.myAlpha); // 72
trace(someObj["myShape"]); // 72
```

ARRAY

- Un *array* è un oggetto le cui proprietà sono identificate da un numero (indice) che ne rappresenta la posizione nella struttura dell'array.
- Un array è un elenco di elementi recuperabile attraverso un indice.
- Non occorre che gli elementi dell'array abbiano lo stesso tipo di dati. È possibile inserire numeri, dati, stringhe, oggetti, array.

USO DEGLI ARRAY

- Gli array possono essere utilizzati in modi diversi.
- Uno degli utilizzi più tipici è quello di organizzare i dati di un database sotto forma di array di oggetti.
- La posizione di un elemento nell'array è detta *indice*. Tutti gli array sono con base zero, ovvero [0] è il primo elemento dell'array, [1] è il secondo, e così via.
- Normalmente i contenuti di un array vengono esaminati utilizzando un ciclo for che consente di scorrere tutti gli elementi di un array.

PROPRIETÀ E METODI DEGLI ARRAY

- Gli array hanno un'unica proprietà:
  - ***length*** (numero di elementi di cui è composto l'array).
- Metodi:
  - concat([value:Object]) : Array: Concatena gli elementi specificati nei parametri con gli elementi in un array e crea un nuovo array.
  - join([delimiter:String]) : String: Converte in stringhe gli elementi di un array, inserisce il separatore specificato tra gli elementi, li concatena e restituisce la stringa risultante.
  - pop() : Object Rimuove l'ultimo elemento di un array e ne restituisce il valore.
  - push(value:Object) : Number Aggiunge uno o più elementi alla fine di un array e restituisce la nuova lunghezza dell'array.
  - reverse() : Void Inverte l'array in posizione.
  - shift() : ObjectRimuove il primo elemento di un array e lo restituisce.
  - slice([startIndex:Number], [endIndex:Number]) : Array Restituisce un nuovo array composto da un intervallo di elementi dell'array originale, senza modificare quest'ultimo.
  - sort([compareFunction:Object], [options:Number]) : ArrayOrdina gli elementi di un array.
  - sortOn(fieldName:Object, [options:Object]) : ArrayOrdina gli elementi di un array in base a uno o più campi dell'array.
  - splice(startIndex:Number, [deleteCount:Number], [value:Object]) : Array: Aggiunge e rimuove gli elementi di un array.
  - toString() : String: Restituisce un valore di tipo stringa che rappresenta gli elementi dell'oggetto Array specificato.
  - unshift(value:Object) : Number: Aggiunge uno o più elementi all'inizio di un array e restituisce la nuova lunghezza dell'array.

DATE

- La classe *Date* consente di recuperare i valori relativi alla data e all'ora del tempo universale (UTC) o del sistema operativo su cui è in esecuzione Flash Player.
- Per chiamare i metodi della classe Date, è prima necessario creare un oggetto Date mediante la funzione di costruzione della classe Date.

METODI DI DATE

- getDay() : Number: Restituisce il giorno della settimana (0 per domenica, 1 per lunedì, e così via) dell'oggetto Date specificato, in base all'ora locale.
- getFullYear() : Number: Restituisce l'anno completo (un numero di quattro cifre, ad esempio 2000) dell'oggetto Date specificato, in base all'ora locale.
