2. Processo di Sviluppo Software e Fondamenti OOP in Java

Questo capitolo esplora le fasi cruciali del ciclo di vita di un sistema software e introduce i pilastri della programmazione orientata agli oggetti (OOP) in Java, spiegando come gli oggetti e le classi siano i blocchi fondamentali per costruire software robusto e manutenibile.

2.1 Il Processo di Sviluppo del Software

La creazione di un sistema software non è un'attività lineare e improvvisata, ma un processo strutturato che si articola in diverse fasi interconnesse. Questo approccio sistematico è fondamentale per gestire la complessità, garantire la qualità e soddisfare i requisiti del cliente.

2.1.1 Fasi del Processo di Sviluppo

Il processo di sviluppo software è tipicamente suddiviso in una serie di fasi sequenziali, sebbene nella pratica queste possano sovrapporsi o essere iterate (come nei modelli agili). Le fasi principali sono:

1. **Analisi**: (da `01-oo-abstraction_slides.pdf`, Pag. 3)
    - **Obiettivo**: Comprendere a fondo i requisiti del sistema. Si tratta di capire *cosa* il sistema deve fare per risolvere il problema del cliente.
    - **Attività**: Raccolta dei requisiti (funzionali e non funzionali) attraverso interviste, questionari, analisi di documenti esistenti. Si definiscono le funzionalità, le prestazioni attese, i vincoli di sicurezza, l'usabilità, ecc.
    - **Output**: Documenti di requisiti chiari e non ambigui, spesso sotto forma di user stories, casi d'uso o specifiche funzionali.
2. **Design (Progettazione)**: (da `01-oo-abstraction_slides.pdf`, Pag. 3)
    - **Obiettivo**: Definire l'architettura del sistema e la struttura interna. Si decide *come* il sistema farà ciò che è stato specificato nell'analisi.
    - **Attività**: Si progetta la struttura generale del software (architettura), si definiscono i moduli, le classi, le interfacce e le relazioni tra essi. Si prendono decisioni su database, tecnologie, pattern di progettazione.
    - **Output**: Documenti di design (es. diagrammi UML, schemi di database), prototipi di interfaccia utente.
3. **Implementazione**: (da `01-oo-abstraction_slides.pdf`, Pag. 3)
    - **Obiettivo**: Tradurre il design in codice eseguibile.
    - **Attività**: Scrittura del codice sorgente utilizzando il linguaggio di programmazione scelto (nel nostro caso, Java). Questa fase include anche la compilazione del codice.
    - **Output**: Codice sorgente funzionante, file eseguibili.
4. **Collaudo (Testing)**: (da `01-oo-abstraction_slides.pdf`, Pag. 3)
    - **Obiettivo**: Verificare che il software soddisfi i requisiti e sia privo di difetti.
    - **Attività**: Esecuzione di test unitari (sulle singole componenti), test di integrazione (sull'interazione tra componenti), test di sistema (sul sistema completo) e test di accettazione (da parte del cliente). Si identificano e si correggono i bug.
    - **Output**: Rapporti di test, lista di bug, software validato.
5. **Deployment (Distribuzione)**: (da `01-oo-abstraction_slides.pdf`, Pag. 3)
    - **Obiettivo**: Rendere il software disponibile agli utenti finali.
    - **Attività**: Installazione del software sui server o sui dispositivi degli utenti, configurazione dell'ambiente, rilascio degli aggiornamenti. Questa fase include anche la manutenzione e il supporto post-rilascio.
    - **Output**: Software operativo nell'ambiente di produzione.

2.1.2 Problem Space (Dominio/Logica Business) vs. Solution Space (Scelte Realizzative)

Un concetto cruciale nello sviluppo software è la distinzione tra "problem space" e "solution space" (da `01-oo-abstraction_slides.pdf`, Pag. 3).

