

Paradigma a oggetti

Programmazione orientata agli oggetti

Paradigma di programmazione:

- programmazione imperativa, ma non solo (Scala, OCaml)
- concetto di oggetto (campi e metodi)
- usato nei linguaggi di programmazione più diffusi (C++, Java, Python, Javascript)
- declinato in vari modi:
 - puro: tutti i dati sono oggetti (Smalltalk, Ruby)
 - oggetti come meccanismo ausiliario (Ada)

Caratteristiche tipiche

Si estende il meccanismo dei tipi di dati astratti che permette:

- information hiding e incapsulamento
 - nascondo la rappresentazione interna di un dato
 - accedo al dato attraverso delle funzioni base
 - definite all'interno del tipo di dato (astratto)

con meccanismi che permettono:

- estensione dei dati con riuso del codice (ereditarietà)
- compatibilità tra tipi (polimorfismo di sottotipo)
- selezione dinamica dei metodi

Esempio

Definisco il tipo di dato contatore

```
abstype Counter {
  type Counter = int;
  signature
    void reset(Counter c);
    int get(Counter c);
    void inc(Counter c);
  operations
    void reset(Counter c){
      c = 0;
    }
    int get(Counter c){
      return c;
    }
  ...
}
```

Per estendere `Counter` devo riscrivere il codice, nessuna correlazione di tipo tra `NewCounter` e `Counter`:

```
abstype NewCounter {
  type NewCounter = struct{
      int count;
      int resets = 0;
  }

  signature
  void reset(NewCounter c);
  int get(NewCounter c);
  void inc(NewCounter c);
  operations
  void reset(NewCounter c){
      c.count = 0;
      c.resets += 1;
  }
  int get(NewCounter c){
      return c.count;
  }
}
```

- Un oggetto è una capsula (record, struct) che contiene:
 - campi: dati,
 - metodi: procedure associate ai dati, formalmente memorizzate insieme ad essi
- Un programma orientato agli oggetti:
 - invocare un metodo su un oggetti
 - un oggetto risponde, eseguendo la procedura associata,

- Estendere un tipo di dato astratto (`NewCounter`), potendo ereditare (parte del codice) (di `Counter`)
- Poter utilizzare il tipo di dato esteso (`NewCounter`) in contesti che accettano il tipo di dato originale (`Counter`); compatibilità tra tipi
- Selezione dinamica dei metodi: procedure che usino in maniera uniforme `Counter` e `NewCounter` devo decidere, a tempo di esecuzione, quale implementazione del metodo `reset` utilizzare.

Definiscono la struttura degli oggetti

Una classe definisce un insieme di oggetti specificando:

- una serie di campi con relativo tipo
- una serie di metodi con relativo tipo e codice
- visibilità dei campi e metodi
- costruttori degli oggetti

Oggetti creati dinamicamente per istanziazione di una classe, mediante costruttori

```
class Counter {
  private int count;
  public void reset(){
      count = 0;
  }
  public int get(){
```

Diversi aspetti nella definizione di una classe:

- **information hiding** e incapsulamento
definisco cos'è visibile all'esterno e cosa no
`private`, `public`, `protected`
- **astrazione sui dati e sul controllo**
assegno un nome alla classe, ai campi, ai metodi
- **estensibilità e riuso del codice**
nel caso definisca delle sottoclassi,

I linguaggi OO si distinguono in:

- **Class based**: più comuni
- **Prototype based** (object-based):
 - oggetti come record con metodi.
 - non sono necessari pattern predefiniti
 - presente solo nei linguaggi con sistema di tipo dinamico
- nessuna parentela con Java,
- linguaggi di default per introdurre codice in pagine html,

- Oggetti come record (oggetti, e record, tipi di dati esprimibili)

```
var person = {firstName:"John", lastName:"Doe", age:50,
  name = function () {
    return this.firstName + " " + this.lastName;
  }
};
```

- oggetti estensibili: posso aggiungere nuovi dati o metodi

```
person.eyeColor = "blue";
```