- getHours() : Number: Restituisce l'ora (un numero intero compreso tra 0 e 23) dell'oggetto Date specificato, in base all'ora locale.
- getMilliseconds() : Number: Restituisce i millisecondi (un numero intero compreso tra 0 e 999) dell'oggetto Date specificato, in base all'ora locale.
- getMinutes() : Number: Restituisce i minuti (un numero intero compreso tra 0 e 59) dell'oggetto Date specificato, in base all'ora locale.
- getMonth() : Number: Restituisce il mese (0 per gennaio, 1 per febbraio, e così via) dell'oggetto Date specificato, in base all'ora locale.
- getSeconds() : Number: Restituisce i secondi (un numero intero compreso tra 0 e 59) dell'oggetto Date specificato, in base all'ora locale.
- getTime() : Number: Restituisce il numero di millisecondi trascorsi a partire dalla mezzanotte del 1 gennaio 1970 (tempo universale) per l'oggetto Date specificato.
- getYear() : Number: Restituisce l'anno dell'oggetto Date specificato, in base all'ora locale.

METODI DI DATE

- setDate(date:Number) : Number: Imposta il giorno del mese per l'oggetto Date specificato, in base all'ora locale, e restituisce il nuovo valore, espresso in millisecondi.
- setFullYear(year:Number, [month:Number], [date:Number]) : Number: Imposta l'anno dell'oggetto Date specificato, in base all'ora locale, e restituisce il nuovo valore, espresso in millisecondi.
- setHours(hour:Number) : Number: Imposta le ore dell'oggetto Date specificato, in base all'ora locale, e restituisce il nuovo valore, espresso in millisecondi.
- setMilliseconds(millisecond:Number) : Number: Imposta i millisecondi per l'oggetto Date specificato, in base all'ora locale, e restituisce il nuovo valore, espresso in millisecondi.
- setMinutes(minute:Number) : Number: Imposta i minuti per un oggetto Date specificato, in base all'ora locale, e restituisce il nuovo valore, espresso in millisecondi.
- setMonth(month:Number, [date:Number]) : Number: Imposta il mese per l'oggetto Date specificato, in base all'ora locale, e restituisce il nuovo valore, espresso in millisecondi.
- setSeconds(second:Number) : Number: Imposta i secondi per l'oggetto Date specificato, in base all'ora locale, e restituisce il nuovo valore, espresso in millisecondi.
- setTime(millisecond:Number) : Number: Imposta la data per l'oggetto Date specificato, espressa in millisecondi a partire dalla mezzanotte del 1 gennaio 1970, e restituisce il nuovo valore, espresso in millisecondi.
- toString() : String: Restituisce un valore di tipo stringa per l'oggetto Date specificato in formato leggibile

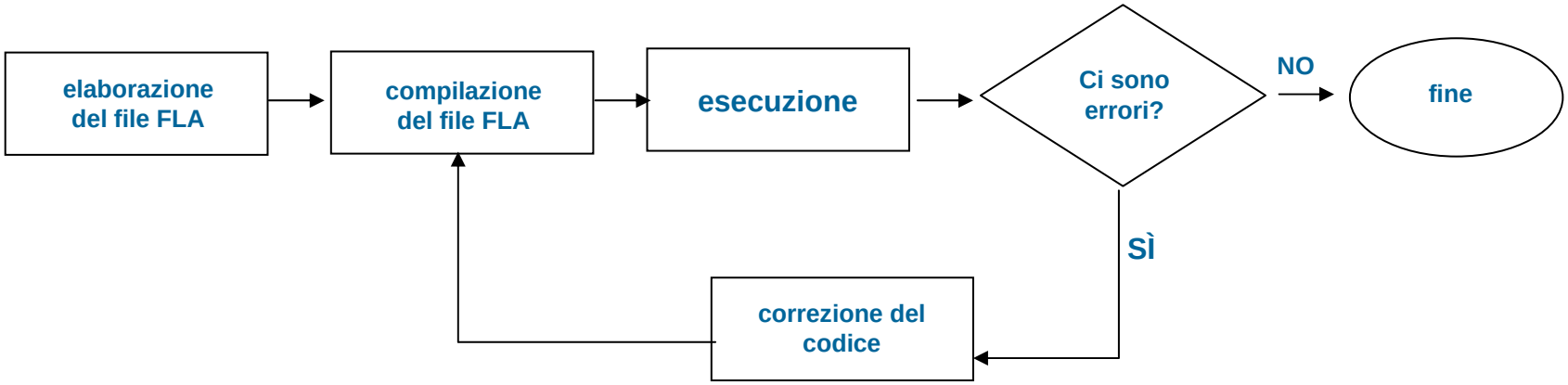
CREAZIONE DI CLASSI

LE CLASSI

- Nella programmazione orientata agli oggetti una classe definisce una *categoria di oggetto*.
- Le classi in pratica sono nuovi tipi di dati che il programmatore può creare per definire un nuovo tipo di oggetto. Una classe descrive le *proprietà* (*dati*) e i *metodi* (*comportamenti*) di un oggetto in modo molto simile a come un progetto di architettura descrive le caratteristiche di un edificio.
