01
Ingegneria e astrazione object-oriented

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2022/2023

Outline

Sistemi software

Obbiettivo di questo corso

Insegnare tecniche moderne per la costruzione di sistemi software
$\Rightarrow$ utilizzando i concetti di programmazione ad oggetti, che sono "mainstream"

Programma vs sistema software

Programma: un set di istruzioni che automatizzano la soluzione di una classe di problemi, spesso associato ad una visione algoritmica della computazione (Input $\rightarrow$ Output).
Sistema software: un'aggregazione di componenti di varia natura (programmi, librerie, parti del sistema operativo, basi di dati, interfacce grafiche, servizi Web, rete, dispositivi hardware) che cooperano per fornire una funzionalità computazionale.

Casi di sistemi software

Esempi

- Calcolatrice matematica
- Applicazione per gestire esami universitari
- Simulatore di circuiti elettronici, movimento di folle, movimento pianeti
- Monitor e elaborazione di dati biometrici (pressione, dati cardiaci,..)
- App per smartphone che mostra dove si trovano i miei amici di FB
- Video-game in cui personaggi vivono in un ambiente virtuale
- Controllore per dispositivi domotici (luce, termosifoni, acqua)

Caratteristiche comuni

- Non meri programmi, ma sistemi "pilotati" da software
- Includono componenti algoritmiche, ma anche interazioni complesse
- Alcuni realizzabili già alla fine di questo corso OOP, altri nei 5 anni

Fasi del processo di sviluppo del software

Analisi

Si definisce in modo preciso il problema da risolvere (non la soluzione!)

Design

Si definisce la struttura del sistema da sviluppare

- progetto architetturale + progetto di dettaglio
- si descrive la soluzione, ad uno specifico livello di dettaglio

Implementazione/codifica

Si realizza il sistema sulla base del progetto, scegliendo le tecnologie adeguate (efficienti, efficaci) – p.e. il linguaggio di programmazione

Post-codifica: Collaudo, Manutenzione, Deployment

Fasi necessarie, che spesso impiegano più del 70% delle risorse complessive

Il problema dello sviluppo di sistemi software

`http://it.wikipedia.org/wiki/Software_crisis`

I progetti software spesso falliscono!

- Tipicamente: tempi non rispettati...
- Più raramente: impossibilità a proseguire
- Spesso: i programmatori a lungo andare si deprimono

Quali cause possono comportare il fallimento di un progetto SW?

- Inadeguata analisi (requisiti non compresi appieno)
- Inadeguata (o assente) progettazione
- Cattive pratiche di programmazione
- Aspetti organizzativi nel team di sviluppo

Problem space vs solution space: il "buon progetto"

Problem space (fase di analisi)

L'insieme delle entità/relazioni/processi nel mondo "reale" sulla base delle quali si formula il problema che il sistema software deve risolvere

- per il gestionale per esami universitari: studente, corso, corso di laurea, appello, voto, iscrizione

Solution space (fase di progetto e implementazione)

Il corrispondente insieme di entità/relazioni/processi nel mondo "artificiale" che devono risolvere il problema (realizzate mediante i linguaggi e le tecnologie scelte e a fronte del progetto).

- funzione per il calcolo della media
- struttura dati per rappresentare i dati di uno studente
- form per visualizzare gli esami di uno studente

Un buon progetto mappa al meglio il problem space nel solution space

Concetti del solution/problem space: "livello di astrazione"

Definizione di astrazione (nell'informatica)

- È il metodo usato per descrivere la parte importante di un sistema informatico complesso con lo scopo di facilitarne la progettazione, implementazione e manutenzione: si basa dul mettere da parte alcune caratteristiche specifiche ritenute non essenziali, concenrtandosi su quelle invece cruciali.
- Un livello di astrazione è un insieme di concetti utilizzati per selezionare e descrivere in modo conveniente il sistema in oggetto

Il livello d'astrazione dei linguaggi di programmazione

Astrazione e programmazione

- Ogni linguaggio di programmazione introduce un livello di astrazione
- I concetti del sistema da realizzare (nel gestionale esami: corsi, voti, studenti, calcolo media voti) devono essere tradotti nei "costrutti" forniti dal linguaggio
- È facile? È conveniente? È flessibile? Dipende dal linguaggio..

