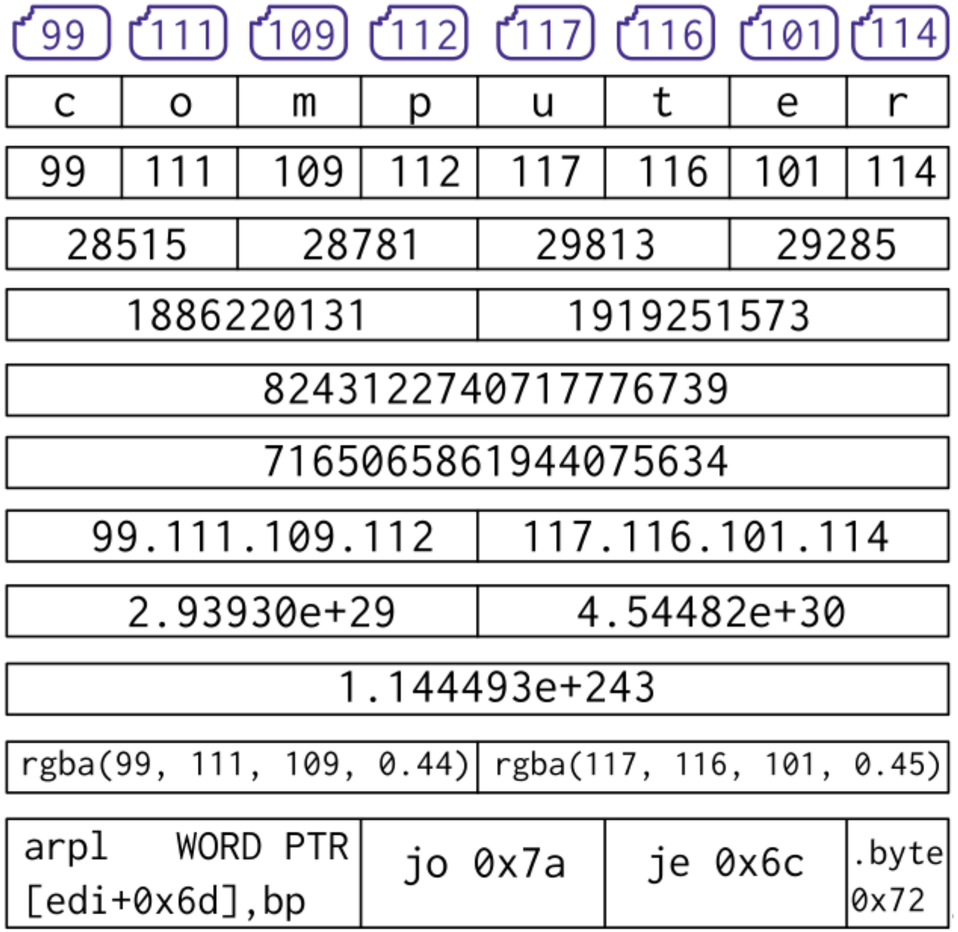
Tipi di Dato Astratto e Object Orientation

Tipi di Dato Astratto (Abstract Data Types)

Come visto, le macchine fisiche gestiscono stringhe di bit.

Al contrario, più i programmi diventano complessi, più è complicato per gli sviluppatori ragionare a quel livello basso di astrazione (dove un array, una tabella o un grafico hanno più o meno la stessa "superficie").

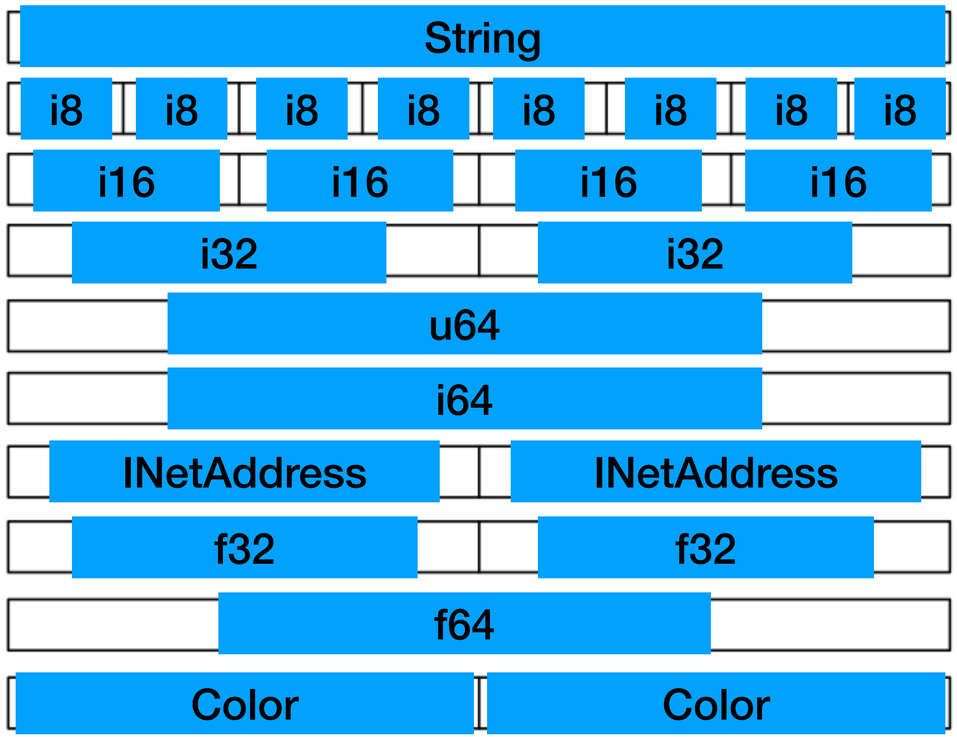


Tipi di Dato Astratto (Abstract Data Types)

Per questo, i linguaggi di programmazione hanno adottato i tipi anche come una sorta di "**capsula**" per organizzare i valori da stringhe indistinguibili di bit in entità distinte con un proprio insieme di operazioni.

Nei linguaggi **type-safe, la capsula** rappresentata da un tipo **è** completamente **opaca** e l'utente non può interagire con i suoi valori se non tramite la mediazione della capsula.

In questi casi, chiamiamo il tipo un **tipo di dato astratto** (**ADT**), cioè **un tipo che definisce i possibili valori che lo compongono e le possibili operazioni che possono agire su questi**.



saverio.giallorenzo@gmail.com

Tipi di Dato Astratto

Gli ADT non si limitano a far **rispettare le astrazioni** incorporate nel linguaggio (ad esempio, che `2+true` è un errore), ma anche quelle **definite dal programmatore**, cioè, oltre a "proteggere" la macchina dal programma, possono **incapsulare e proteggere parti del programma l'una dall'altra**.

Convenzionalmente un ADT è costituito da:

- il nome del tipo astratto A;
- il tipo di rappresentazione concreta T;
- implementazioni di operazioni per la creazione, l'interrogazione e la manipolazione di valori di tipo A;
- un **confine di astrazione** che racchiude T e lo rende accessibile solo attraverso le operazioni di A.

Pertanto, gli utenti possono creare nuovi valori di tipo A, passarli nel loro programma, memorizzarli in strutture dati e così via, ma non possono ispezionare e modificare direttamente la rappresentazione concreta di tipo T.

```
type Counter
 representation int
 signature
  new: Counter
  get: Counter -> int
  inc: Counter -> Counter
 operations
  new = 1
  get = fn( int i ){ i }
  inc = fn( int i ){ i+1 }
```

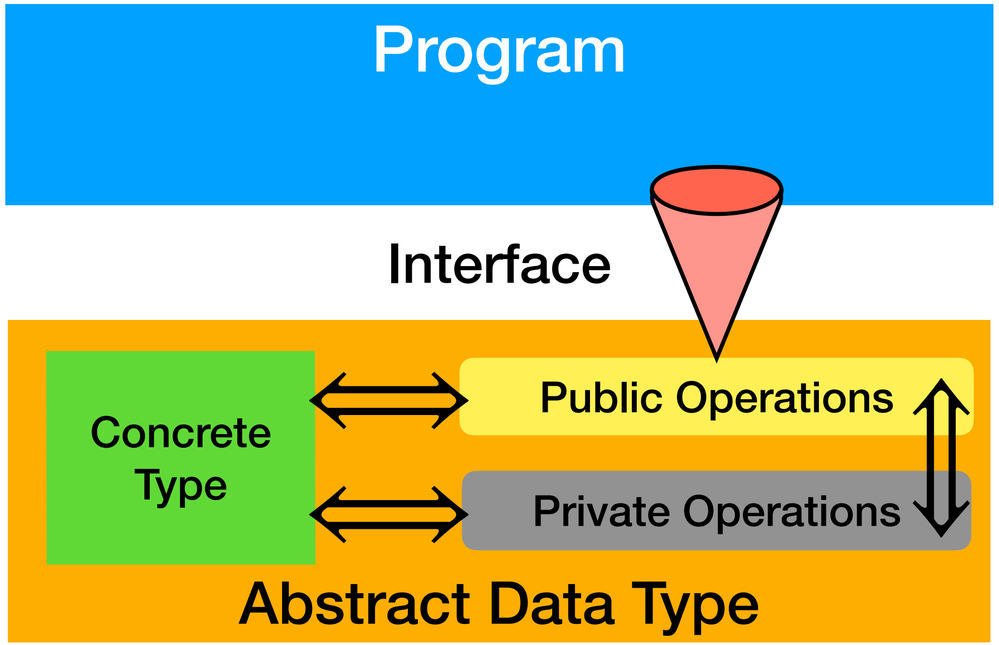
Using the CLU / Ada notation for Abstract Data Types

Nascondere le informazioni (information hiding)

L'astrazione delle implementazioni attraverso le **interfacce** è un principio fondamentale dei linguaggi di programmazione: consente al produttore e al consumatore di componenti software di **stabilire un contratto** concordando una “superficie" — l’interfaccia — che il consumatore si aspetta di utilizzare e che il produttore promette di supportare.

Seguendo la metafora, il "linguaggio" dei tipi di dato (astratto) è quello che le due parti usano per formalizzare questo contatto.

Un sinonimo di "astrazione", in questo caso, è **nascondere le informazioni**, cioè occultare dietro le interfacce i dettagli implementativi di tipo di dato (sia la sua struttura che il funzionamento delle sue operazioni).



Indipendenza della rappresentanza

È interessante notare che **l'information hiding migliora la componibilità del codice** in modo simile a quanto visto per il polimorfismo di sottotipo/parametrico: omettendo (alcuni) dettagli dell'implementazione (e.g., nel polimorfismo, il tipo effettivo di un parametro di tipo) noi (consumatori) non possiamo fare ipotesi sull'effettiva implementazione del tipo e possiamo scrivere codice che usa solo le interfacce date.

Questo ci porta alla proprietà dei type-safe ADT, chiamata **indipendenza dalla rappresentazione** (**representation independence**), secondo la quale **implementazioni corrette** (ben tipate) **dello stesso ADT sono osservabilmente indistinguibili dai consumatori dell'ADT**.

Poiché gli ADT sono indipendenti dalla rappresentazione, è sicuro utilizzare qualsiasi implementazione compatibile con il tipo di un dato ADT e scambiare implementazioni alternative. Ovviamente, le garanzie dei tipi riguardano solo quanto descrivibile/controllabile dai tipi, ad esempio non tengono conto delle differenze di prestazioni o di effetti non catturati dal sistema di tipi.