- linguaggio con tipi dinamici, controllo dei tipi a run-time
- nessuna distinzione tra campi e metodi,
 - formalmente ogni oggetto contiene i propri campi e metodi
 - concettualmente indistinti
entrambi possono essere ridefiniti
 - implementazione: metodi descritti da puntatori a codice condiviso

- Possibilità di definire dei costruttori

```
function Person(first, last, age) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.name = function () {
    return this.firstName + " " + this.lastName;};
}
```

```
var myFather = new Person("John", "Doe", 50);
```

- Possibilità di estendere i costruttori

```
Person.prototype.altName = function() {
  return this.lastName + " " + this.firstName;
```

JavaScript: costruttori predefiniti

- Costruttori predefiniti, creano oggetti standard con un ricco insieme di metodi

```
var x1 = new Object(); // A new Object object
var x2 = new String(); // A new String object
var x3 = new Number(); // A new Number object
var x4 = new Boolean(); // A new Boolean object
var x5 = new Array(); // A new Array object
var x6 = new RegExp(); // A new RegExp object
var x7 = new Function(); // A new Function object
var x8 = new Date(); // A new Date object
```

Identificatori `this` e `self`

All'interno dei metodi di un oggetto spesso si fa riferimento ad altri metodi o campi dell'oggetto stesso

questo può essere fatto in maniera:

- esplicita, con le parole chiave `this` o `self`
`this.field`, `self.field` (a seconda del linguaggio)
per affermare che faccio riferimento ad un campo interno
- implicita: nel metodo appare solo `field`
implicitamente si intende il campo `field` dell'oggetto corrente

In Java:

- riferimento implicito
- possibile usare la keyword `this`
utile in caso di mascheramento del campo o del metodo

JavaScript: meccanismo delle closure

```
function counter() {
  var count = 0;
  return function() {
    return ++count;
  };
}
var closure = counter();
closure(); // restituisce 1
closure(); // restituisce 2
```

Funzioni oggetti del primo livello,
funzioni possono restituire funzioni

- `counter` costruisce una funzione e la restituisce
- `closure` fa riferimento all'ambiente di `counter` in cui è stata definita
- `count` definita in questo ambiente viene preservata (e incrementata)

closure

Modello riferimento-valore

Normalmente oggetti usano il modello a [riferimento](#)

- variabile contiene un puntatore all'oggetto
- oggetti memorizzati nell'heap
- creazione esplicita degli oggetti mediante funzioni di costruzione, allocazione dello spazio
- meccanismi di garbage collection per riuso della memoria

Modello a [valore](#)

- variabile contiene i campi dell'oggetto
- può essere memorizzata nello stack
- creazione implicita degli oggetti (chiamata di procedure)
- usata in C++,
- accesso diretto ai dati e maggiore efficienza

Incapsulamento

Le classi garantiscono incapsulamento

- opportuni modificatori determinano se campi e metodi sono
 - **pubblici**, `public`
 - **privati**, `private`
 - **protetti**, `protected`
- la parte pubblica definisce l' "interfaccia" della classe
- la parte privata è usata nell'implementazione
- `protected` definisce una parte privata, visibile nelle sottoclassi

Linguaggi diversi offrono funzionalità diverse

- `protected` può aver diversi significati
 - visibile nelle sottoclassi
 - visibile nel package

Costruttori

Nel definire una classe devo definire una serie di costruttori:

- inizializzano i campi del nuovo oggetto creato
- possibili più costruttori, selezione
 - per numero e tipo degli argomenti, se il nome del costruttore coincide con quello della classe (Java)
 - per nome
- nel caso sottoclassi:
 - eseguo prima il costruttore della super-classe eventualmente con chiamata esplicita
 - e poi quello della sottoclasse
- chiamata con sintassi ad hoc, `new`