- **Problem Space (Dominio/Logica Business)**:
  - Rappresenta il **dominio del problema** che il software deve risolvere.
  - Si concentra sulla **logica di business** e sui requisiti dell'utente.
  - Descrive *cosa* il sistema deve fare dal punto di vista dell'utente o del dominio applicativo.
  - È il **livello di astrazione** più alto, focalizzato sul "perché" e sul "cosa".
  - *Esempio*: In un sistema di gestione universitaria, il problem space riguarda concetti come "studente", "corso", "esame", "iscrizione", e le regole che li governano.
- **Solution Space (Scelte Realizzative)**:
  - Rappresenta il **dominio della soluzione**, ovvero come il software verrà effettivamente costruito.
  - Si concentra sulle **scelte realizzative** e tecnologiche.
  - Descrive *come* il sistema sarà implementato, utilizzando linguaggi di programmazione, database, architetture specifiche.
  - È il **livello di astrazione** più basso, focalizzato sul "come".
  - *Esempio*: Nel sistema universitario, il solution space riguarda la scelta di Java per la programmazione, un database SQL per la persistenza dei dati, un framework web per l'interfaccia utente, e la definizione delle classi Java che rappresentano studenti, corsi, ecc.

La sfida è tradurre efficacemente i concetti del problem space in una soluzione tecnica nel solution space, mantenendo un'adeguata astrazione.

2.2 Astrazione Orientata agli Oggetti

L'astrazione è un principio fondamentale dell'ingegneria del software, che permette di gestire la complessità concentrandosi sugli aspetti essenziali e ignorando i dettagli irrilevanti per un dato livello di comprensione. La programmazione orientata agli oggetti (OOP) è un paradigma che implementa l'astrazione attraverso il concetto di "oggetto".

2.2.1 Oggetto: Stato, Comportamento e Identità (Incapsulamento)

Nel paradigma orientato agli oggetti, un **oggetto** è l'unità fondamentale. Ogni oggetto è caratterizzato da tre aspetti principali (da `01-oo-abstraction_slides.pdf`, Pag. 3):

1. **Stato (State)**:
   - Rappresenta i dati o le proprietà interne dell'oggetto.
   - È ciò che l'oggetto "sa" o "ricorda".
   - In Java, lo stato è definito dai **campi** (o attributi, variabili di istanza) di una classe.
   - *Esempio*: Per un oggetto `Automobile`, lo stato potrebbe includere `colore`, `velocità_attuale`, `livello_carburante`.
2. **Comportamento (Behavior)**:
   - Rappresenta le azioni che l'oggetto può eseguire o le operazioni che possono essere eseguite su di esso.
   - È ciò che l'oggetto "fa".
   - In Java, il comportamento è definito dai **metodi** di una classe.
   - *Esempio*: Per un oggetto `Automobile`, il comportamento potrebbe includere `accelera()`, `frena()`, `cambiaMarcia()`.
3. **Identità (Identity)**:
   - Ogni oggetto è un'entità distinta e unica, anche se due oggetti hanno lo stesso stato.
   - L'identità permette di distinguere un oggetto da un altro.
   - In Java, l'identità è intrinseca all'oggetto stesso e non può essere modificata.
   - *Esempio*: Due oggetti `Automobile` possono essere entrambi rossi e viaggiare a 50 km/h, ma sono comunque due automobili distinte.

Il concetto di **incapsulamento** è strettamente legato a questi tre aspetti:

- L'incapsulamento significa raggruppare lo stato (dati) e il comportamento (metodi) che operano su quello stato all'interno di una singola unità (l'oggetto/classe) e **nascondere i dettagli interni** di implementazione
- Questo limita il più possibile le dipendenze con chi usa la classe, riducendo l'impatto delle modifiche (da `07-encapsulation_slides.pdf`, Pag. 5).

2.2.2 Interazione attraverso "Scambio di Messaggi"

Gli oggetti in un sistema software interagiscono tra loro non accedendo direttamente ai dati interni degli altri oggetti, ma attraverso lo **"scambio di messaggi"** (da `01-oo-abstraction_slides.pdf`, Pag. 3). Questo significa che un oggetto invoca un metodo su un altro oggetto.

- Quando un oggetto `A` chiama un metodo su un oggetto `B`, si dice che `A` sta inviando un "messaggio" a `B`.
- L'incapsulamento garantisce che l'oggetto `B` decida come rispondere a quel messaggio, senza che `A` debba conoscere i suoi dettagli interni.