Abstract Data Types — Rust

```
trait Counter {
 fn new() -> Self;
 fn get( &self ) -> u32;
 fn inc( &self ) -> Self;
}
struct SC { counter: u32 }
impl Counter for SC {
    fn new() -> Self { SC { counter: 0 } }
    fn get( &self ) -> u32 { self.counter }
    fn inc( &self ) -> Self
     { SC { counter: self.counter + 1 } }
}
fn use_counter<C>( c: &mut C ) where C: Counter {
  let c = c.inc(c); let c = c.inc(c); print!( "{}", c.get() );
}
fn main(){ use_counter( &mut SC::new() ) }
```

segnatura/signature

implementazione

Rust implementa in modo diretto gli ADT tramite i **tratti** (**`trait`**).

In generale, i tratti sono un meccanismo linguistico per aggregare funzionalità comuni a più tipi.

Il tratto rappresenta il tipo astratto la cui interfaccia dà accesso agli utenti al tipo concreto, attraverso l'**`impl`**ementazione delle sue operazioni.

Moduli

Dal punto di vista dell'organizzazione del codice, gli ADT sono anche un modo per "aggregare" il codice (definizioni e logica delle operazioni) che appartengono allo stesso tipo (di dato astratto) in un’unico artefatto (linguistico).

Quando scriviamo librerie o progetti di grandi dimensioni, può essere necessario un modo per fornire più ADT all'interno dello stesso "pacchetto". Il meccanismo linguistico che permette di ottenere questo risultato è solitamente chiamato **pacchetto** (**package**) o **modulo**.

I diversi linguaggi di programmazione attribuiscono ai moduli proprietà diverse, ma una comune è quella di consentire agli sviluppatori di suddividere i programmi, in modo che ogni modulo contenga dati (tipi, variabili, ecc.) e operazioni (funzioni, codice, ecc.) disponibili all'interno del modulo. Il modulo definisce anche la **visibilità** dei dati che racchiude, cioè ciò che un utente del modulo può "vedere" e utilizzare.

Abstract Data Types e Tipi Esistenziali

Dal punto di vista della teoria dei tipi, i tipi di dato astratto e i moduli introducono il concetto di **tipi esistenziali** — come abbreviazione di **tipi quantificati esistenzialmente** — dove una definizione come

```
trait Counter { fn new…; fn get…; fn inc…; }
```

corrispondere al tipo esistenziale

$$CounterADT = \{\exists X, \{\text{new} : () \to X, \text{get} : X \to int, \text{inc} : X \to X\}$$

Si noti che il tipo non "vede" il tipo concreto, che è legato alla rappresentazione interna di un valore di tipo `Counter`. Questo indica che possiamo avere abitanti di $CounterADT$ di cui possiamo ignorare i dettagli interni, ottenendo comunque la type safety. Con l'esistenziale, finché abbiamo un abitante di quel tipo, possiamo "accedere" in modo sicuro alle operazioni del tipo.

Abstract Data Types e Tipi Esistenziali

Dato il tipo esistenziale

$$CounterADT = \{\, \exists X, \{\text{new} : () \to X,\ \text{get} : X \to int,\ \text{inc} : X \to X\}$$

Un'implementazione valida di $CounterADT$ è

$$\{Counter, c\} = \{\overbrace{*int, \{\text{new} = 1,\ \text{get} = \text{fn}(i : int)\{i\},\ \text{inc} = \text{fn}(i : int)\{i + 1\}\}}^{SC}\} \text{ as } CounterADT$$

$$Counter\ c_1 = c\,.\,\text{new}()$$

$$c\,.\,\text{get}(\ c\,.\,\text{inc}(\ c_1\ )\ )\ \text{// 2}$$

Il tipo concreto (riferimento) utilizzato dalle implementazioni delle operazioni, **assegnato a X in $Counter$**

Poiché la stessa implementazione può appartenere a tipi esistenziali diversi, introduciamo la parola chiave **as** per attribuire il tipo esistenziale previsto dell'implementazione.

Dopo aver "aperto" l'esistenziale (nella implementazione), il nome del tipo è vincolato, cioè possiamo avere una sola implementazione di $Counter$ nella continuazione del programma. Gli ADT garantiscono la possibilità di sostituire l'assegnazione di una data implementazione con una qualsiasi implementazione conforme senza rischiare la type-safety del nostro programma.

N.B. La definizione precedente di SC non gestisce uno stato interno (né lavora con i riferimenti) come l'esempio più complesso di Rust, che implementa una forma ibrida di ADT.

Moduli

Concettualmente, gli ADT e i moduli sono abbastanza vicini—possiamo vedere gli ADT come un caso degenere di un modulo che trasporta un ADT. Tuttavia, a seconda del linguaggio, gli ADT e i moduli possono fornire diversi gradi di flessibilità all'utente, ad esempio i moduli possono esprimere la visibilità dei dati e delle operazioni più finemente (la "permeabilità" della capsula) rispetto agli ADT.

```
mod counter { pub trait Counter { … } }
mod sc {
 use crate::counter::Counter;
 pub struct SC { … }
 impl Counter for SC { … }
}
use crate::counter::Counter;
use crate::sc:SC;
fn use_counter<C>( c:&mut C ) where C: Counter{ … }
fn main(){ … }
```

Oggetti Esistenziali

L'interpretazione esistenziale degli ADT non è legata a questi ultimi. Al contrario, osserviamo che **i tipi esistenziali catturano in generale l'information hiding e l'indipendenza dalla rappresentazione**.

Ad esempio, se usati per rappresentare gli ADT, il tipo esistenziale nasconde (omette) il tipo concreto solo fino a un certo punto. Quando usiamo un'implementazione, il nostro programma vincola l'implementazione al nome del tipo nella continuazione del programma, rendendo **impossibile avere più implementazioni interoperanti dello stesso tipo di dato astratto**. **Questo è un tratto distintivo degli ADT e non dei tipi esistenziali.**

$\{Counter, c\} = \{*int, \{\text{new} = 1,\ \text{get} = \text{fn}(i : int)\{i\},\ \text{inc} = \text{fn}(i : int)\{i + 1\}\}\}\ \text{as}\ CounterADT$

$\{ACounter, ac\} = \{\ *\{c : int\}, \{\text{new} = \{c : 1\},\ \text{get} = \text{fn}(i : \{c : int\})\{i\,.\,c\},\ \text{inc} = \text{fn}(i : \{c : int\})\{\{c : i\,.\,c + 1\}\}\}\}\ \text{as}\ CounterADT$

$c1 = c\,.\,\text{new}()$

$c2 = ac\,.\,\text{new}()$

$ac\,.\,\text{inc}(c1)$ Type mismatch error

Oggetti Esistenziali

Gli **oggetti** (esistenziali) forniscono una visione alternativa degli ADT, che **consente a più implementazioni dello stesso tipo esistenziale di interagire**.

A tal fine, è necessario che le definizioni di tipo esistenziale “vivano" nei programmi, in modo che $Counter = \{\exists X, \ldots\}$ sia un valore rappresentabile del sistema di tipi—al contrario degli ADT, dove l’esistenziale viene assegnato ad una implementazione e “sparisce”.

Con tale supporto, possiamo definire **oggetti** come **tipi concreti** che **mantengono il loro stato** (implementazione) **interno** e **portano con sé l'associazione con il loro tipo esistenziale**.

Oggetti Esistenziali

Modificando l'esempio di ADT fatto con $Counter$, abbiamo

$$Counter = \{\exists X, \{\text{state} : X, \text{methods} : \{\text{get} : X \to int, \text{inc} : Counter \to X\}\}\}$$

$$c1 = \{\ *int, \{\text{state} : 1,\ \text{methods} : \{\ \text{get}(int\ i)\{i\},$$
$$\text{inc}(\ \{*A, c\}\ \text{as}\ Counter\ )\{\ c\,.\,get(c) + 1\ \}\ \}\ \}\ \}\ \text{as}\ Counter$$

$$c2 = \{\ *\{c : int\}, \{\text{state} : \{c : 1\},\ \text{methods} : \{\ \text{get}(\ *\{c : int\}\ i)\{i\,.\,c\},$$
$$\text{inc}(\ \{*A, c\}\ \text{as}\ Counter\ )\{\ \{c : c\,.\,get(c) + 1\ \}\ \}\ \}\ \}\ \text{as}\ Counter$$

$$f(\ \{*A, a\}\ \text{as}\ Counter,\ \{*B, b\}\ \text{as}\ Counter\ )\{$$
$$\{*A,\ \{\ \text{state} :\ a\,.\,inc(b),\ \text{methods} : a\,.\,\text{methods}\ \}\ \}\ \text{as}\ Counter\ \}$$

$$f(c1,c2);\ f(c2,c1)$$

Dove le due implementazioni di $Counter$ possono coesistere, poiché il loro stato è sempre "nascosto" all'interno della loro implementazione e l'unico modo per leggerlo è tramite le operazioni.

Oggetti vs ADTs

Riassumendo, intuitivamente gli **ADT adottano una visione "aperta" dei tipi esistenziali** e delle implementazioni che essi inducono. Quando "importiamo" un'implementazione (o, in maniera complementare, quando ne "costruiamo" una) **la “apriamo” immediatamente, prima dell’uso effettivo**. Con gli ADT, i valori manipolati dal codice client (i programmi che importano l'implementazione) sono gli elementi del tipo di rappresentazione sottostante (nell'esempio del contatore, gli interi). Nell’esempio, a tempo di esecuzione, tutti i valori di `Counter` che implementano *CounterADT* sono elementi dello stesso tipo di rappresentazione interna e c'è un'unica implementazione delle operazioni sui contatori che può gestire tale rappresentazione interna (`Counter`).

L'opposto accade con gli **oggetti**, dove **manteniamo sempre "chiuso" un oggetto e usiamo i suoi metodi per accedere al suo stato interno**. Nel nostro esempio del contatore, ogni contatore è un oggetto che comprende il proprio tipo di rappresentazione e l’insieme di operazioni (che implementano l’interfaccia data dal tipo esistenziale) che funzionano per questo tipo di rappresentazione.

Oggetti vs ADTs

Quindi, una differenza rilevante degli oggetti rispetto agli ADT è che, poiché ogni oggetto ha la propria rappresentazione interna e implementa le proprie operazioni, **un programma può liberamente mescolare implementazioni diverse dello stesso tipo di oggetto (esistenziale)**.

Vedremo come questo diventi particolarmente conveniente in presenza di **sottotipaggio** (ed **ereditarietà**): possiamo definire un'unica classe generale di oggetti e poi produrre molti raffinamenti diversi, ognuno con una rappresentazione leggermente (o completamente) diversa.

Poiché le istanze di queste classi raffinate condividono tutte lo stesso tipo esistenziale, possono essere manipolate dallo stesso codice client, memorizzate insieme in liste, ecc.

Oggetti

Concentriamoci ora sul paradigma orientato agli oggetti e sulle peculiarità che lo determinano.

Il costrutto principale di ogni linguaggio orientato agli oggetti è quello di **oggetto**: **una capsula che contiene sia dati che operazioni per manipolarli e che fornisce un'interfaccia al mondo esterno attraverso la quale è possibile accedervi**.

Nel gergo degli oggetti, le operazioni sono chiamate **metodi** e possono accedere internamente ai dati contenuti nell'oggetto, raggiungibili tramite variabili interne, chiamati **campi**.

Nella loro proposta originale, le operazioni degli oggetti *ricevono un messaggio*, contenente il nome del metodo da eseguire e i suoi eventuali parametri. Tuttavia, (soprattutto per motivi di prestazioni) **la forma più comune di invocazione di operazioni avviene tramite l'invocazione di procedure** (da cui l'ampia adozione del termine "invocazione" per l'esecuzione di metodi) e **passaggio per riferimento**.