- Le proprietà (variabili definite all'interno di una classe) e i metodi (funzioni definite all'interno di una classe) di una classe sono detti *membri* della classe.
- In generale per utilizzare le proprietà e i metodi definiti da una classe, è necessario creare prima un'istanza di tale classe. La relazione tra un'istanza e la relativa classe è simile a quella che intercorre tra un edificio e il relativo progetto.

IL PROCESSO

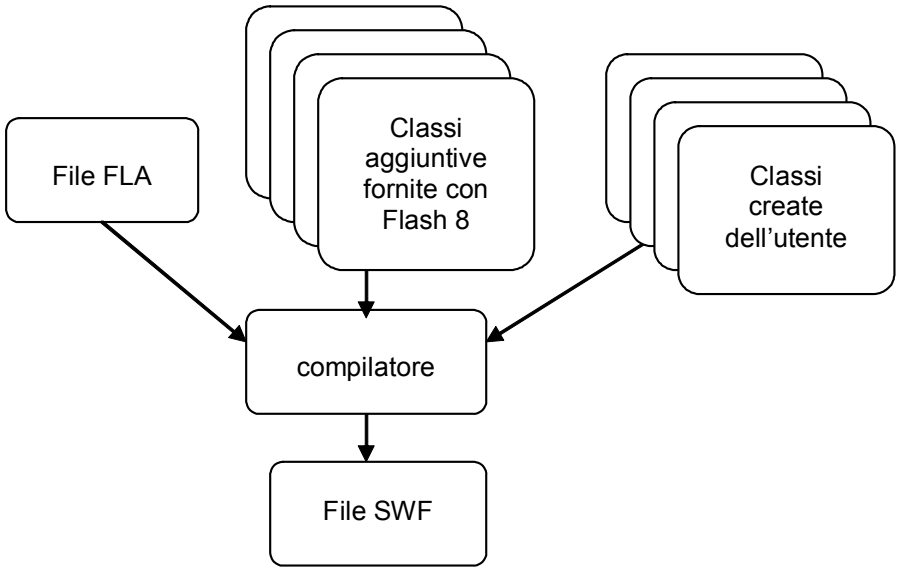
- Il processo di realizzazione di un progetto flash è strutturata in un flusso simile a questo:



- Con la programmazione con le classi introduciamo un nuovo tipo di file: il file **Action Script (.as)** e il processo si complica leggermente.

IL PROCESSO

- Per produrre il filmato finale in formato SWF il compilatore mette insieme tre tipi di risorse:



- Questo processo avviene quando scelgo l'opzione menu -> file -> pubblica

I FILE DI CLASSE

- Il file .as è un normale file di testo (tipo blocco note) che contiene codice Action Script.
- Una classe in *ActionScript* viene sempre definita in un file esterno (un normale file di testo con estensione *.as*) che ha lo stesso nome della classe e che viene chiamato *file di classe*.
- Quando un filmato flash viene compilato (utilizzando Controllo > Prova filmato o File > Pubblica) per generare il file *.swf*, il codice contenuto nei file di classe necessari viene compilato e aggiunto al file *.swf*..

I FILE DI CLASSE

- Quando il compilatore trova che nel filmato da compilare viene utilizzata una classe **DEVE** trovare il file che contiene il codice relativo a quella classe:
- Il file di classe deve avere esattamente lo stesso nome della classe (case sensitive).
- Il compilatore deve sapere in che cartelle cercare.

I FILE DI CLASSE

1. Esiste un elenco globale di cartelle che contengono classi che si può modificare andando in:
   Modifica>Preferenze>ActionScript
   e scegliendo il bottone “Impostazioni Action Script 2
2. Il compilatore oltre che nelle cartelle globali cerca anche nella cartella in cui è stato salvato il file .fla.

LE CLASSI AGGIUNTIVE

- Per default l'elenco globale delle cartelle contiene la cartella in cui si trovano le classi aggiuntive fornite con flash 8 (filtri grafici, funzioni geometriche, ecc)
- Queste classi sono organizzate in sottocartelle.
- In java e in Action Script le sottocartelle in cui sono organizzate le classi si chiamano **pakages** (pacchetti).

I PACCHETTI

- L'uso dei pacchetti serve esclusivamente ad organizzare il meglio il mio lavoro. Nella mia vita di programmatore svilupperò sempre nuovi progetti e presumibilmente creerò nuove classi, che per loro natura sono codice riciclabile, utilizzabile cioè in nuovi progetti.
- Se organizzerò queste classi in gruppi sarà più facile per me ritrovarle e riutilizzarle.

I PACCHETTI

- Per identificare una classe in un pacchetto (o in un sottopacchetto inserito in un pacchetto) devo usare la sintassi del punto.
- Per utilizzare la classe **BlurFilter** dovro scivere:

```
// senza importazione
var myBlur:flash.filters.BlurFilter =
  new flash.filters.BlurFilter(10,10,3);
```

IL COMANDO IMPORT

- Per semplificare la scrittura del codice posso usare il comando **import**.
- Utilizzando il comando eviterò di dovere scrivere sempre l'intero percorso della classe:

```
// con importazione
import flash.filters.BlurFilter
var myBlur:BlurFilter = new BlurFilter(10,10,3);
```

CREAZIONE DI CLASSI

- Per definire una classe devo creare un file action script esterno. Vediamo di definire le regole (alcune obbligatorie, altre solo consigliate) che devo seguire nello scrivere un file di classe saranno presenti questi elementi:
  - Il file di classe inizierà con commenti di documentazione, tra cui una descrizione generale del codice e informazioni sull'autore e sulla versione.
  - Seguiranno le istruzioni import (se necessarie).
  - Scriverò poi la dichiarazione di classe come segue:

```
class Studente {
...
}
```

CREAZIONE DI CLASSI

- Inserirò gli eventuali commenti relativi all'implementazione della classe.
- Dichiarerò, quindi, le variabili statiche.
- Dichiarerò le variabili di istanza rispettando questo ordine: prima le variabili pubbliche, poi quelle private.
- Aggiungerò l'istruzione relativa alla funzione di costruzione, come indicato nell'esempio seguente:

```
public function Studente(nome:String,
cognome:String, anno_nascita:Number)
{
  ...
}
```

CREAZIONE DI CLASSI

- Scriverò i metodi, raggruppandoli per funzionalità. Questo tipo di organizzazione dei metodi consente di migliorare la leggibilità e la chiarezza del codice.
- Scriverò i metodi getter/setter.

LA CLASSE STUDENTE

```
/*
   Classe Studente
   autore: Bruno Migliaretti
   versione: 0.8
   data modifica: 28/1/2008
   copyright: Accadema di Belle Arti di Urbino
   Questo codice definisce una classe Stundente personalizzata per
   Creare nuovi utenti e specificare Nome Cognome e data di nascita.
*/
class User {
   // Variabili di istanza private
   private var __nome:String;
   private var __cognome:String;
   private var __anno_nascita:Number;
   // Funzione di costruzione
   public function User(p_nome:String, p_cognome:String, anno:Number) {
            this.__nome = p_username;
            this.__cognome = p_cognome;
            this.__anno_nascita = anno;
   }
   public function get nome():String {
            return this.__nome;
   }
   public function get cognome():String {
            return this.__cognome;
   }
   public function set username(value:String):Void {
            trace("propietà a sola lettura");
   }
   public function set cognome(value:String):String {
            trace("propietà a sola lettura");
   }
}
```

LA CLASSE STUDENTE

- Un *commento di documentazione* standard che specifica il nome della classe, l'autore, la versione, la data di modifica, le informazioni di copyright e una breve descrizione dello scopo della classe. Scriverò i metodi getter/setter.
- funzione di costruzione della classe Studente
- le proprietà getter e setter per le variabili di istanza private

PROVARE UNA CLASSE

- Per creare e usare una classe è necessario:
  - Definizione di una classe in un file di classe *ActionScript* esterno.
  - Salvataggio del file di classe nella directory specificata per il percorso della classe (o nel percorso in cui Flash cerca le classi) oppure nella stessa directory del file FLA dell'applicazione.
  - Creazione di un'istanza della classe in un altro script, ossia un documento FLA o un file di script esterno, oppure tramite creazione di una sottoclasse basata sulla classe originale.

USARE UNA CLASSE

- Per creare un'istanza di una classe *ActionScript*, si utilizza l'operatore ***new*** per richiamare la funzione di costruzione della classe. Tale funzione ha sempre lo stesso nome della classe e restituisce un'istanza della classe che generalmente viene assegnata a una variabile.

```
var uno_studente:Studente = new
Studente(“Pio”,”rossi”,1988);
```

- Usando l'operatore punto (.) si accede al valore di una proprietà di un'istanza.