2.2.3 Classe: Tipo di Oggetti e "Template di Costruzione"

Una **classe** è un concetto fondamentale in OOP. Non è un oggetto in sé, ma funge da (da `01-oo-abstraction_slides.pdf`, Pag. 3):

- **Tipo di Oggetti**: Definisce la struttura (campi) e il comportamento (metodi) comuni a tutti gli oggetti che ne sono istanze.
- **"Template di Costruzione" (Blueprint)**: È una sorta di progetto o stampino da cui vengono creati gli oggetti. Un oggetto è un'**istanza** di una classe.

Esempio: La classe `Cane` definisce che ogni cane ha un `nome` (stato) e può `abbaiare()` (comportamento). Quando creiamo `new Cane("Fido")`, stiamo creando un'istanza (un oggetto) della classe `Cane`.

2.2.4 Interfaccia vs. Implementazione (Information Hiding)

La distinzione tra interfaccia e implementazione è un altro aspetto chiave dell'incapsulamento e dell'astrazione (da `01-oo-abstraction_slides.pdf`, Pag. 3).

- **Interfaccia (Interface)**:
  - Definisce *cosa* un oggetto può fare (i suoi metodi pubblici), senza specificare *come* lo fa.
  - È il "contratto" che un oggetto offre al mondo esterno.
  - Permette agli altri oggetti di interagire con esso senza conoscere i dettagli interni.
  - In Java, le interfacce sono un costrutto specifico (che vedremo in dettaglio nel Capitolo 6).
- **Implementazione (Implementation)**:
  - Definisce *come* un oggetto esegue le azioni specificate dalla sua interfaccia.
  - Riguarda i dettagli interni, i campi privati e la logica dei metodi.
  - Il principio di **Information Hiding** (occultamento delle informazioni) suggerisce di nascondere i dettagli di implementazione all'esterno della classe, esponendo solo l'interfaccia necessaria. Questo riduce la complessità e facilita le modifiche future.

2.2.5 Riuso mediante Composizione ed Ereditarietà

La OOP promuove il riuso del codice attraverso due meccanismi principali

1. **Composizione (Composition)**:
   - Un oggetto "contiene" o "ha un" altro oggetto.
   - Si realizza quando una classe include istanze di altre classi come suoi campi.
   - *Esempio*: Una classe `Automobile` potrebbe contenere un oggetto `Motore` e quattro oggetti `Ruota`. L'automobile **è "composta" da** questi elementi.
2. **Ereditarietà (Inheritance)**:
   - Una classe "è un tipo di" un'altra classe.

- Permette a una nuova classe (sottoclasse o classe derivata) di riutilizzare (ereditare) campi e metodi da una classe esistente (superclasse o classe base).
- *Esempio*: Una classe `Cane` potrebbe ereditare da una classe `Animale`. Il `Cane` **"è un tipo di"** `Animale` e quindi eredita le sue proprietà e comportamenti generici.

Questi meccanismi saranno esplorati in dettaglio nei capitoli successivi.

2.3 Elementi Base dei Tipi di Java

Java è un linguaggio fortemente tipizzato, il che significa che ogni variabile deve avere un tipo dichiarato. In Java, i tipi si dividono in due categorie principali: tipi primitivi e tipi oggetto.

2.3.1 "Everything is an object" e Riferimenti ad Oggetti

La filosofia di Java è spesso riassunta con la frase **"Everything is an object"** (da `02-objects_slides.pdf`, Pag. 5). Sebbene ci siano delle eccezioni (i tipi primitivi), la maggior parte degli elementi con cui si interagisce in Java sono oggetti.

A differenza di linguaggi come C++ che supportano l'accesso ai dati per valore o tramite puntatori espliciti, in Java **le variabili conterranno dei riferimenti agli oggetti veri e propri** (da `02-objects_slides.pdf`, Pag. 5). Queste variabili sono quindi dei nomi "locali" utilizzabili per denotare l'oggetto.

- Quando si crea un oggetto in Java, si usa l'operatore `new`. Questo alloca memoria per l'oggetto e restituisce un riferimento ad esso.
- *Esempio*: `String s = new String();` (da `02-objects_slides.pdf`, Pag. 5). Qui, `s` non è l'oggetto String stesso, ma un riferimento che "punta" all'oggetto String creato in memoria.