Oggetti

La sintassi dell'invocazione di un metodo non è nuova (è simile a quella degli ADT e dei record, in generale), dove `o.m( p1, p2, ... )` si legge come "invochiamo il metodo `m`, con i parametri `p1`, `p2`, ..., sull'oggetto `o`".

Poiché gli oggetti (come ADT) sono essenzialmente dei record, possiamo usare una sintassi simile per accedere alle loro variabili interne, ad esempio, con `o.v` intendiamo "accediamo alla variabile `v` dell'oggetto `o`".

In entrambi i casi, le implementazioni dei linguaggi orientati agli oggetti consentono di **esprimere in dettaglio l'opacità della loro capsula**: possiamo esprimere che le operazioni e le variabili **possono essere visibili ovunque**, alcune possono essere **visibili solo per alcuni oggetti** e altre possono essere **completamente private**, cioè disponibili solo all'interno dell'oggetto stesso.

Oggetti e Classi

Sebbene sia concettualmente ammissibile che ogni oggetto specifichi la propria implementazione, può diventare difficile da gestire e dispendioso specificare molte volte la stessa implementazione solo per avere **istanze diverse** dello stesso oggetto (ad esempio, immaginiamo di aver bisogno di tre contatori per tenere traccia di alcuni eventi distinti nel nostro codice, che poi vogliamo sommare).

Questo ha portato all'introduzione di un nuovo concetto, chiamato **classe**, che ci permette di **specificare un canovaccio o un modello di implementazione di riferimento che contiene le variabili e i metodi comuni alla stessa classe** (da cui il nome) **di oggetti**. Una volta definita una classe, per creare nuovi oggetti si invoca una operazione, di solito chiamata **new**, in grado di prendere la classe e di create—in gergo, “istanziare”—un nuovo oggetto a partire da essa.

Classi

Una classe è un modello per un insieme di oggetti: stabilisce quali sono i loro dati (quanti, di che tipo, con quale visibilità) e fissa il nome, la segnatura, la visibilità e l'implementazione dei suoi metodi.

In un linguaggio con classi, ogni oggetto "appartiene" ad (almeno) una classe, nel senso che la struttura dell'oggetto corrisponde alla struttura fissata dalla classe.

```
class Counter {
 private int x=1;
 public int get(){
  return x;
 }
 public void inc( int i ){
  x = x+i;
 }
}

Counter c1 = new Counter();
Counter c2 = new Counter();
Counter c3 = new Counter();
```

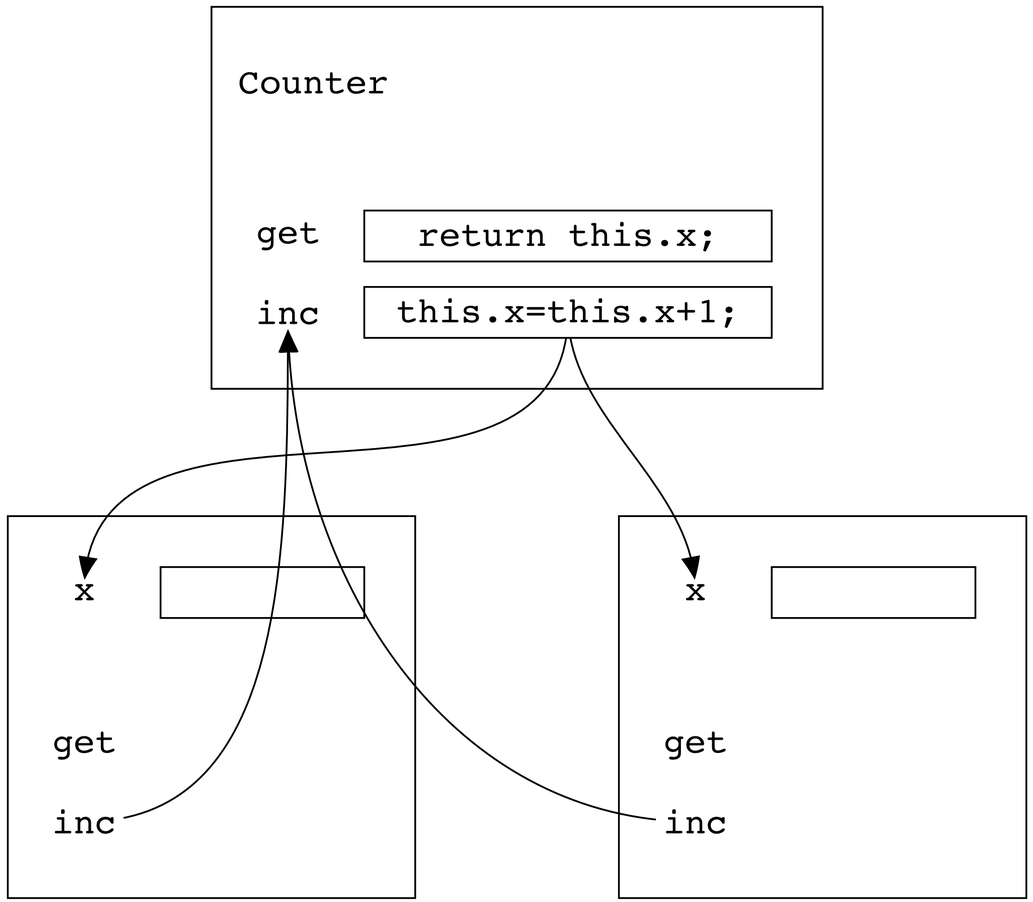
Classi

A sinistra, vediamo del codice **Java** di esempio (analogo a C++) dove una classe corrisponde a un tipo (ne parleremo più avanti) e tutti gli oggetti istanza della classe sono valori di quel tipo.

Ovviamente, l'interpretazione delle classi può cambiare a seconda del linguaggio. Ad esempio, in Simula, una classe è una procedura che restituisce uno stack frame (puntatore a un record di attivazione) contenente variabili locali e definizioni di funzioni, mentre in Smalltalk una classe è linguisticamente un oggetto, che serve come schema per la definizione dell'implementazione di un insieme di oggetti.

```
class Counter {
 private int x=1;
 public int get(){
  return x;
 }
 public void inc( int i ){
  x = x+i;
 }
}

Counter c1 = new Counter();
Counter c2 = new Counter();
Counter c3 = new Counter();
```

Classi



In generale, le classi memorizzano la (sola) implementazione del codice di tutti i suoi oggetti e quando invochiamo un metodo di un oggetto ci rivolgiamo all'implementazione legata alla classe, a cui fa riferimento l'oggetto.

Naturalmente, il metodo non viene eseguito sullo stato (inesistente) della classe, ma sullo stato dell'oggetto.

Classi

```
class Counter {
 private int x=1;
 public int get(){
  return this.x;
 }
 public void inc( int i ){
  this.x = this.x+i;
 }
}

Counter c1 = new Counter();
Counter c2 = new Counter();
Counter c3 = new Counter();
```

In pratica (come abbiamo visto in Rust, con il parametro **self**) i metodi degli oggetti ricevono implicitamente come parametro l'oggetto che li ha invocati, per cui quando il corpo del metodo fa riferimento a variabili di istanza, c'è un riferimento implicito all'oggetto corrente che sta eseguendo il metodo.

Da un punto di vista linguistico, l'oggetto corrente è solitamente indicato con un nome particolare, di solito **self** o **this**.

Gestire gli Oggetti in Memoria

Tutti i linguaggi orientati agli **oggetti** creano oggetti dinamicamente, il che di solito fa pensare a **strutture allocate sull'heap**.

Sebbene questo sia il caso di linguaggi come, ad esempio, Java—che alloca gli oggetti sull’heap, vi accede per riferimento (il linguaggio non ha un tipo puntatore e le variabili non-primitive sono riferimenti a oggetti memorizzati sull'heap) e utilizza garbage collection per gestire la loro deallocazione—altri linguaggi, come C++, consentono di specificare l'allocazione e la deallocazione degli oggetti sullo stack.

Classi vs Prototipi

Un'alternativa alle classi sono i **prototipi**. Questo stile prende anche il nome di **delegazione**, poiché si basa sulla **possibilità che gli oggetti deleghino parti della loro implementazione ad altri oggetti**.

Javascript (JS) è uno dei più famosi linguaggi basati sui prototipi.

Nei prototipi, gli oggetti possono delegare la definizione di valori e metodi a un altro oggetto, tramite la loro proprietà “prototipo”. Questi linguaggi offrono due modi per creare nuovi oggetti. Uno, chiamato **ex-nihilo**, avviene attraverso una forma di letterale dell'oggetto (ad esempio, { ... } in JS) e non assegna alcun prototipo all'oggetto creato. L'altro, chiamato **clonazione**, crea (ad esempio, attraverso la parola chiave **new**, in JS) un oggetto facendo una copia di uno esistente, che è il suo prototipo.

```
function Counter() {
 this.x = 1;
}
Counter.prototype.get=function(){
  return this.x;
}
Counter.prototype.inc=function(i){
  this.x = this.x+i;
}
c = new Counter();
c.inc( 1 );
c.get();
```

Prototype

Classi vs Prototipi

I prototipi sono simili alle classi, nel senso che servono come modelli per la struttura e il funzionamento di altri oggetti. Tuttavia, **contrariamente alle classi, i prototipi sono (linguisticamente) oggetti**, eventualmente utilizzati come modelli.

Analogamente alle classi, i prototipi possono definire metodi comuni ai loro deleganti, cioè, quando si cerca di accedere a un campo o di invocare un metodo di un oggetto, se l'oggetto non possiede quel campo o non definisce quel metodo, questo delega implicitamente l’azione al delegato. Se il delegato possiede quel campo o implementa quel metodo, allora esegue l'azione associata e riferisce al figlio il risultato. Al contrario, la catena di chiamate prosegue ai delegati dei delegati e così via, eventualmente fino a raggiungere il prototipo vuoto e a segnalare un errore.

Classi vs Prototipi

Una differenza pratica tra prototipi e classi sta nella flessibilità dei primi rispetto alle garanzie di sicurezza fornite dalle seconde.

Un oggetto basato su prototipi può cambiare il suo delegato a tempo di esecuzione (eventualmente cambiando completamente la sua interfaccia); un comportamento solitamente impedito o strettamente normato (e.g., vedremo, tramite sottotipaggio) nei linguaggi basati su classi.

```
function Counter() { this.x = 1; }
Counter.prototype.get=function()
 { return this.x; }
Counter.prototype.inc=function(i)
 { this.x = this.x+i; }
function OtherCounter(){Counter.call(this);}
function Multiplier(){}
Multiplier.prototype.mult=function( i )
 { this.x = this.x*i; }
OtherCounter.prototype = Counter.prototype;
c = new OtherCounter();
c.inc( 1 );
Object.setPrototypeOf(c,Multiplier.prototype);
c.mult( 2 );
Object.setPrototypeOf(c, Counter.prototype);
c.get(); // 4
```

Incapsulamento e Interfacce

L'incapsulamento e l'information hiding sono capisaldi degli ADT. Ciò vale anche per l'object orientation, dove i linguaggi OO permettono di definire un oggetto nascondendo parti di esso (i suoi dati e/o metodi).

Per questo motivo, si distinguono almeno due viste: quella **privata** e quella **pubblica**. La vista privata è quella più completa, dove tutti i metodi e i campi sono visibili. Quella pubblica vede solo le parti dell'oggetto che sono state esplicitamente esposte nella definizione dell'oggetto.

La vista pubblica di un oggetto viene solitamente chiamata **interfaccia**, con il significato indicato per gli ADT: i metodi (e i campi) che il codice client può utilizzare per interagire con il valore di un certo tipo.