Static field, static method

Oltre alle etichette di visibilità: `public...`

i campi e metodi possono essere etichettati come `static`

- un **campo static**
 - variabile, in singola copia, condivisa tra tutti gli oggetti di una classe
 - le modifiche fatte da un oggetto sono visibili agli altri
- **metodo static**
 - può accedere solo alle variabili `static`
 - non accede alla variabile dei singoli oggetti (`instance`)
 - non usa l'identificatore `this`, nemmeno in maniera implicita

In altri ambiti:

- in una procedura una **variabile locale static**
 - variabile memorizzata nella parte statica della memoria
 - preserva il suo valore tra una chiamata e l'altra
 - nel caso di procedure ricorsive presente in singola copia

Distruttori

Possibile definire distruttori

- metodi da eseguire prima della deallocazione di un oggetto

Recupero spazio di memoria heap

- esplicita
- garbage collector
meno efficiente ma più sicura

Sottoclassi

Posso estendere classi con nuovi metodi o campi,

```
class NamedCounter extends Counter{
  private String name;
  public void set_name(String n){
    name = n;
  }
  public String get_name(){
    return name;
  }
}
```

NamedCounter è una sottoclasse (o classe derivata) di Counter:

- ogni istanza di NamedCounter risponde a tutti i metodi di Counter
- il tipo NamedCounter è **compatibile** con Counter

Ridefinizione di metodo (overriding)

```
class NewCounter extends Counter{
  private int num_reset = 0
  public void reset (){
    count = 0;
    num_reset++;
  }
  public int get_num_reset() {
    return num_reset;
  }
}
```

- NewCounter contemporaneamente:
 - estende l'interfaccia di Counter con nuovi campi
 - ridefinisce il metodo reset
 - il metodo reset ha due diverse definizioni in Counter e NewCounter

Ridefinizione dei campi porta al **shadowing**

Ruby, getter and setter

Ruby vuol essere un linguaggio ad oggetti puro

- tutti i dati sono oggetti
- posso accedere ai campi interni solo attraverso i metodi

```
class Temperatura
  def celsius
    return @celsius
  end
  def celsius=(temp)
    @celsius = temp;
  end
  def fahrenheit
    return @celsius * 9/5 + 32;
  end
  def fahrenheit=(temp)
    @celsius = (temp - 32) * 5/9
  end
end
```

Ruby, zucchero sintattico

Le uniche operazioni ammesse sono chiamate di metodi

si usa la sintassi standard attraverso **zucchero sintattico**

```
Temperatura.celsius = 0;
# sta per Temperatura.celsius=(0)
y = Temperatura.fahrenheit
y + 5 # sta per per y.+(5)
```

Una sottoclasse può modificare la visibilità della super-classe

- C++ nasconde metodi pubblici nella super-classe

```
class derived : protected base { ... }
class derived2 : private base { ... }
```

- Eiffel modifica in maniera arbitraria
- Java, C# non modifica la visibilità della superclasse

Attraverso la definizione di sottoclassi introduco un meccanismo di ereditarietà per il riutilizzo e la condivisione del codice

- in un esempio precedente, qui riportato
 - NamedCounter eredita da Counter i metodi (e i campi) non ridefiniti
- l'ereditarietà permette condivisione del codice in maniere modificabile, estendibile:
 - ogni modifica all'implementazione di un metodo in una classe automaticamente disponibile a tutte le sottoclassi

```
class NamedCounter extends Counter{
    private String name;
    public void set_name(String n){
        name = n;
    }
    public String get_name(){
        return name;
    }
}
```

- Singola (Java)
 - ogni classe `extends` una sola super-classe
 - eredita al più da una sola super-classe immediata
- Multipla (C++)
 - una classe può ereditare da più di una super-classe immediata
- Ereditarietà multipla
 - più flessibile il riuso del codice
 - fonte di problemi
 - name clash: se lo stesso metodo definito in due super-classi, quale metodo ereditare?
problema di coerenza tra definizioni multiple

```
class Top{
    int w;
    int f(){
        return w;
    }
}
class A extending Top{
    int x;
    int g(){
        return w+x;
    }
}
class B extending Top{
    int y;
    int f(){
        return w+y;
    }
    int k(){
        return y;
    }
}
class Bottom extending A,B{
    int z;
    int h(){
        return z;
    }
}
```