```
Uno_studente.nome;
```

PROPRIETÀ E METODI PUBBLICI E PRIVATI

- La parola chiave *public* specifica che una variabile o una funzione è disponibile per qualsiasi chiamante. Poiché le variabili e le funzioni sono pubbliche per impostazione predefinita, la parola chiave *public* viene utilizzata principalmente per ragioni di stile e leggibilità.
- La parola chiave *private* specifica che una variabile o una funzione è disponibile solo per la classe che la dichiara o la definisce o per le relative sottoclassi.

METODI E PROPRIETÀ STATICI

- La parola chiave *static* specifica che una variabile o una funzione viene creata solo una volta per ogni classe anziché in ogni oggetto basato sulla classe. È possibile accedere a un membro di classe statico senza creare un'istanza della classe. I metodi e le proprietà statici possono essere sia pubblici che privati.
- In ActionScript ci sono classi predefinite che hanno solo metodi e proprietà statiche.

METODI GETTER E SETTER

- I metodi getter e setter sono un modo alternativo per definire una proprietà. Quando creo una proprietà definendo una variabile pubblica all'interno della classe consento a chi usa la classe di interagire direttamente con la variabile. Quando leggerà la proprietà leggerà il contenuto della variabile, quando la modificherà il contenuto della variabile.
- Scrivere metodi setter e getter consente di controllare quanto viene scritto (o letto) ed eventualmente modificarlo.
- La sintassi dei metodi getter e setter è la seguente:
  - Un metodo getter non accetta parametri e restituisce sempre un valore.
  - Un metodo setter accetta sempre un parametro e non restituisce mai valori
- Per definire tali metodi, utilizzare gli attributi dei metodi ***get*** e ***set***. I metodi creati ottengono o impostano il valore di una proprietà e aggiungono la parola chiave *get* o *set* prima del nome del metodo

CREAZIONE DELLA FUNZIONE DI COSTRUZIONE

- Per funzioni di costruzione si intendono funzioni che consentono di inizializzare (*definire*) le proprietà e i metodi di una classe. Per definizione, le funzioni di costruzione sono funzioni incluse in una definizione di classe, con la stessa denominazione della classe
- La funzione di costruzione di una classe è una funzione speciale che viene chiamata quando si crea un'istanza di una classe utilizzando l'operatore new e presenta lo stesso nome della classe che la contiene.

EREDITARIETÀ

EREDITARIETÀ

- Uno dei vantaggi assicurati dalla programmazione orientata agli oggetti è la possibilità di creare *sottoclassi* di una classe.
- La sottoclasse eredita tutte le proprietà e i metodi di una ***superclasse***.
  - Posso creare una classe estendendo una classe predefinita.
  - Ma posso anche creare un set di classi che estenda una superclasse sempre creata da me.

EREDITARIETÀ

Vedura

EREDITARIETÀ

- La sottoclasse può aggiungere dei metodi alla superclasse e ridefinire i metodi ereditati creando così nuovi comportamenti per gli stessi metodi.
- L'uso di sottoclassi consente di riutilizzare il codice, in quanto estendendo una classe esistente si evita di riscrivere tutto il codice comune a entrambe le classi.

LE SOTTOCLASSI IN FLASH

- Nella programmazione orientata agli oggetti, una sottoclasse può ereditare le proprietà e i metodi di un'altra classe, detta ***superclasse***. È possibile estendere le classi personalizzate e molte delle classi di ActionScript. Non possono essere estese la classe ***TextField*** e le classi statiche, quali ***Math***, ***Key*** e ***Mouse***.
- Per creare questo tipo di relazione tra due classi, è necessario utilizzare la clausola ***extends*** dell'istruzione *class*:

```
class SubClass extends SuperClass {
     //corpo della classe
}
```

GESTIONE DEGLI EVENTI

COSA È UN EVENTO

- In programmazione un evento è un avvenimento che il mio programma è in grado di registrare e in seguito al quale posso programmare un azione.
- Esempi di eventi:
  - Le interazioni dell'utente con il mouse o la tastiera
  - Il passaggio al frame successivo in un clip filamato
  - Eventi legati al flusso dei dati (inizio caricamento, completamento del caricamento, ecc)
  - Eventi legati alla riproduzione di suono
  - Registrazione di errori
  - Ecc.

GESTIONE DEGLI EVENTI

- Gestire un evento significa programmare un'azione che viene eseguita ogni volta che l'evento si verifica.
- Gli eventi, in Action Script si gestiscono attraverso le classi.
- Quando un determinato evento si verifica (ad esempio l'utente clicca con il mouse su un pulsante o viene mostrato sullo schermo il frame successivo di un clip filmato) l'evento viene notificato all'istanza della classe bersaglio dell'evento.
- Io posso gestire questa notifica facendo in modo che l'istanza della classe in questione compia un azione ogni volta che l'evento viene notificato.

I GESTORI DI EVENTI

- Il target di un evento è sempre l'istanza di una classe
- In flash ho due tipi di gestori di eventi:
  - Gestione di eventi attraverso metodi specifici della classe
  - Gestione di eventi attraverso oggetti listener (modello listener-broadcaster).
- Alcune classi utilizzano il primo modello altre il secondo

METODI GESTORI DI EVENTI

- Un metodo gestore di un evento è un metodo della classe che viene richiamato quando un'istanza della classe riceve la notifica di un evento.
- La classe MovieClip, ad esempio, definisce un gestore di evento **onPress** che viene richiamato ogni volta che viene premuto il pulsante sinistro del mouse quando il puntatore si trova sulla MovieClip stessa.
- Per default il metodo gestore di eventi non contiene alcuna azione. Per programmare una azione in risposta ad un determinato evento devo assegnare al gestore di eventi una funzione anonima che contiene il codice che verrà eseguito ogni volta che l'evento si verifica.

METODI GESTORI DI EVENTI

- La gestione di un evento, secondo questo approccio, è costituita da tre elementi:
  - l'oggetto al quale si applica l'evento (o che ***riceve la notifica*** dell'evento),
  - il nome del metodo che la classe definisce per gestire l'evento
  - la funzione assegnata al gestore di eventi.
- Ecco la struttura di base di un gestore di eventi:

```
object.eventMethod = function () {
   /* Inserire qui il codice che
   risponde all'evento.*/
}
```

MODELLO LISTENER/BROADCASTER

- Il modello listener/broadcaster per gli eventi, prevede che ci sia un oggetto (detto ***listener***, *che sta in ascolto*) che riceve notifiche di eventi trasmesse da un altro oggetto detto ***broadcaster*** (trasmettitore).
- Le classi che usano questo modello non implementano, quindi, specifici metodi per rispondere agli eventi, ma un metodo generico (***addListener***) che consente di comunicare all'istanza della classe a quali oggetti mandare le notifiche degli eventi.
- L'oggetto listener può ascoltare più eventi, l'oggetto broadcaster puo spedire notifica dello stesso evento a più oggetti.

MODELLO LISTENER/BROADCASTER

- Tipicamente la gestione di un evento, secondo questo approccio, comporta tre passaggi:
  - Creazione dell'oggetto che ascolta gli eventi (o oggetto listener)
  - Assegnazione all'oggetto delle funzioni che gestiscono gli eventi sotto forma di proprietà
  - Registrazione dell'oggetto listener da parte dell'oggetto broacaster utilizzando il metodo ***addListener***.

MODELLO LISTENER/BROADCASTER [1]

- Il primo passo consiste nel creare un oggetto generico (cioè di tipo **Object**) che diventrà il mio listener:

```
myListener = new Object();
```

- Con questo codice ho creato un oggetto di tipo **Object** che ha nome myListener

MODELLO LISTENER/BROADCASTER [2]

- Una proprietà di un oggetto può contenere dati di qualsisasi tipo, una serie di comandi ActionScript (cioè una funzione). Per far diventare il mio oggetto un ***listener*** devo creare una proprietà che abbia il nome dell'evento da gestire e assegnare a quella proprietà il codice da eseguire quando l'evento si verifica:

```
myListener.eventName = function (p1, p2,…) {
   /* Inserire qui il codice che risponde
   all'evento.*/
}
```

- Il codice è praticamente identico a quello usato per i metodi gestori di eventi. La differenza è che invece di applicarlo all'istanza della classe che riceve l'evento lo applichiamo ad un oggetto creato ad hoc.

MODELLO LISTENER/BROADCASTER [3]

- Infine devo comunicare all'oggetto ***broadcaster*** (l'oggetto principale, quello per il quale devo gestire gli eventi) che gli eventi vanno inviati al mio listener

```
broadcasterObject.addListener(myListener);
```

- Con il metodo ***addListener*** posso registrare più oggetti listener allo stesso broadcaster.
- Con il metodo ***removeListener*** posso, se necessario, togliere un listener dalla lista a cui il broadcaster notifica gli eventi.

LA PAROLA RISERVATA THIS

- Un'altra importante differenza tra il codice scritto per gestire un evento secondo il modello 'metodo gestore di evento' e il modello oggetto brodcaster→oggetto listener è il significato che nei due contesti ha la parola chiave ***this***
- ***this*** indica sempre l'oggetto a cui appartiene la funzione in cui viene usato (o l'istanza della classe a cui è applicato il metodo in cui viene usato)

THIS IN UN METODO GESTORE DI EVENTI

- In un metodo gestore di eventi this rappresenta l'oggetto bersaglio dell'evento:

```
myClip.onEnterFrame = function () {
   this._rotation = this._rotation + 1;
}
```

- Il codice precedente usa l'evento onEnterFrame per far ruotare il Clip Filmato myClip
- La proprietà onEnterFrame è una proprietà di myClip quindi in questo caso this indica myClip

THIS NEL MODELLO LISTENER => BROADCASTER

- In un listener this rappresenta l'oggetto listener stesso:

```
myLoader = new MovieClipLoader();
myListener = new Object();
myListener.onLoadInit = function
(myMc:MovieClip) {
   myMc._alpha = 100;
}
myLoader.addListener(myListener);
theClip._alpha = 0;
myLoader.loadClip("pippo.swf",theClip);
```

THIS NEL MODELLO LISTENER => BROADCASTER

- Il codice della slide precedente carica una clip utilizzando l'oggetto ***MovieClipLoader.*** Quando si verifica l'evento **loadInit** la clip viene visualizzata impostanto la sua proprietà **_alpha** a 100 (assenza di trasparenza).
- In questo caso non si potrebbe ulitizzare ***this*** per modificare la proprietà della clip. ***this*** infatti indica sempre l'oggetto a cui appartiene la funzione in cui viene usato, il questo caso l'oggetto listener.
- Per semplificarci le cose la notifica dell'evento ci procura come parametro il riferimento alla MovieClip caricata. Per intervenire sulla clip si utilizzerà, quindi, questo parametro.

CLASSI CHE DEFINISCONO METODI GESTORI DI EVENTI:

- Button
- ContextMenu, ContextMenuItem
- LoadVars
- LocalConnection
- MovieClip
- SharedObject
- Sound
- TextField
- XML, XMLSocket.

CLASSI CHE USANO IL MODELLLO LISTENER:

- Key
- Mouse
- MovieClipLoader
- Selection
- Stage

PER APPROFONDIRE

fl8_learning_as2.pdf

Pagina 311 e seguenti

AGGIUNTA DI METODI E PROPRIETÀ

LA CLASSE JUKEBOX

- Ora definiremo una classe personalizzata ***JukeBox*** che estende la classe ***Sound***:
- Vengono aggiunti la proprietà
  - ***song_array*** di tipo *Array*
  - ***titles_array*** sempre di tipo *Array*,
- Vengono aggiunti i metodi
  - ***playSong***() che riproduce un brano musicale richiamando il metodo *loadSound*() ereditato dalla classe Sound,
  - ***addSong***() che aggiunge un brano al JukeBox
  - ***getTitleList***() che restituisce la lista dei titoli inseriti nel JukeBox.

CREIAMO IL FILE DI CLASSE

```
/**
      Classe JukeBox
      autore: Bruno Migliaretti
      versione: 1.0
      data modifica: 01/12/2006
      copyright: Accademia di Belle Arti di
      Urbino
      Questo codice definisce una classe JukeBox
      che estende la classe prdefinita Sound e
      consente di gestire una lista di brani
      musicali.
*/

class JukeBox extends Sound {
}
```

LE PROPRIETÀ AGGIUNTE

- Definire e inizializzare ora nel codice le proprietà ***song_array*** e ***titles_array*** aggiungendo al corpo della classe:

```
class JukeBox extends Sound {
	private var song_array:Array = new Array();
		//lista dei brani del JukeBox
	private var titles_array:Array = new Array();
		//lista dei titoli del JukeBox
}
```

IL METODO ADDSONG

- Aggiungere ora il codice per il metodo ***addSong***():

```
class JukeBox extends Sound {
.......
      public function addSong (titolo:String,
                  nomeFile:String):Number {
            /* aggiungo in coda ai due array il
               titolo del brano e il file musicale
               corrispondente*/
            this.titles_array.push(titolo);
            this.song_array.push(nomeFile);

            //restituisco l'indice del brano inserito
            return (song_array.length - 1);
      }
}
```

IL METODO PLAYSONG

- Il metodo *addSong*() accetta due parametri entrambi di tipo *String*: *titolo* e *nomeFile*. I due elementi vengono aggiunti rispettivamente agli array *song_array* e *titles_array*. Infine il metodo restituisce l'indice dell'elemento appena inserito.
- Aggiungere ora il codice per il metodo *playSong*()

```
class JukeBox extends Sound {
.......
      public function playSong (songId:Number) {
            /* il file memorizzato nella proprietà
               song_array con indice songId viene
               caricato ed eseguito */
            this.loadSound(song_array[songId], true);
      }
}
```

IL METODO GETTITLELIST

- Aggiungere ora il codice per il metodo ***getTitleList***()
- Il metodo ***getTitleList***() consente l'accesso in lettura alla proprietà *title_array* che devo poter essere modificata solo dal metodo ***addSong***() e quindi e definita come privata.