Encapsulation 

Sottotipi

In OO, le classi identificano l'insieme di oggetti che sono sue istanze, cioè possiamo considerare le **classi come** i **tipi** di quegli oggetti. I linguaggi tipizzati rendono esplicita questa relazione: una definizione di classe introduce anche una definizione di tipo, i cui valori sono le istanze della classe.

Come detto, i sistemi di tipi sono solitamente dotati di una relazione di compatibilità. In particolare, possiamo vedere il sottotipaggio in OO come una relazione di compatibilità $S <: T$ dove il tipo associato alla classe S è un sottotipo del tipo associato alla classe T quando tutto il codice client che si aspetta di lavorare con oggetti di tipo T può lavorare con oggetti di tipo S.

Questo concetto, generalmente noto come principio di sostituzione di Liskov, è più formalmente specificato come "Sia $p(o)$ una proprietà dimostrabile per qualsiasi oggetto o di tipo T e sia S un sottotipo di T, allora, per qualsiasi oggetto o' di tipo S, $p(o')$ è dimostrabile."

Sottotipi

Concretamente, le proprietà del principio di sostituzione di Liskov si riferiscono alla possibilità di accedere a campi di S disponibili in T e di invocare metodi di S disponibili in T.

Mentre **in un sistema di tipi strutturali questa relazione si ridurrebbe a un controllo di corrispondenza tra gli elementi di S rispetto a quelli di T** (come si è visto per i record), la strada normalmente percorsa dai **linguaggi basati sulle classi è quella di adottare uno stile nominale**, che aiuta a evitare possibili equivalenze di tipo accidentali.

Questo ulteriore livello di sicurezza comporta delle limitazioni: **l'utente deve ora dichiarare le relazioni di sottotipaggio previste tra i tipi**, in modo che il linguaggio possa verificare che le proprietà ad esso legate siano valide.

Classi, Interfacce, Tipi e Sottotipi

In linea di principio, la relazione diretta che abbiamo stabilito in precedenza tra tipi e classi è un po' fuorviante: i tipi parlano di struttura e operazioni, mentre le classi definiscono anche vincoli di visibilità, mantengono lo stato e trasportano codice eseguibile.

È qui che entrano in gioco le **interfacce**, che **fanno da ponte (linguistico) tra classi e tipi**.

Per questo, **consideriamo** la definizione di **una classe come accompagnata da una definizione implicita di un'interfaccia della vista pubblica di quella classe**. A sua volta, la classe implementa tale interfaccia, stabilendo con essa una relazione di sottotipaggio.

```
interface CounterInterface {
 int get();
 void inc( int i );
}

class MyCounter
 implements CounterInterface {
  private int x=1;
  public int get(){ … }
  public void inc( int i ){ … }
  private void doubleInc( int i ){ … }
}
```

inherently public

Classi, Interfacce, Tipi e Sottotipi

Inoltre, le interfacce ci permettono di fornire ai clienti una **descrizione del "contratto" che i nostri oggetti promettono di soddisfare**, senza costringerci a fornire la loro effettiva implementazione.

Questa nozione costituisce un altro pilastro dei linguaggi orientati agli oggetti, che va sotto il nome di **principio di astrazione**.

```
interface CounterInterface {
 int get();
 void inc( int i );
}

class CounterUser {
 void incCounter( CounterInterface c ){
  c.inc( 1 );
 }
}
```

Abstraction 

Spesso aggregato al principio di incapsulamento

Sottotipaggio e Ereditarietà

La relazione interfaccia-classe non è l'unico modo per specificare il sottotipaggio in OO. Infatti, in linea di principio, il sottotipaggio da interfaccia a classe, ad esempio Counter **implements** CounterInterface, comporta due azioni:

- definiamo una classe, e.g., Counter, che **implementa** la sua interfaccia associata, chiamiamola *Counter_Interface* (distinta dalla CounterInterface nel codice a fianco);
- dichiariamo che *Counter_Interface* è/deve essere un sottotipo di CounterInterface.

Quindi, il sottotipaggio da interfaccia a classe presuppone una relazione di sottotipaggio da interfaccia a interfaccia, che di solito viene chiamata **estensione**.

```
interface CounterInterface {
 int get();
 void inc( int i );
}

interface MultCounterInterface
 extends CounterInterface
{
 void mult( int i );
}
```

Sottotipaggio and Ereditarietà

Quando applichiamo il **sottotipaggio a livello di classi**, cioè il sottotipaggio da classe a classe, **applichiamo l'idea di estensione da interfaccia a interfaccia per coprire anche lo stato, i vincoli di incapsulamento e l'implementazione dei metodi delle classi**.

Questa relazione prende il nome di **ereditarietà**, poiché il sottotipo della classe "eredita" da quest'ultima la sua definizione di stato (andremo nel dettaglio più avanti, parlando dei costruttori), i suoi vincoli di incapsulamento e le sue implementazioni di metodo.

```
class Counter {
 int x=1;
 public int get(){ return this.x; }
 public void inc( int i )
 { this.x = this.x+i; }
}
class MyCounter extends Counter {
 private void doubleInc( int i ){…}
}
class MultCounter extends MyCounter {
 public void mult( int i ){ … }
}

MultCounter mc = new MultCounter();
mc.inc( 1 );
mc.mult( 2 );
mc.doubleInc( 3 );
```

subclass superclass

subclass superclass

Inheritance 

Sottotipaggio e Ereditarietà • Shadowing/Mascheramento delle Variabili

Lo **shadowing delle variabili** indica che una sottoclasse può "mascherare" i campi della sua superclasse (da cui eredita) definendo campi con lo stesso nome (ma non necessariamente con gli stessi tipi).

Il mascheramento delle variabili deriva dalle regole standard di scoping a livello di blocco, con la particolarità di considerare la superclasse come un blocco esterno alla sottoclasse, di cui il blocco interno (la sottoclasse) può mascherare qualsiasi variabile.

Per evitare errori, Java adotta una **notazione esplicita**: se una sottoclasse vuole accedere a una variabile della sua superclasse, deve usare il prefisso **super** (analogamente a come **this** è un riferimento all'oggetto stesso).

```
class Counter {
 int x=1;
 int get(){ return x; }
 void inc( int i ){ x = x + i; }
}

class MultCounter extends Counter {
 int x=2;
 void multMult( int i ){
  super.x = super.x * this.x * i;
 }
}
MultCounter mc = new MultCounter();
mc.inc( 1 );
mc.multMult( 2 );
mc.get(); // 8
```

Sottotipaggio ed Ereditarietà • Overriding dei Metodi

Come già visto per le interfacce, una classe che estende un'altra classe può aggiungere nuove parti di stato e nuovi metodi (con i relativi vincoli di incapsulamento), oltre a quelle ereditate dal suo supertipo.

Un'altra peculiarità dell'ereditarietà è la possibilità di fare l'**override** ("sovrascrivere") delle implementazioni dei metodi ereditati.

Questa possibilità è data, tra l'altro, dal principio di astrazione, che dice che possiamo cambiare le implementazioni dei metodi purché rispettino le interfacce.

```
class Counter {
 int x=1;
 int get(){ return x; }
 void inc( int i ){ x = x + i; }
}
class MultCounter extends Counter {
 void mult( int i ){
  super.x = super.x * i;
}
 @Override
 void inc( int i ){ mult( i ); }
}
Counter c = new MultCounter();
c.inc( 2 );
c.inc( 3 );
c.get(); // 6
```

Override annotation

Sottotipaggio ed Ereditarietà • Overriding dei Metodi e Variable Shadowing

Una differenza tra l'**overriding dei metodi** e lo **shadowing delle variabili** è che il primo è **risolto dinamicamente**, mentre il secondo è **risolto staticamente**.

Nell'overriding di un metodo, **è la classe effettiva dell'oggetto** (da quale classe è stato istanziato) **a determinare a quale metodo verrà inviata l'invocazione**.

Al contrario, lo shadowing delle variabili viene risolto staticamente, cioè indicando esplicitamente a quale variabile si vuole accedere (this, super) o tramite type cast/ coercion.

```
class Counter {
 int x=1;
 int get(){ return x; }
 void inc( int i ){ x = x + i; }
}
class MultCounter extends Counter {
  int x=1;
  void mult( int i ){
  super.x = super.x * i;
}
 @Override
 void inc( int i ){ mult( i ); }
}
MultCounter mc = new MultCounter();
Counter c = mc;
c.inc( 2 );
c.inc( 3 );
c.get();  // 6
mc.get(); // 6
c.x;      // 6
mc.x;     // 1
```

Override annotation

This is one of the reasons behind the practice of setter and getter methods

Sottotipaggio ed Ereditarietà • Raffinamento della Visibilità

Abbiamo visto due nozioni di visibilità dell'incapsulamento (chiamate anche **modificatori di visibilità**): **privato** e **pubblico**. Sebbene questa divisione binaria copra il caso base dell'incapsulamento trasparente-vs-opaco, ci sono casi in cui potremmo voler definire l'incapsulamento come semi-opaco, ad esempio, per consentire alle sottoclassi di "vedere" metodi e campi delle loro super-classi. Questo è il caso coperto da due modificatori di visibilità aggiuntivi: **package** e **protected**.

Il caso **package** (quello predefinito, in Java) estende la visibilità dei campi/metodi di una classe a tutte le classi che appartengono allo stesso modulo di quella classe (ricordando la relazione tra ADT, moduli e tipi esistenziali).

Il caso **protected** estende il caso package per consentire a qualsiasi sottoclasse (in qualsiasi modulo) di interagire (internamente) con i campi/metodi protetti della propria super-classe.

```
class Counter {
 int x=1;
 public int get(){ return this.x; }
 public void inc( int i )
 { this.x = this.x+i; }
}
class MyCounter extends Counter {
 package void doubleInc( int i ){…}
}
class MultCounter extends MyCounter {
 public int mult( int i ){ … }
}

MultCounter mc = new MultCounter();
mc.inc( 1 );
mc.mult( 2 );
mc.doubleInc( 3 );
```

subclass superclass

subclass superclass

Sottotipaggio ed Ereditarietà • Riassumendo

È importante tenere a mente la differenza tra la relazione di ereditarietà e quella di sottotipo:

- i **sottotipi** hanno a che fare con la **possibilità di utilizzare un oggetto in un altro contesto**: è una relazione tra le **interfacce** di due classi;
- **l'ereditarietà** ha a che fare con la **possibilità di riutilizzare il codice che manipola un oggetto**: è una relazione tra le **implementazioni** di due classi.

Pur essendo indipendenti, queste due relazioni spesso si mescolano, a causa delle convenzioni che i linguaggi impongono, ad esempio che l'estensione da classe a classe (ereditarietà) implica la relativa estensione da interfaccia a interfaccia (sottotipaggio).

```
class Counter {
 int x=1;
 int get(){ … }
 void inc( int i ){ x = x + i; }
}

class MultCounter extends Counter {
 int x;
 int multMult( int i ){
  super.x = super.x * this.x * i;
 }
}

MultCounter mc = new MultCounter();
mc.inc( 1 );
mc.multMult( 2 );
```

Classi Astratte

In alcuni casi, l'ereditarietà è una relazione troppo stretta: ad esempio, potremmo avere un insieme di classi che implementano 1) la stessa interfaccia e 2) potrebbero condividere parte della loro implementazione *ma* non sono in una relazione di ereditarietà tra loro, cioè sono "fratelli".

Le **classi astratte** rappresentano una via di mezzo tra le interfacce e le classi, in quanto possono definire campi e implementazioni di metodi, ma anche lasciare alcuni metodi astratti, come le interfacce, che le sottoclassi dovrebbero implementare/sovrascrivere.