Il livello di astrazione del C (e della progr. procedurale/imperativa)

- Stato del sistema — È costituito da strutture dati (costruite con tipi primitivi, array, puntatori e struct) tenute in stack e/o heap
- Dinamica — Esecuzione di procedure imperative, che si richiamano l'un l'altra
- Organizzazione — Librerie come set di funzioni, ricongiunte in un unico programma all'atto della compilazione/linking

Alcune caratteristiche/limiti del linguaggio C

Il C porta ad una visione piuttosto machine-oriented

- È un livello di astrazione fortemente influenzato dall'HW sul quale si eseguono i programmi (CPU, memoria)
- Ciò porta a
  - allocazione/deallocazione dinamica della memoria via librerie
  - difficile controllo degli errori di esecuzione
  - difficoltà a controllare gli aspetti HW-dependent
  - difficoltà a comprendere/gestire codice scritto da altri
- Nota: il C nasce negli anni '70 per rimpiazzare l'Assembly nell'implementazione del sistema operativo UNIX

La direzione dei linguaggi moderni – o di "alto livello" (d'astrazione)

- Introdurre un livello di astrazione vicino al problema da risolvere, ignorando il più possibile i dettagli dell'HW per risolverlo

L'ecosistema dei linguaggi di programmazione

Linguaggi e livelli di astrazione / paradigma

- C, Pascal: Computing function/procedure over data structures
- Lisp,ML: Everything is a function
- Prolog: Everything is a decision predicate
- Java,C++,C#: Everything is an object (OO Programming)

$\Rightarrow$ L'OOP si è dimostrata ideale per sistemi complessi general-purpose

L'evoluzione del "mainstream"

Machine Lang $\xrightarrow{'50-'60}$ Assembly $\xrightarrow{'70-'80}$ C $\xrightarrow{'90-2000}$ OOP (Java,..) $\xrightarrow{?}$?

Il futuro dei linguaggi (e anche la direzione dei linguaggi OOP)

- OO + funzionale + concorrenza: Scala, Kotlin, Java 8+

Il caso dei linguaggi dinamici...

- Python, Ruby, JavaScript: orientati a scripting
- molto diffusi in applicazioni Web, machine learning,...

I vantaggi della programmazione ad oggetti

Vantaggi

- Poche astrazioni chiave (classe, oggetto, metodo, campo)
- Usabili sia in progettazione che in codifica
- Supporto a estendibilità e riuso
- Supporto alla costruzione di librerie di qualità
- Facilmente integrabile in linguaggi C-like
- Eseguibile con alta efficienza

$\Rightarrow$ (quasi) tutti aspetti da sviscerare nel corso

Le critiche all'OOP

- Serve molta disciplina per "scalare bene" con la complessità del problema, ossia, per non incorrere in problemi di gestibilità al crescere della complessità del sistema da realizzare
- Altri paradigmi (p.e., funzionale) suggeriscono in parte come farlo...

Outline

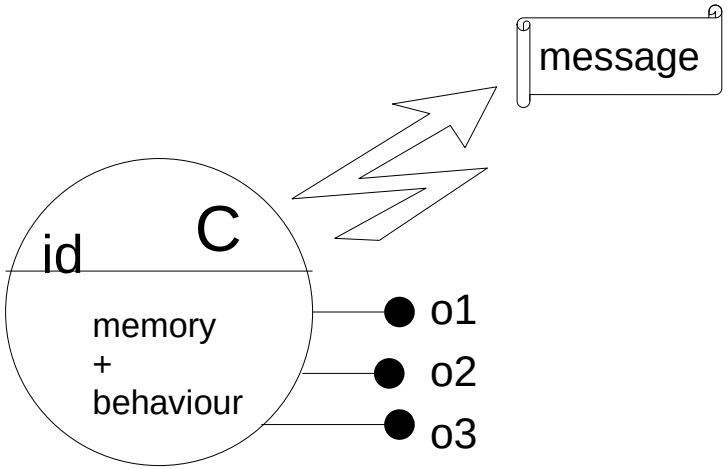
L'astrazione object-oriented (OO)