Nell'esempio precedente quale metodo f eredita `Bottom`

- quello ereditato nella classe A
- quello ridefinito della classe B

Possibile soluzioni:

- impedisco ogni conflitto tra metodi
- impongo che ogni conflitto venga risolto nel codice, specificando da quale classe ereditare il metodo (C++)
- definisco un criterio di scelta dei metodi presenti in più classi (es. quelle che appare per ultimo nel codice)

Non esiste una soluzione ottimale,

- per ciascuna delle precedenti si possono fare esempi in cui risulta innaturale.

Tipico dei linguaggi OO

Un tipo T è sottotipo di S ($T <: S$) quando:

- T compatibile con S ,
- nei contesti di tipo S posso inserire un oggetto T

Definendo una relazione di sottotipo posso definire funzione **polimorfe**

se

- $T <: S$
- $f : S \rightarrow R$

allora automaticamente:

- $f : T \rightarrow R$
- posso assegnare a f tipi diversi
- a f possa applicare valori di diverso tipo

Soluzione sofisticata, alternative all'ereditarietà multipla,

- permette un riutilizzo del codice, evitando le ambiguità dell'ereditarietà multipla
- usa classi astratte
- permette di incorporare metodi da una classe senza diventarne sottoclasse,
- quindi metodi generici usabili su più classi
- realizzato
 - in varie forme nei diversi linguaggi
 - esempio: classi astratte con codice di default per i metodi

Spesso

- se la classe T è sottoclasse di S
- allora la classe T è sottotipo di S

Infatti

- se T possiede tutti i metodi di S
- posso usare oggetti T in tutti i contesti S

- C++, Eiffel
 - perché nelle sottoclassi posso rendere privati metodi della pubblici della superclasse
 - sottotipo solo se questo non accade
- in C++

```
class B : public A {... \\ B sottotipo di A
class C : A {... \\ C non sottotipo di A ma eredita
```

- classi con **binary methods**
 - una classe A contiene un metodo che ha un parametro di tipo A esempio test di uguaglianza, un oggetto controlla se un altro oggetto è identico a lui
 - A contiene un test di uguaglianza ==
 - B <: A, a : A e b : B
 - b.== b1 controlla che b e b1 abbiano tutti i campi uguali il contesto [_ == a1] accetta oggetti di tipo A ma non di tipo B
 - i binary methods sono incompatibili con la relazione di sottotipo

- Sottoclasse relazione d'ordine:
A <: B, B <: C implica A <: C
- In alcuni linguaggi esiste:
Object classe più generale di tutti, tutte le classi sono sottoclassi di Object contiene metodi comuni a tutte le classi creazione, **uguaglianza** (stesso riferimento, non uguaglianza dei campi)
- Senza ereditarietà multipla, la relazione di sottoclasse descritta da un albero

Metodi astratti, classi astratte, interface

Utilizzati anche per arricchire la relazione di sottotipo

- Metodi astratti: contengono solo la dichiarazione, senza il corpo (codice) etichettata come `abstract`
- Classe astratta se contiene almeno un metodo astratto
 - non è possibile creare oggetti di classe astratte
 - servono come modello per classi concrete (tutti i metodi istanziati)
- Interface: classi
 - con solo metodi
 - nessun campo

Vantaggi delle `interface`:

- definire la struttura di una classe senza specificare il codice
 - si definiscono tipo dei metodi, ma non il codice dei metodi
- semplici dichiarazioni di tipo
- non dovendo ereditare il codice, nessuna controindicazione ad implementare (essere sottotipo) di più di un interface

Esempio

```
abstract class A{
    public int f();
}

abstract class B{
    public int g();
}

class C extending A,B{
    private x = 0;
    public int f(){
        return x;
    }
    public int g(){
        return x+1;
    }
}
```