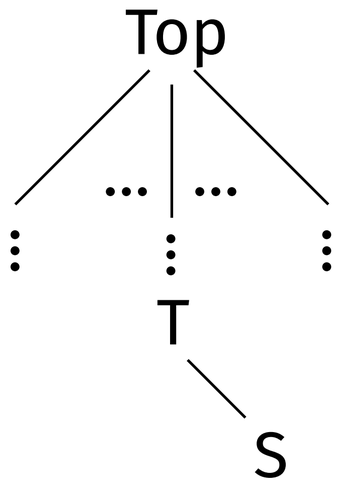
```
abstract class AbstractCounter {
 int x=1;
 int get(){ return x; };
 abstract void inc( int i );
}

class Counter extends AbstractCounter {
 @Override
 void inc( int i ){
  super.x = super.x + i;
 }
}
```

Relazione di Sottotipaggio, Top

Quando abbiamo presentato la relazione tra sottotipi $<:$, abbiamo detto che è un **preordine** (riflessiva, $T <: T$, e transitiva, $S <: T \land R <: S \implies R <: T$) **parziale** (antisimmetrico, $S <: T \land T <: S \implies T = S$)

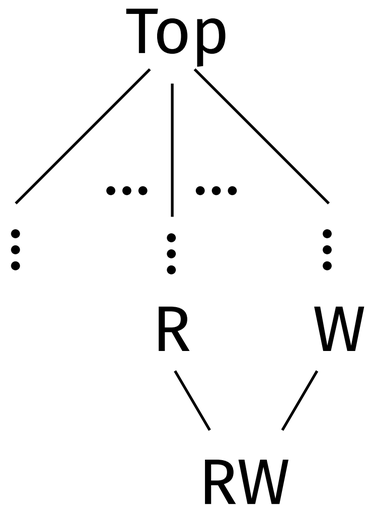
Se definiamo cicli come $T <: R, S <: T, R <: S$, la proprietà di antisimmetria li invaliderebbe ($S <: T \land R <: S \implies R <: T$, ma dato che $T \neq R$, nega $T <: R \land R <: T \implies T = R$).
Pertanto, la relazione di sottotipaggio assume la forma di un **grafo aciclico diretto** (DAG) **tra tipi**.

Gli ordini parziali (e i relativi DAG) non garantiscono la presenza di un singolo elemento massimo, solitamente chiamato **Top**, che non ha un super-tipo e che fa da padre (attraverso la transitività) per tutti gli altri tipi. Tuttavia, avere Top nel sistema dei tipi è generalmente utile (ad esempio, per specificare operazioni che accettano valori di qualsiasi tipo) e molti linguaggi impongono l'esistenza di Top, ad esempio, in Java, Top è **`Object`**, da cui tutte le altre classi ereditano (metodi di base a livello di oggetto, come la clonazione e il controllo di uguaglianza per “riferimento”).



Relazione di Sottotipaggio, Tipi Intersezione

Poiché la relazione di sottotipaggio descrive un DAG, in linea di principio è possibile esprimere qualcosa come $S <: T \wedge S <: R$, cioè che S è sia sottotipo di T e di R, dove T e R non sono nella relazione. Visualmente, stiamo facendo convergere la parte del DAG che riguarda T e R verso S.



In generale, questo tipo di tipi prende il nome di **tipi intersezione**, scritti $S <: T \wedge R$, che deriva dall'osservazione che, quando i valori di T e R si sovrappongono (anche parzialmente), S **indica l'intersezione dei valori che abitano sia** T **che** R**.**

D'altra parte, se T e R non si sovrappongono, S rappresenta una sorta di unione (che è diversa dai tipo Union/Somma) delle capacità dei due tipi. E.g., un tipo RW che è un sottotipo sia di Reader (R) che di Writer (W). In Java, questo è espresso, a livello di interfacce, con una **lista di interfacce che un'interfaccia estende (extends)**.

```
interface Reader {Readable read()}
interface Writer {void write(Writable w)}
interface RW extends Reader, Writer {}
```

Costruttori

Come detto, le classi offrono un modo per definire modelli per la creazione di oggetti e alcuni linguaggi OO forniscono una parola chiave speciale, ad esempio **new** in Java, per istanziare un oggetto. Tuttavia, possono esistere **procedure diverse per creare un oggetto**, ad esempio, potremmo voler creare un contatore impostando x, invece di utilizzare il valore iniziale predefinito 1.

I **costruttori** rispondono a questa esigenza e possono essere visti come metodi speciali di una classe che possono accettare alcuni parametri e restituire un oggetto istanziato di quella classe.

```
class Counter {
 int x;
 public Counter(){
  this.x = 1;
 }
 public Counter( int i ){
  this.x = i;
 }
 public get(){
  return this.x;
 }
 public inc( int i ){
  this.x = this.x + i;
 }
}
```

Costruttori

Sia a livello di tipi che di implementazione, gli oggetti sono strutture complesse e la loro creazione non fa eccezione: a) occorre **allocare la memoria necessaria** (sull'heap o sullo stack) e b) occorre **inizializzare correttamente i dati**.

L'azione b) è quella per cui esistono classi e costruttori: definiscono il codice la cui esecuzione garantisce la creazione di un'istanza corretta della classe. Questo codice diventa tanto più **complesso** quanto maggiori sono le funzionalità utilizzate, ad esempio, con l'ereditarietà b) non solo deve inizializzare gli elementi interni dell'oggetto (campi, puntatori ai metodi, ...), ma **deve anche collegare i dati dichiarati nelle super-classi**.

```
class Counter {
 int x;
 public Counter(){
  this.x = 1;
 }
 public Counter( int i ){
  this.x = i;
 }
 public get(){
  return this.x;
 }
 public inc( int i ){
  this.x = this.x + i;
 }
}
```

Costruttori • Scegliere un Costruttore

Come detto, le classi possono fornire diversi costruttori tra i quali il compilatore/runtime deve scegliere.

In alcuni linguaggi (ad esempio, C++, Java), il nome del costruttore coincide con il nome della classe e la distinzione tra questi **segue le stesse regole dei metodi sovraccaricati** (risolti staticamente, in base al numero e ai tipi di argomenti). Altri linguaggi permettono al programmatore di scegliere liberamente il nome dei costruttori, anche se questi rimangono sintatticamente distinti dai metodi ordinari.

```
class Counter {
 int x;
 public Counter(){
  this.x = 1;
 }
 public Counter( int i ){
  this.x = i;
 }
 public get(){
  return this.x;
 }
 public inc( int i ){
  this.x = this.x + i;
 }
}
```

Costruttori • Ereditarietà e Concatenamento dei Costruttori

Un altro aspetto riguarda come e quando inizializzare le parti di un oggetto che provengono da superclassi.

Alcuni linguaggi eseguono semplicemente il costruttore della classe di cui si sta creando l'istanza; se il programmatore desidera chiamare i costruttori delle super-classi, deve farlo esplicitamente.

Altri linguaggi (ad esempio, C++ e Java) impongono che l'inizializzazione di un oggetto richiami prima il costruttore della super-classe (concatenamento di costruttori). Anche in questo caso, il compilatore/runtime deve decidere quale dei molti costruttori disponibili della/e super-classe utilizzare.

```
class Counter {
 int x;
 public Counter( int i ){
  this.x = i;
 }
 public int get(){…}
 public void inc( int i ){…}
}
class MyCounter extends Counter {
 public MyCounter( int i ){
  super( i );
 }
}
```

Ereditarietà Singola vs Multipla

In alcuni linguaggi una classe può ereditare da una **sola super-classe** (immediata): **la gerarchia di ereditarietà è quindi un albero** e si dice che il linguaggio supporta **ereditarietà singola**. Questo è il caso di Java — n.b., *ereditarietà* singola convive coi tipi intersezione.

Altri linguaggi, invece, consentono a una **classe di ereditare metodi da più super-classi**; in questi linguaggi si parla di **ereditarietà multipla** e la **gerarchia di ereditarietà è un DAG** (come nel caso generale del sottotipaggio).

La maggior parte dei linguaggi supporta il caso più semplice dell'ereditarietà singola, ma alcuni, come C++ ed Eiffel, supportano l'ereditarietà multipla. Il motivo è che **l'ereditarietà multipla pone problemi concettuali e di implementazione** che non hanno ancora trovato una soluzione elegante.

Problemi della (e soluzioni per la) Ereditarietà Multipla

Concettualmente, abbiamo un problema di **conflitto di nomi**, che si verifica quando una classe eredita da due o più classi che forniscono l'implementazione di metodi con la stessa segnatura.

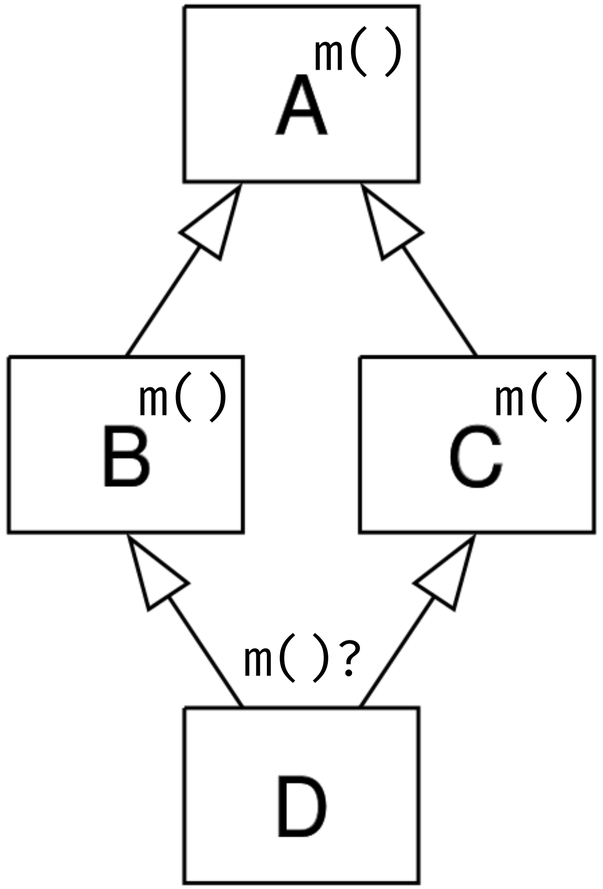
Come accennato, finora sono state proposte solo soluzioni parziali: a) **vietare sintatticamente i conflitti**; b) **chiedere allo sviluppatore di risolvere esplicitamente ogni conflitto**, ad esempio, qualificando in modo appropriato ogni riferimento al nome in conflitto (ad es, in C++, se C eredita da A e B dove il metodo m() è presente in entrambe; nel corpo di C, il programmatore deve chiamare B::m() o A::m()); c) **stabilire una convenzione** per risolvere i conflitti (come nei mix-in di Scala), ad esempio considerando come "vincente" il metodo della classe più a sinistra che compare nella clausola di estensione.

Problemi della (e soluzioni per la) Ereditarietà Multipla

Nella pratica, abbiamo un problema chiamato **Deadly-Diamond-of-Death**, dalla forma a diamante assunta dalla rappresentazione schematica della relazione di ereditarietà che emerge quando a) due classi B e C ereditano da A, e la classe D eredita sia da B che da C e b) c'è un metodo in A che B e C hanno sovrascritto, e D non lo sovrascrive.

Quale delle implementazioni del metodo eredita D?

Possiamo seguire una delle soluzioni concettuali (esclusa la prima) menzionate in precedenza. Sebbene queste soluzioni risolvano il problema dal punto di vista architetturale, rimane il problema pratico di risolvere in modo efficiente i conflitti ed eseguire l'implementazione corretta.