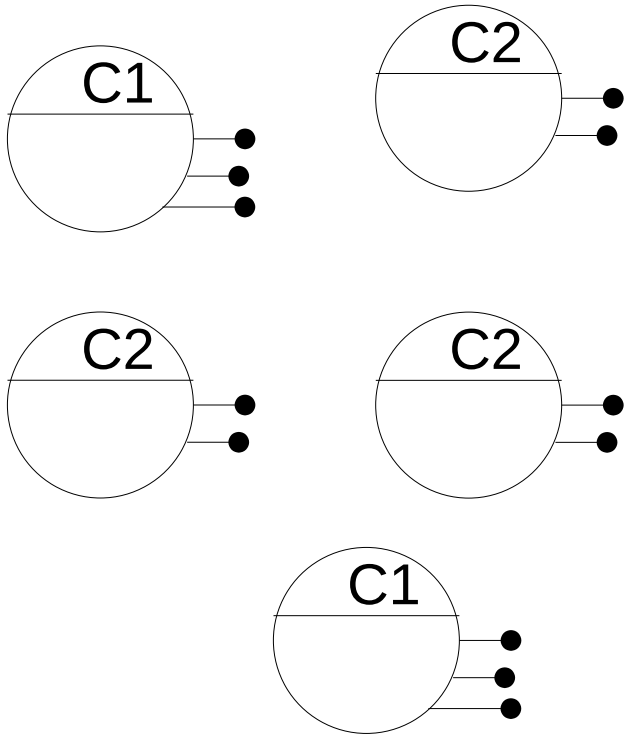
Un oggetto ha stato, comportamento e identità.

Definizione più dettagliata

1. **Everything is an object.** Un oggetto è una entità che fornisce operazioni per essere manipolata.
2. **Un programma è un set di oggetti che si comunicano cosa fare scambiandosi messaggi.** Questi messaggi sono richieste per eseguire le operazioni fornite.
3. **Un oggetto ha una memoria fatta di altri oggetti.** Un oggetto è ottenuto impacchettando altri oggetti.
4. **Ogni oggetto è istanza di una classe.** Una classe descrive il comportamento dei suoi oggetti.
5. **Tutti gli oggetti di una classe possono ricevere gli stessi messaggi.** La classe indica tra le altre cose quali operazioni sono fornite, quindi per comunicare con un oggetto basta sapere qual è la sua classe.

Oggetto e sistema ad oggetti





Sono questi concetti utili per il problem space?

Esempio di sistema reale: la gestione informativa di un ateneo

- Come è opportuno organizzarla? Quale servizi fornisce?

Visione object-oriented

- Un corso di laurea (CdL) è un "oggetto"
- Che operazioni consente? Iscrivi studente, laurea studente, assegna docente a corso, concludi anno accademico,..
- Un programma (sistema) è fatto da oggetti: facoltà, corso di laurea, corsi, studente, docente
- Le interazioni fra questi oggetti sono "scambi di messaggi"
- Il CdL ha uno stato fatto da altri oggetti: docenti, studenti, corsi
- Il CdL è istanza di una classe: la classe dei CdL (con una gestione comune) – UNIBO ne gestisce dozzine contemporaneamente

OO: problem space, solution space

L'esperienza mostra che:

- È piuttosto agevole modellare sistemi reali (o artificiali) come sistemi orientati agli oggetti
- Infatti, gli strumenti di modellazione standard usano il paradigma object-oriented! (Vedi UML)

Il vantaggio delle soluzioni object-oriented

- Consentono di "portare" il problem space nel solution space in modo diretto
- Usando gli stessi concetti object-oriented anche a livello di programmazione
- . . . ossia supportano il concetto di "buon progetto" che abbiamo discusso