Se una classe concreta C implementa un interface (classe astratta) A

- C sottotipo di A
- oggetti di C rispettano le specifiche di A

Un tipo può avere più di un sovratipo immediato

- C sottotipo sia di A che di B
- per le interface non esistono i problemi di name-clash dell'ereditarietà multipla
 - nessun codice,
 - tipi forzatamente identici
- la gerarchia dei sottotipi non è un albero,

- Sottotipo

- meccanismo che permette di usare un oggetto in un altro contesto
- è una relazione tra le **interfacce** di due classi
interfaccia insieme dei campi e metodi pubblici di una classe

- Ereditarietà

- meccanismo per riusare il codice dei metodi,
- è una relazione tra le **implementazioni** di due classi.

Due concetti formalmente indipendenti

- ma spesso collegati nei linguaggi OO, con le sottoclassi ho:
 - l'ereditarietà dei metodi
 - una relazione di sottotipo.
- sia C++ che Java hanno costrutti che introducono contemporaneamente entrambe le relazioni tra due classi
 - in Java
extends introduce sia ereditarietà e sottotipo
implements solo sottotipo

```
interface A{
    int f();
}
interface B{
    int g();
}
class C{
    int x;
    int h(){
        return x+2;
    }
}
class D extends C implements A, B{
    int f(){ return x; };
    int g(){ return h(); };
}
```

Selezione dinamica dei metodi

- Un metodo `m` viene invocato su un parametro `count` di tipo oggetto :
 - nelle diverse invocazioni `count` può essere istanziato con sottotipi diversi
 - con diverse versioni del metodo `m`
 - come avviene la scelta della versione di `m` da invocare?
- Esempio

```
Counter V[100];
```

```
...
```

```
for (Counter count : V){count.reset()}
```

- `V` può contenere elementi di tipo `Counter` che `NewCounter`
- Selezione dinamica (Java)
 - a tempo d'esecuzione, in funzione del tipo dell'oggetto che riceve il messaggio

selezione determinata dal tipo dell'oggetto
non dal tipo del parametro `count` all'oggetto

Polimorfismo di sottotipo

- La relazione di sottotipo utile per implementare polimorfismo di sottotipo

```
interface Printable{void print() {...};};
```

```
void f (Printable x) {...};
```

```
// f : (T <: Printable) -> void
```

definisco una funzione che può essere chiamata su una qualsiasi istanziazione di `A`

Selezione statica

- Selezione statica (C++),
 - dipende dal tipo del parametro
 - più efficiente
 - ma si introducono meccanismi per permettere la selezione dinamica metodi `virtual`

Polimorfismo parametrico

- uso parametri di tipo, `<T>`, per definire classi e funzioni parametriche

```
class Tree<T>{ ... };
```

```
public static <T> int height (Tree<T> t) {...};
```

```
Tree<int> tint = new Tree<int>();
```

```
...
```

```
n = height<int> tint;
```

```
m = height tint;
```

```
// height: forall T . Tree(T) -> int
```

Definisco funzioni parametriche con vincoli sul parametro:

```
interface Printable{ void print(); }  
public static <T extends Printable> void printAll (Tree<T> tTree) {  
  
    // printAll: forall T <: Printable . Tree(T) -> void
```

Alcuni linguaggi (Python, JavaScript)
non si basano sulla relazione di sottotipo ma sul

- duck test — “If it walks like a duck and it quacks like a duck, then it must be a duck”
 - l’uso di oggetto in un contesto è lecito se l’oggetto possiede tutti i metodi necessari
 - indipendentemente dal suo tipo
 - usato nei linguaggi con un sistema di tipo dinamico
 - controllo a tempo di esecuzione che se un oggetto riceve un messaggio m , m appartenga ai suoi metodi
 - tipi di dati dinamici
 - maggiore espressività rispetto a tipi statici
 - minore efficienza, maggior numero di controlli a runtime
- altrimenti minor controllo degli errori, possibili errori di tipo nascosti