Dispatch Dinamico dei Metodi

Il dispatch dinamico dei metodi, chiamato anche **overriding** dei metodi, è il cuore del paradigma orientato agli oggetti; è **il punto in cui astrazione ed ereditarietà si incontrano e danno origine a uno dei tratti paradigmatici dell'orientamento agli oggetti**.

Concettualmente, il meccanismo è semplice: una sottoclasse può ridefinire (**override**) l'implementazione di un metodo, in modo che il codice eseguito dipenda dal tipo dell'oggetto che riceve il messaggio. L'accento sui tipi è importante: il dispatch è dinamico perché, in generale (ad esempio, in un metodo), conosciamo il tipo effettivo dell'oggetto solo a tempo di esecuzione.

Dynamic Dispatch 

```
class Counter {
 int x=1;
 int get(){ … }
 void inc( int i ){ x = x + i; }
}

class MultCounter extends Counter {
 void mult( int i ){ … }
 @Override
 void inc( int i ){ mult( i ); }
}

Counter c = new MultCounter();
c.inc( 2 );
c.inc( 3 );
```

Overriding, Overloading e Early/Late Binding

L'**overriding** dei metodi è simile all'**overloading**; infatti, entrambi risolvono una situazione ambigua in cui lo stesso nome può avere diversi significati.

La differenza è riassunta dai termini **early** e **late binding**. Nel **early binding** (dell'overloading) si utilizzano **informazioni statiche** (sul tipo delle variabili) per risolvere l'ambiguità e legare il nome. Al contrario, nel **late binding** le informazioni sono disponibili solo a **tempo di esecuzione** (i tipi degli oggetti reali)

Dynamic Dispatch 

```
class Counter {
 int x=1;
 int get(){ … }
 void inc( int i ){ x = x + i; }
}

class MultCounter extends Counter {
 void mult( int i ){ … }
 @Override
 void inc( int i ){ mult( i ); }
}

Counter c = new MultCounter();
c.inc( 2 );
c.inc( 3 );
```

Metodi Statici

I **metodi statici** sono più specifici per un certo linguaggio di programmazione rispetto al dispatch dinamico, caratteristico del paradigma OO.

Ad esempio, alcuni linguaggi li forniscono come un **modo per indicare** (e ottimizzare) **i metodi che il compilatore può risolvere staticamente**, perché sono indipendenti dagli stati/variabili dell'istanza (dell'oggetto) (lo trascurano) e non dipendono dalla classe effettiva di un dato oggetto.

```
class Counter {
 int x=1;
 int get(){ … }
 void inc( int i ){ x = x + i; }
 static void inc( Counter c ){
  c.inc( 1 );
 }
}
class MultCounter extends Counter {
 static void inc( MultCounter c ){
  c.inc( 2 );
 }
}

Counter c = new MultCounter();
MultCounter.inc( c ); // Statically solved
c.inc( 1 );
c.get(); // 3
```

Metodi Statici

Un esempio di questi sono i metodi `static` di Java, ai quali si accede dalla classe che li definisce, piuttosto che da un'istanza (oggetto) di quella classe. Essendo risolti staticamente, **i metodi statici non possono essere sovrascritti, ma solo sovraccaricati**.

Tuttavia, **è** anche **possibile mascherare** (**shadow**) **i metodi statici** (in modo simile alle variabili) e accoppiarli con il sottotipaggio. In questo caso, la risoluzione segue la relazione di sottotipaggio, e.g., se abbiamo un metodo applicato da un tipo e da un suo sottotipo, disambiguiamo applicando il metodo specifico al tipo della variabile (non dell'oggetto), come mostrato a destra.

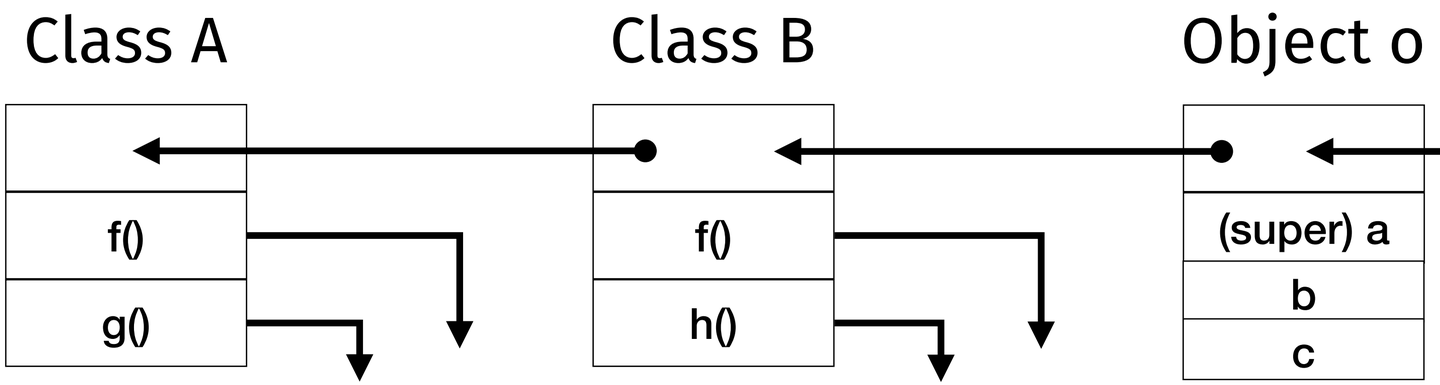
```
class Counter {
 int x=1;
 int get(){ … }
 void inc( int i ){ x = x + i; }
 static void inc( Counter c ){
  c.inc( 1 );
 }
}
class MultCounter extends Counter {
 static void inc( MultCounter c ){
  c.inc( 2 );
 }
}

Counter c = new MultCounter();
MultCounter.inc( c ); // Statically solved
c.inc( 1 );
c.get(); // 3
```

Aspetti di Implementazione • Oggetti

Concettualmente, le classi sono simili a record che definiscono campi e (il codice delle) operazioni. Questo vale anche per le implementazioni, dove possiamo rappresentare un oggetto come se fosse un record che contiene i **campi** della classe di cui è un'istanza, **più tutti quelli** che compaiono **nelle** sue **super-classi**.

In caso di **shadowing** (early binding), l'oggetto ha campi che corrispondono a una dichiarazione diversa (spesso il nome usato nella super-classe non è accessibile nella sottoclasse, se non tramite qualche qualificatore, per esempio super).

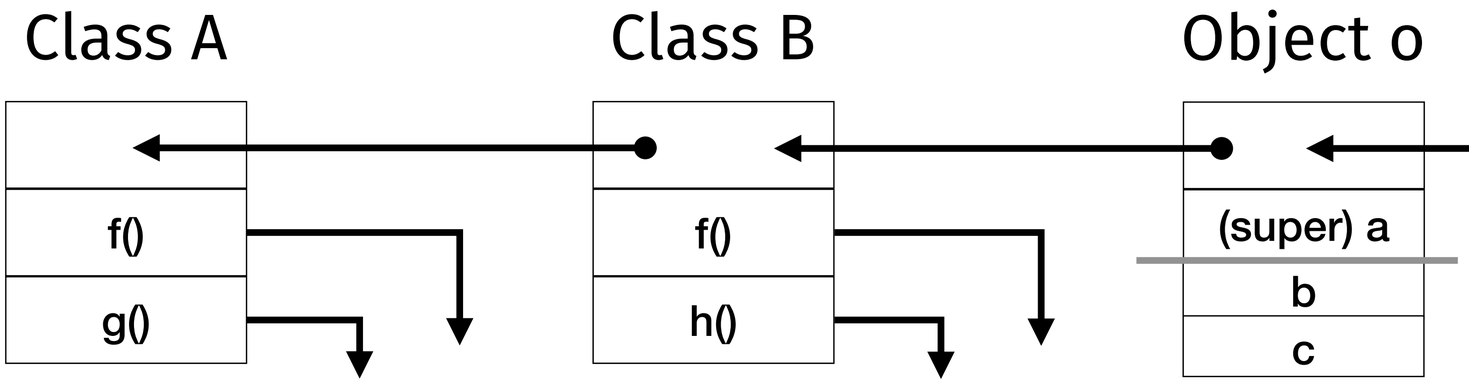
```
class A {
 int a;
 void f(){...}
 void g(){...}
}
class B extends A {
 int b; int c;
 void f(){...}
 void h(){...}
}
A o;
o = new B();
```



Aspetti di Implementazione • Oggetti

Nei linguaggi a tipaggio statico questa rappresentazione consente una semplice **implementazione della compatibilità** tra sottotipi (in caso di ereditarietà singola): poiché conosciamo staticamente l'offset (posizione) di ogni variabile, possiamo risolvere i riferimenti statici calcolando l'offset del blocco di partenza appartenente a una determinata super-classe e calcolando l'offset del campo referenziato da lì—ad esempio, per trovare `o.c`, sappiamo che dobbiamo iniziare dopo l'offset dei campi della classe A e poi seguire il tipo/ordine dei campi in B.

```
class A {
 int a;
 void f(){...}
 void g(){...}
}
class B extends A {
 int b; int c;
 void f(){...}
 void h(){...}
}
A o;
o = new B();
```



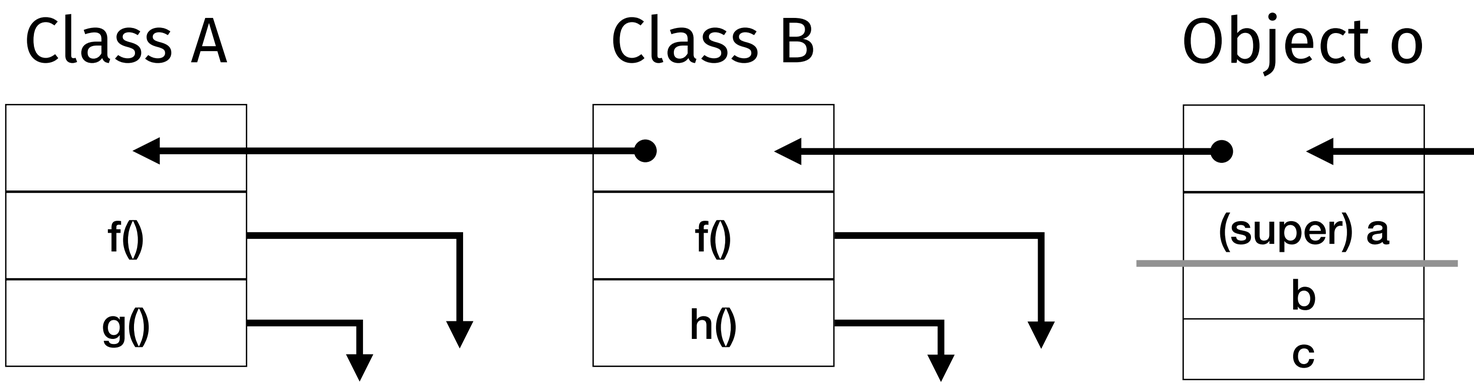
Aspetti di Implementazione • Classi e Ereditarietà

L'implementazione più semplice e intuitiva delle classi e dell'ereditarietà è quella di un **elenco concatenato**, dove ogni elemento: a) rappresenta una classe e contiene (puntatori a) l'implementazione di tutti i metodi esplicitamente definiti o ridefiniti in quella classe e b) punta alla sua immediata super-classe.

Per implementare il dispatch dinamico dei metodi, usiamo il puntatore di un oggetto alla sua classe per verificare se contiene un'implementazione per quel metodo: se lo contiene, eseguiamo il codice puntato lì, altrimenti seguiamo il puntatore fino alla super-classe di quella classe e così via.