```
class JukeBox extends Sound {
.......
      public function getTitleList():Array {
            /* in questo modo titles_array è
               accessibile solo in lettura e può
               essere modificato solo usando il metodo
               addSong */
            return titles_array;
      }
}
```

USARE CLASSE JUKEBOX

- Ora creeremo un nuovo file flash per provare la classe JukeBox:
- Creiamo un testo statico e scriviamo Lista dei brani.
- Collochiamo poi sullo Stage un'istanza della componente List trascinandone l'icona dalla finestra componenti allo stage. Usiamo il pannello proprietà per dare alla componente il nome song_list.
- Creiamo un nuovo livello Azioni. Selezionimao il primo fotogramma del nuovo livello per scrivere il cosice che farà funzionare la nostra classe

USARE CLASSE JUKEBOX

- Creiamo un'istanza della classe JukeBox e poi aggiungiamo i brani:

```
//creo un'istanza di JukeBox
var jb:JukeBox = new JukeBox();

// aggiungo le informazioni sui brani musicali
jb.addSong("Blue", "0169_Blue.mp3");
jb.addSong("Cafe caldo", "0217_Cafe Caldo.mp3");
jb.addSong("Cello sweet", "0246_Cello Sweet.mp3");
jb.addSong("Conto alla rovescia",
"0328_Countdown.mp3");
jb.addSong("Vai facile", "0459_Easy Does It.mp3");
……
```

USARE CLASSE JUKEBOX

- E riempiamo la list Song List:

```
//riempio la lista song_list
var titoli:Array = jb.getTitleList();

for (var i = 0; i < titoli.length; i++) {
	song_list.addItem({label:titoli[i],data:i});
}

//gestisco l'evento change delle lista
var listHandler = new Object();

listHandler.change = function(evt:Object) {
	var index = evt.target.selectedItem.data;
	jb.playSong(index);
}
this.song_list.addEventListener("change",listHandler);
```

SOSTITUZIONE DI METODI E PROPRIETÀ

LA CLASSE DATAIT

- Uno dei vantaggi offerti dall'uso delle classi e dalla loro estensione è la possibilità non solo di garantire nuove funzionalità a una classe esistente aggiungendo metodi e proprietà, ma anche di modificare le funzionalità già presenti.
- Come esempio modificheremo il comportamento del metodo ***toString***() nella classe predefinita ***Date*** creando una classe che chiameremo ***DataIt***.
- Il medito ***toString***() è comune a tutte le classi predefinite e viene invocato automaticamente quando un'istanza di una qualsiasi classe viene usata come una stringa. Se scrivo:

```
var now:Date = new Date();
trace(now);
```

- Otterrò il seguente output:

  Tue Nov 28 22:07:32 GMT+0100 2006

LA CLASSE DATAIT

- Quando passo un oggetto come parametro alla funzione ***trace***() (che manda un messaggio sulla finestra di output) di fatto lo converto in stringa e quindi applico implicitamente il metodo ***toString***() all'oggetto.
- Modificando quindi il comportamento standard del metodo ***toString***() per un oggetto personalizzarò il modo con cui quell'oggetto verrà convertito in stringa.

IL CODICE

```
/**
		Classe DataIt
		autore: Bruno Migliaretti
		versione: 1.0
		data modifica: 01/12/2006
		copyright: Accademia di Belle Arti di Urbino
		Questo codice definisce una classe DataIt che estende la classe predefinita Date
		traducendone l'output in italiano.
*/
class DataIt extends Date {
		//creo una proprietà mesi che contiene i nomi dei mesi
		private var mesi:Array = new Array("gennaio", "febbraio", "marzo", "aprile", "maggio", "giugno", "luglio", "agosto",
					"settembre", "ottobre", "novembre", "dicembre");
		//e una proprietà giorni che contiene i nomi dei giorni della setimana
		private var giorni:Array = new Array("domenica", "lunedì", "martedì", "mercoledì","giovedì", "venerdì", "sabato");

		//scrivo una semplice metodo che aggiunge uno "0"
		//davanti ad un numero se è composto da una sola cifra
		private function zeroPrima(n:Number):String {
				var s:String = n.toString();
				if (s.length == 1) {
					s = "0" + s;
				}
				return (s);
		}

		//ridefinisco il metodo toString()
		public function toString():String {
				var giorno_sett:String = giorni[this.getDay()];
				var giorno:Number = this.getDate();
				var mese:String = mesi[this.getMonth()];
				var anno:Number = this.getFullYear();
				var ora:String = zeroPrima(this.getHours());
				var minuti:String = zeroPrima(this.getMinutes());
				var secondi:String = zeroPrima(this.getSeconds());
				return( giorno_sett + " " + giorno + " " + mese + " " + anno +
					" " + ora +":" + minuti + ":" + secondi);
		}
}
```