Overview

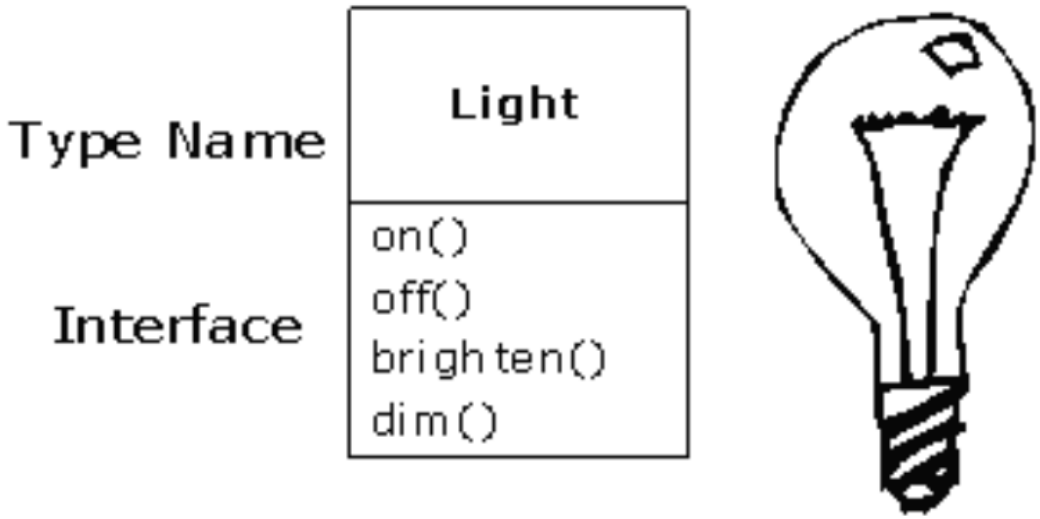
1. Ogni oggetto ha una interfaccia
2. Un oggetto fornisce un servizio
3. Un oggetto deve nascondere l'implementazione
4. Le implementazioni possono essere riusate
5. Il riuso tramite ereditarietà

Ogni oggetto ha una interfaccia

Classi, istanze, metodi, interfaccia

- Oggetti simili sono istanze della stessa classe, o tipo
- La classe definisce i messaggi ricevibili, attraverso metodi
- L'insieme dei metodi prende il nome di interfaccia
- Un messaggio ha effetto su stato e comportamento dell'oggetto

Esempio "lampadina" in notazione UML (Unified Modelling Language):



Un oggetto fornisce un servizio

Quale entità del problem space deve diventare un oggetto?

- Conviene considerare un oggetto come un fornitore di un servizo
- Tutto il programma può essere visto come un servizio dato all'utente
- Principio di decomposizione: i sotto-servizi sono affidati ai vari oggetti

Vantaggi di questo approccio

- Semplifica la progettazione degli oggetti, e il mapping col problema
- Semplifica il loro riuso in programmi diversi – come parte di libreria
- Semplifica la comprensione dei programmi, specialmente da terzi

Opportunità: $\Rightarrow$ Se non se ne riesce a descrivere il servizio, allora probabilmente un oggetto non ha ragione d'esistere!
Coerenza: $\Rightarrow$ Se sembra che un oggetto realizzi due servizi diversi, allora probabilmente bisogna in realtà realizzare due oggetti!

Un oggetto deve nascondere l'implementazione

Le due figure: creatore di classi vs. programmatore cliente

- Chi produce la classe (e ha la responsabilità del suo funzionamento)
- Chi usa la classe (per fornire un servizio più di alto livello)

Information hiding

- Il creatore rende visibile solo una piccola parte della classe
- Il resto è invisibile perché suscettibile di modifiche future
- Principio: "less is more"
- Tipica struttura di una classe
  - interfaccia: la sola visibile al cliente, di norma
  - campi della classe (i sotto-oggetti di cui è costituito)
  - implementazione metodi (cosa fa l'oggetto quando riceve messaggi)
    - come cambia lo stato (campi)
    - quali messaggi manda ad altri oggetti
    - quale risultato fornisce (risposta al messaggio)

Le implementazioni possono essere riusate

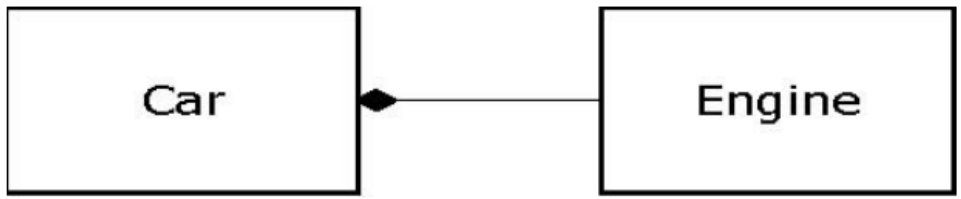
Fasi del processo di riuso

1. Il creatore produce una classe e ne verifica il corretto funzionamento: diventa una unità riusabile di codice
2. Un cliente può ri-utilizzarla per creare nuovi concetti

La tecnica di riuso più usata è la composizione

- Un nuovo oggetto è costituito da oggetti di altre classi
- Relazione chiamata "has-a" ("ha un")
- Questa relazione può essere nascosta, e resa dinamica

Esempio "car+engine" in notazione UML:

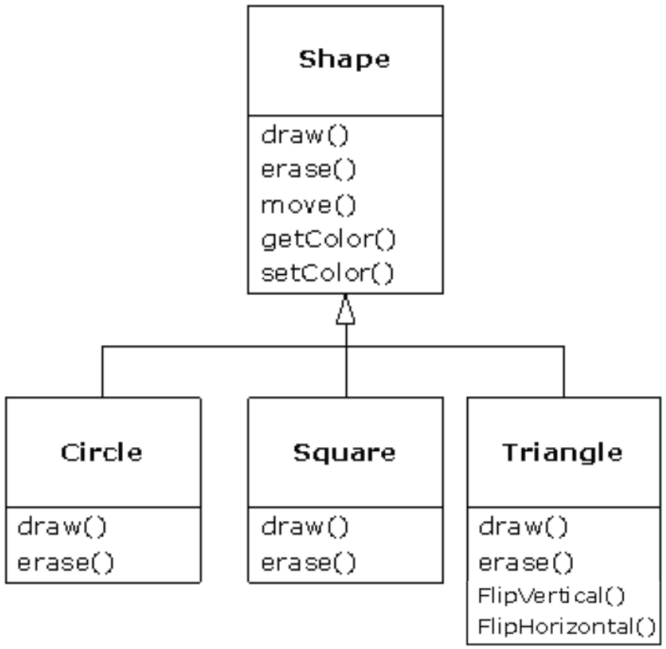


Il riuso tramite ereditarietà

È una ulteriore, fondamentale, tecnica di riuso

- Un nuovo oggetto(/classe) estende i servizi di uno esistente
- Fornisce i metodi della sopra-classe, ma anche altri
- Relazione chiamata "is-a" ("è un")

Esempio "shape" in notazione UML:



Altri aspetti – organizzazione del corso

Funzionalità base – I parte

- I concetti appena introdotti

Funzionalità aggiuntive – II parte

- Polimorfismo per genericità
- Gestione delle eccezioni
- Riflessività, classi innestate, enumerazioni

Tecniche Avanzate – III parte

- Librerie avanzate (I/O, GUI, Concorrenza)
- Integrazione con la programmazione funzionale
- Pattern di progettazione
- Prassi di programmazione efficace

Outline

Qualche nota introduttiva..

Java

- Linguaggio inventato da J. Gosling, alla Sun Microsystems (1995)
- . . . ora è gestito commercialmente da Oracle, con anche versioni "open" che sono comunque "reference" (OpenJDK)
- Una semplificazione del C++ (ossia C + OO), per sistemi embedded
- Filosofie: "Write once run everywhere" + "Keep it simple, stupid"
  - . . . la prima filosofia porta all'uso di Java sia su PC, che sistemi embedded (Raspberry), o mobile (Android)
  - . . . la seconda filosofia necessariamente abbandonata nel tempo.
- Al momento il terzo linguaggio più popolare secondo l'indice Tiobe
  - fonte: `https://www.google.it/search?q=tiobe+tcpi`
  - (indice usato per decisione strategiche nello sviluppo di nuovi SW)
- . . . Python (in crescita), C, Java sono gli unici sopra al 10%

In questo corso

- Useremo Java come riferimento per programmazione/progettazione
- Le guideline progettuali che daremo sono di validità generale
- Forniremo anche gli elementi base del linguaggio C#