Sebbene questa implementazione sia semplice, è anche piuttosto inefficiente, poiché il late binding implica la visita lineare della gerarchia delle classi.

```
class A {
 int a;
 void f(){...}
 void g(){...}
}
class B extends A {
 int b; int c;
 void f(){...}
 void h(){...}
}
A o;
o = new B();
```



Aspetti di Implementazione • Late *self* binding

L'esecuzione di un metodo è simile a quella di una procedura, in cui si caricano sullo stack le variabili locali, i parametri e le altre informazioni per la sua esecuzione. Tuttavia, a differenza delle procedure, i **metodi devono accedere** anche **alle variabili di istanza dell'oggetto su cui vengono invocati**, il cui indirizzo è noto solo a tempo di esecuzione.

Una soluzione inefficiente sarebbe quella di fare riferimento all'oggetto (`this`) nello stack frame del metodo e poi eseguire una doppia ricerca per trovare l'oggetto in memoria e quindi accedere ai campi di istanza.

Al contrario, possiamo evitare di caricare nello stack frame il riferimento a `this` e la doppia ricerca utilizzando la **conoscenza statica sulla struttura dell'oggetto**, data dalla sua classe: il compilatore può definire l'**accesso ai campi di istanza** non come un offset dallo stack frame (come avviene per le variabili/parametri locali) ma **come l'offset dato dall'indirizzo dell'oggetto corrente** (`this`) **più l'offset** (specifico) **di ogni campo**, come dichiarato dalla classe dell'oggetto.

Aspetti di Implementazione • Ereditarietà Singola

Dato un sistema di tipi statici, possiamo migliorare l'implementazione a tempo lineare e a lista concatenata della selezione dei metodi a ereditarietà singola in una a tempo costante.

Infatti, se i tipi sono statici, gli oggetti hanno un insieme finito e statico (a tempo di compilazione) di metodi, che corrispondono a quelli presenti nel descrittore della loro classe — che contiene sia i metodi esplicitamente definiti/ridefiniti nella classe sia tutti quelli ereditati dalle super-classi.

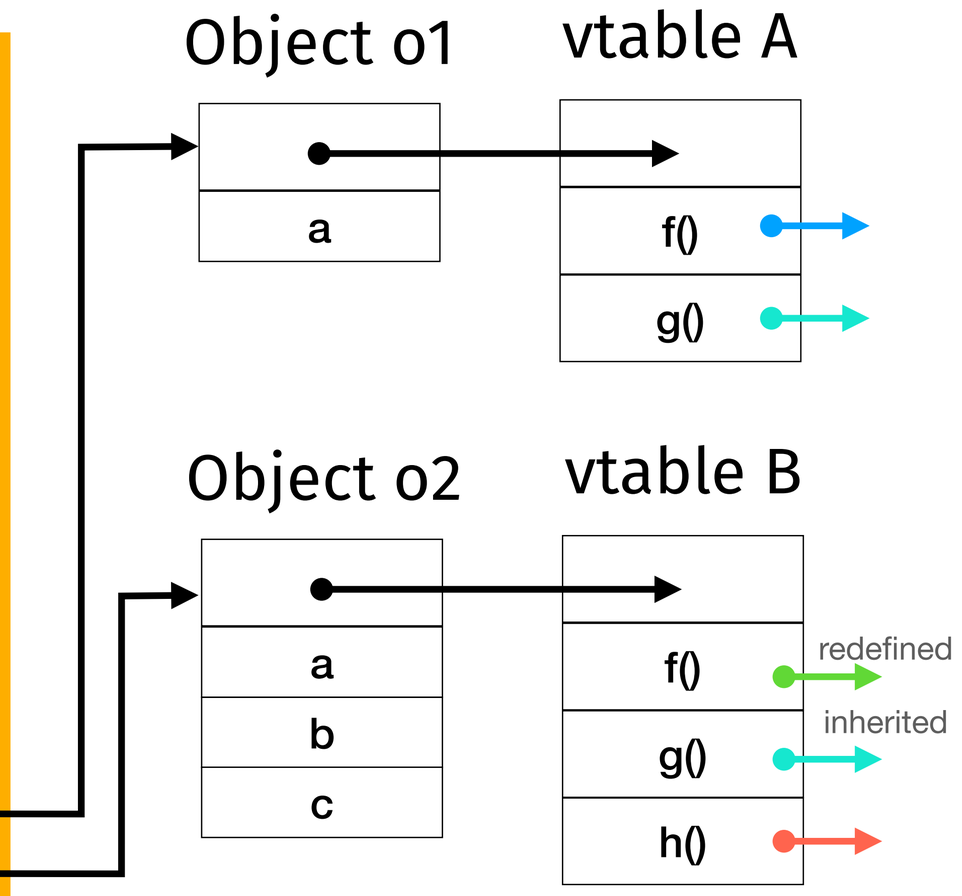
Questa struttura di dati prende solitamente il nome di **vtable** (dal C++, che sta per **virtual function table**).

Aspetti di Implementazione • Ereditarietà Singola

Con le vtables, **ogni definizione di classe corrisponde a una vtable** e tutte le istanze di quella classe condividono la stessa vtable.

Quando definiamo una sottoclasse B della classe A, costruiamo la vtable di B **facendo una copia della vtable** di A, sostituendo in questa copia tutti i metodi ridefiniti in B e **aggiungendo poi in fondo** alla vtable **i nuovi metodi** definiti in B.

```
class A {
 int a;
 void f(){...}
 void g(){...}
}
class B extends A {
 int b; int c;
 void f(){...}
 void h(){...}
}
A o1 = new A();
A o2 = new B();
```



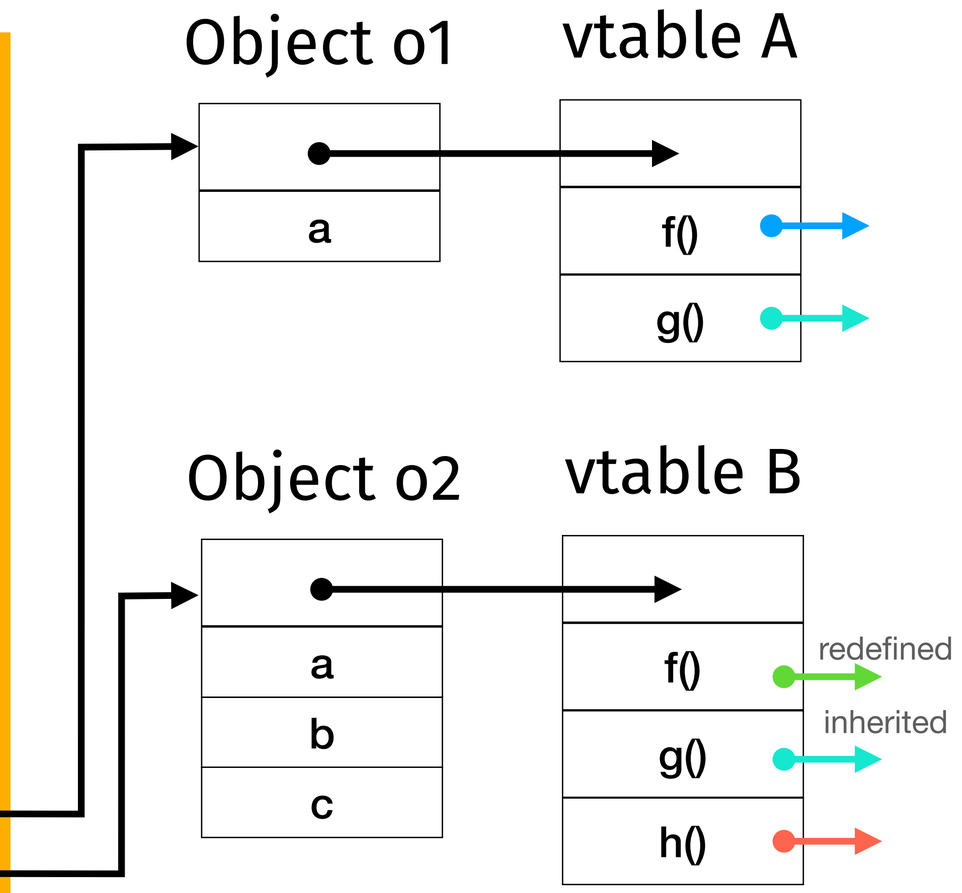
Aspetti di Implementazione • Ereditarietà Singola

In questo modo l'invocazione di un metodo:

A) avviene al prezzo (costante) di due accessi indiretti, poiché è noto staticamente l'offset di ciascun metodo all'interno della vtable e

B) tiene conto del fatto che si può accedere a un oggetto come a una delle sue super-classi; ad esempio, quando si invoca il metodo f, il compilatore calcola un offset per quel metodo che rimane lo stesso sia che f venga invocato su un oggetto di classe A sia che venga invocato su un oggetto di classe B, anche se, nella vtable, lo stesso indirizzo corrisponde a implementazioni diverse.

```
class A {
 int a;
 void f(){...}
 void g(){...}
}
class B extends A {
 int b; int c;
 void f(){...}
 void h(){...}
}
A o1 = new A();
A o2 = new B();
```



Aspetti di Implementazione • Classe Base Fragile

Le vtables per l'ereditarietà singola sono molto efficienti, poiché la maggior parte delle informazioni è determinata staticamente. Tuttavia, il late binding di `this` (`self`) è fonte di problemi in un contesto noto come problema della **classe base** (o super-classe) **fragile**.

In effetti, la propagazione dall'alto verso il basso dei cambiamenti imposti dalla gerarchia delle classi (dalle super-classi alle sottoclassi) rompe la composizionalità: l'unico modo per rilevare se alcuni cambiamenti in una super-classe hanno causato incompatibilità nelle sottoclassi comporta la necessità di considerare l'intera gerarchia di ereditarietà. Concettualmente, la modularizzazione rende impossibile questo controllo, poiché chi scrive la super-classe non ha accesso a tutte le possibili sottoclassi.