Strumenti Java

Java Development Kit (JDK) – useremo la versione OpenJDK 17

- Insieme di tool per lo sviluppatore
- `https://adoptopenjdk.net/`

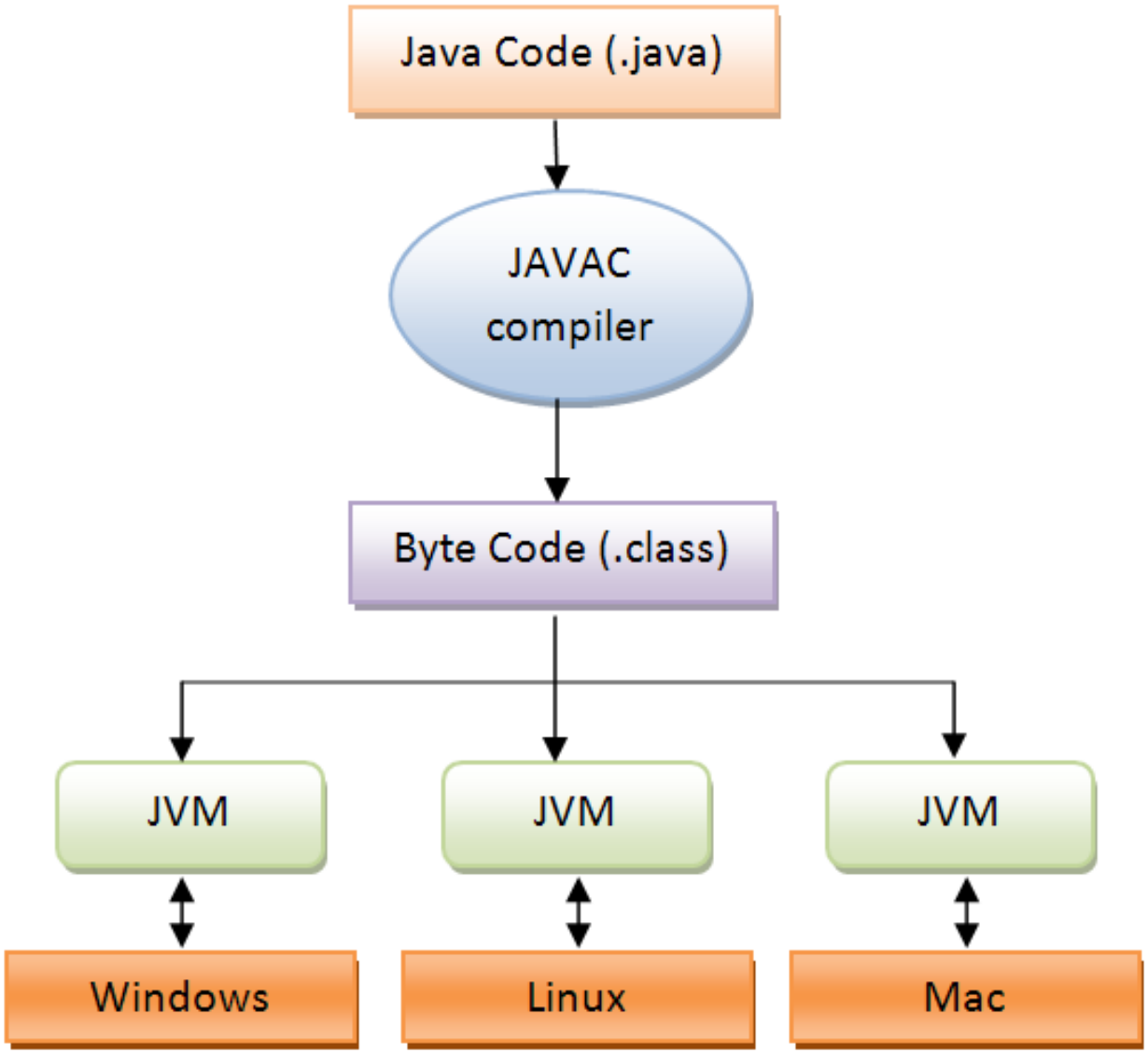
Java Virtual Machine (JVM) – inclusa nel JDK

- Un programma (C/C++) che carica i binari delle classi e le esegue
- Fornisce indipendenza dall'HW, e servizi aggiuntivi (SISOP, GC)

Schema compilazione/interpretazione

1. Compilazione dei programmi col compilatore Java (comando `javac`)
2. Esecuzione del programma con la JVM (comando `java`)

Write once run everywhere



Sviluppo storico di Java

Pricipali versioni

- Java (1996): versione base iniziale
- Java 2 (1998): aggiunta del framework Swing
- Java 5 (2004): aggiunta di generici, inner class, annotazioni
- Java 8 (2014): aggiunta lambda expression e streams
- Java 9-16 (2017-2021): aggiunte minori Java Module System,

$\Rightarrow$ Java 17 (sept. 2021): versione "long-term support" (LTS) che useremo

- ...

Modello di sviluppo attuale

- Da Java 9, si ha una nuova release ogni 6 mesi
- Java 17 è l'ultima LTS, e si raccomanda di non rimanere indietro
- Oracle detiene il marchio e una implementazione "commerciale"
- L'iniziativa OpenJDK produce una ulteriore implementazione open che è di riferimento – quella che useremo noi
- Esistono anche altre JVM (Eclipse OpenJ9, Amazon Corretto,...)