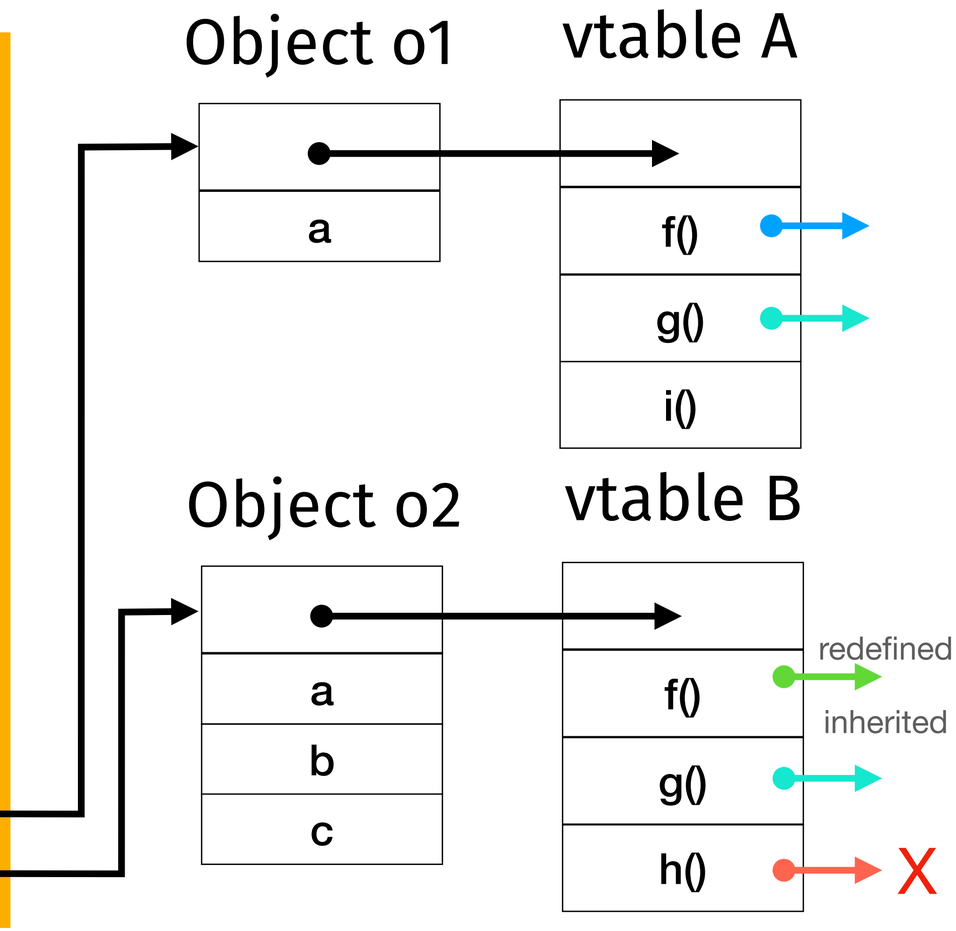
Dal punto di vista architetturale, si verifica un problema se qualche sottoclasse usa parti della super-classe che sono state modificate. Si tratta di un problema di ingegneria del software che può essere risolto limitando l'ereditarietà a favore del sottotipaggio.

Aspetti di Implementazione • Classe Base Fragile

Dal punto di vista dell'implementazione, il fallimento della sottoclasse dipende solo da come il compilatore ha rappresentato la gerarchia in memoria. Questo secondo caso prende il nome di **problema dell'interfaccia binaria fragile**.

Ad esempio, nel codice a fianco, aggiungiamo il metodo `i` ad A. Questo ci costringe a ricompilare anche B per aggiungere alla sua vtable il nuovo metodo, oppure rischiamo di incorrere in malfunzionamenti, ad esempio l'esecuzione di h al posto di `i`.

```
class A {
 int a;
 void f(){...}
 void g(){...}
 void i(){...}
}
class B extends A {
 int b; int c;
 void f(){...}
 void h(){...}
}
A o1 = new A();
A o2 = new B();
```



Aspetti di Implementazione • Dynamic Method Dispatch (JVM)

Il modo in cui la JVM implementa il **Dynamic Method Dispatch** supera il problema della classe base fragile, calcolando dinamicamente (e in modo efficiente) l'offset dei metodi nella vtable (ma anche delle variabili di istanza nella rappresentazione dell'oggetto).

Semplificando, Java compila le classi separatamente l'una dall'altra: ogni classe dà origine a un file che la macchina virtuale **carica dinamicamente** quando il programma in esecuzione fa riferimento a quella classe.

Questo file contiene una tabella di simboli (la **constant pool**) utilizzati nella classe stessa: variabili di istanza, metodi pubblici e privati, metodi e campi di altre classi utilizzati nel corpo del metodo, nomi di altre classi utilizzati nel corpo della classe, ecc.

Aspetti di Implementazione • Dynamic Method Dispatch (JVM)

Nel codice compilato, a ogni nome di variabile di istanza e di metodo sono associate informazioni, tra cui il tipo di nomi e la classe in cui sono definiti.

Per risparmiare spazio, ogni volta che il codice sorgente utilizza un nome, la intermediate representation della JVM **utilizza l'indice di quel nome nella constant pool** (e non il nome stesso). Questi indici non sono utili solo per la compattezza della rappresentazione.

Quando, in fase di esecuzione, **si fa riferimento a un nome per la prima volta** (attraverso il suo indice), **questo viene risolto**: la macchina virtuale carica le classi necessarie (ad esempio quelle in cui viene introdotto il nome) utilizzando le informazioni del pool di costanti, controlla i vincoli sui tipi e sulla visibilità (ad esempio, che il metodo invocato esista realmente nella classe a cui si fa riferimento, che non sia privato, ecc.) e poi riscrive il codice in esecuzione per sostituire le istruzioni di ricerca con istruzioni per l'esecuzione diretta del codice caricato in memoria.

Aspetti di Implementazione • Dynamic Method Dispatch (JVM)

Possiamo pensare che la rappresentazione dei metodi in un descrittore di classe sia simile a una vtable: la tabella di una sottoclasse inizia con una copia di quella della super-classe e sostituiamo i metodi sovrascritti con quelli ridefiniti dalla classe. Tuttavia, non calcoliamo subito gli offset, ma seguiamo queste quattro modalità di invocazione (legate alle loro istruzioni distinte nel bytecode):

- **`invokestatic`**: il metodo è **statico** e non può fare riferimento a `this`;
- **`invokevirtual`**: il metodo deve essere **selezionato dinamicamente** (i cosiddetti metodi “virtuali");
- **`invokespecial`**: usato o per **l’invocazione statica** di un costruttore, che inizializza i campi di `this`, o per **l’invocazione dinamica** di metodi privati, che non possono essere chiamati direttamente dall'esterno della classe, ma possono essere invocati da altri metodi della stessa classe stessa, riferendosi a `this`;
- **`invokeinterface`**: si chiama un metodo di interfaccia, che alcuni oggetti implementano.

Aspetti di Implementazione • Dynamic Method Dispatch (JVM)

invokestatic ha la risoluzione più semplice: poiché i metodi statici non possono essere sovrascritti e non fanno riferimento a variabili di istanza, ci limitiamo a vincolare staticamente la definizione della classe correlata.

invokespecial segue lo stesso percorso di `invokestatic`, con l'eventuale verifica dell'esistenza di super/this (e del relativo binding).

invokevirtual si occupa dell'overriding dei metodi e lo risolve attraverso la ricerca di vtable, ottimizzando le visite con la ricerca di indici, cioè calcolando che i metodi sovrascritti abbiano la stessa segnatura delle super-classi. Partendo dalla sottoclasse, si cerca l'indice nella vtable della classe e si verifica se è stato trovato il metodo che si stava cercando. In caso affermativo, ci fermiamo e risolviamo l'indice, altrimenti continuiamo a seguire il puntatore alla super-classe.

Aspetti di Implementazione • Dynamic Method Dispatch (JVM)

`invokeinterface` poiché ignoriamo quale classe implementa il metodo dell'interfaccia, ispezioniamo la classe dell'oggetto per determinare a) se quella classe implementa effettivamente l'interfaccia e b) dove sono registrati i metodi dell'interfaccia all'interno di quella particolare classe. Dato che non assumiamo uno schema fisso (che introdurrebbe il problema della classe base fragile), dobbiamo cercare nell'elenco delle interfacce implementate dalla classe. Una volta trovata l'interfaccia, possiamo procedere in modo diretto: dall'interfaccia, calcoliamo un **itable** che rappresenta lo schema fisso comune a tutte le porzioni di classi che implementano l'interfaccia di destinazione, e usiamo l'offset dalla itable per trovare il metodo e procedere come per le invocazioni dinamiche/virtuali.

`invokedynamic`, (per completezza) utilizzata da Java 8, l'istruzione aggiunge maggiore flessibilità ai meccanismi di dispatch dinamico della JVM ed è utilizzata principalmente nell'implementazione delle espressioni lambda di Java.

Polimorfismo Parametrico e Generici

Come visto, Java supporta il polimorfismo parametrico con la sintassi, e.g., `Set< T >`.

In particolare, Java adotta la nomenclatura **generics** [1] per indicare l'inclusione di questa caratteristica per supportare la programmazione generica—Java non è l'unico ad adottare lo stesso termine, altri esempi sono C#, F#, Python, Go, Rust, Swift e TypeScript. La distinzione con ciò che ML e Haskell chiamano *polimorfismo parametrico* è sottile, ma una differenza lessicale è che questi ultimi intendono il polimorfismo come implicito, ad esempio OCaml

```
let max x y = if x > y then x else y;; dove max: 'a → 'a → 'a
```

mentre i generici presuppongono l'indicazione esplicita dei parametri di tipo, ad esempio, Java

```
<T> max ( T x, T y ) { return x > y ? x : y; }
```

Anche C++ supporta un concetto simile a quello dei generici con i template.

[1] Bracha, G., Odersky, M., Stoutamire, D., & Wadler, P. (1998). Making the future safe for the past: Adding genericity to the Java programming language.

Generici e Type Erasure

Scrivendo `Set<T>` possiamo usare i generici per parametrizzare intere classi, ma come possiamo generare una versione parametrica di `Set`, in grado di "funzionare" con qualsiasi parametro di tipo?

Come nel caso del dispatch dinamico, i generici possono essere implementati in diversi modi. Ad esempio, C++ li implementa staticamente, dove il compilatore crea una copia separata del codice per ogni istanza utilizzata.

Grazie alla **cancellazione dei tipi** (**type erasure**), Java fa sì che tutte le istanze di una determinata classe generica condividano lo stesso codice. In fase di compilazione, se il type checker di Java convalida l'uso dei generici, il compilatore procede **cancellando tutti i parametri** di tipo dal codice (in modo che `Set<T>` diventi il tipo "raw" `Set`) e tutti gli oggetti della classe generica diventino istanze del tipo Top, `Object` [1]—dal momento che il type checker ha convalidato il programma, il compilatore non ha bisogno di aggiungere cast. Questo "trucco" ha permesso a Java 5 di introdurre i generici *senza rompere la compatibilità* con le versioni precedenti del linguaggio, le implementazioni della VM e le librerie.

[1] Questo esclude l'utilizzo dei tipi base di Java, che non estendono Object

Type Parameter Erasure

Sebbene la type erasure in Java abbia molti vantaggi, ha anche introdotto alcuni problemi.

Il più evidente è che non possiamo invocare **new** T( ), (parametro di tipo T), poiché il compilatore non sa quale oggetto creare. Allo stesso modo, il meccanismo di reflection di Java (**instanceof**) non è in grado di distinguere tra `Set<Integer>` e `Set<String>`, poiché in fase di esecuzione entrambi confluiscono nel tipo di raw `Set`—anche se esistono tecniche, generalmente indicate come **reificazione** (inteso come complemento dell'astrazione del tipo ottenuta attraverso la erasure), in cui la classe porta con sé il tipo/classe reificato (chiamato anche **witness**/testimone) del parametro di tipo.

```
class Box< T > {
 T c;
 Box( T c ){ this.c = c; }
}
class WBox< T > extends Box< T >{
 Class< T > klass;
 WBox( T c, Class< T > klass ) {
  super( c );
  this.klass = klass;
 }
}

Box< String > b1 = new Box<>( "a" );
Box< Integer > b2 = new Box<>( 1 );
WBox< String > wb1 =
  new WBox<>( "a", String.class );
WBox< Integer > wb2 =
  new WBox<>( 1, Integer.class );
```

Generici di Java • Wildcards

Per poter esprimere annotazioni di **varianza sui generici**, Java ha introdotto il carattere wildcard **?** come tipo speciale di argomento di tipo che esprime che T<?> è un super-tipo di qualsiasi applicazione del tipo generico T.

? può essere raffinato per indicare la co(ntro)varianza del sottotipaggio, cioè il caso covariante T<? **extends** S> consente l'uso di S e dei suoi sottotipi, mentre il caso controvariante T<? **super** S> consente l'uso di S e di tutti i suoi supertipi (si vedano gli esempi di covarianza e controvarianza sul polimorfismo parametrico vincolato).