2.3.2 Tipi Primitivi vs. Tipi Oggetto (Classi)

Java distingue chiaramente tra **tipi primitivi** e **tipi oggetto** (o tipi classe) (da `02-objects_slides.pdf`, Pag. 4).

Tabella 2.1: Tipi Primitivi di Java

Tipo Primitivo	Descrizione	Dimensione (bit)	Esempio di Valore
`byte`	Numero intero con segno	8	120
`short`	Numero intero con segno	16	30000
`int`	Numero intero con segno (più comune)	32	1000000
`long`	Numero intero con segno (grande)	64	9000000000L
`float`	Numero in virgola mobile a precisione singola	32	3.14f
`double`	Numero in virgola mobile a precisione doppia	64	3.14159
`boolean`	Valore booleano (vero/falso)	1	`true`
`char`	Carattere Unicode	16	'A'

(Informazioni basate su `02-objects_slides.pdf`, Pag. 6-7 e conoscenza generale di Java)

Tipi Oggetto (Classi):

- Rappresentano entità più complesse.
- Sono istanze di classi definite dall'utente o fornite dalle librerie Java (es. `String`, `Scanner`, `ArrayList`).
- Le variabili di tipo oggetto contengono **riferimenti** agli oggetti in memoria, non i valori degli oggetti stessi.
- L'allocazione della memoria per gli oggetti avviene tramite l'operatore `new`.

Tabella 2.2: Differenze tra Tipi Primitivi e Tipi Oggetto

Caratteristica	Tipi Primitivi	Tipi Oggetto (Classi)
Memorizzazione	Valore diretto nella variabile	Riferimento all'oggetto in memoria (heap)
Default	0, false, '\u0000' (dipende dal tipo)	`null` (nessun oggetto a cui si riferisce)
Creazione	Dichiarazione e assegnazione diretta	Uso di `new` per istanziare l'oggetto
Operazioni	Operatori aritmetici, logici, ecc.	Invocazione di metodi sull'oggetto
Null	Non possono essere `null`	Possono essere `null`
Esempio	`int x = 10;`	`String s = new String("Hello");`

2.4 Classi, Metodi e Campi in Java

Le **classi** sono il fondamento della programmazione orientata agli oggetti in Java. Esse **definiscono** la **struttura** e il **comportamento** degli oggetti.

2.4.1 Definizione di una Classe

Una classe in Java è definita utilizzando la parola chiave `class` seguita dal nome della classe. Al suo interno, si definiscono i **campi** (variabili di istanza) che rappresentano lo **stato** dell'oggetto e i **metodi** che rappresentano il suo **comportamento.**

- **Convenzioni di Naming**: I nomi delle classi in Java seguono la convenzione **PascalCase** (o UpperCamelCase), dove ogni parola inizia con una lettera maiuscola (es. `MyClass`, `Point3D`).

2.4.2 Campi (Variabili di Istanza)

I **campi** (o attributi o variabili di istanza) sono variabili dichiarate all'interno di una classe, ma al di fuori di qualsiasi metodo. Essi definiscono lo stato di un oggetto. Ogni istanza di una classe avrà la propria copia di questi campi (da `02-objects_slides.pdf`, Pag. 13).

- **Dichiarazione**: `tipo nomeCampo;` o `tipo nomeCampo = valoreIniziale;`
- **Accesso**: All'interno della classe, si accede ai campi direttamente. Dall'esterno, si accede tramite un riferimento all'oggetto e l'operatore punto (`.`) (es. `oggetto.nomeCampo`).

Esempio (da `02-objects_slides.pdf`, Pag. 13):

```
class Point {
    double x; // Campo che rappresenta la coordinata x
    double y; // Campo che rappresenta la coordinata y
}
```

2.4.3 Metodi

I **metodi** sono blocchi di codice che definiscono il comportamento di un oggetto. Essi operano sui dati dell'oggetto (i suoi campi) o eseguono altre operazioni. I metodi sono il mezzo principale attraverso cui gli oggetti interagiscono tra loro.

- **Dichiarazione**: `modificatore_accesso tipo_ritorno nomeMetodo(parametri) { // corpo del metodo }`
- **Convenzioni di Naming**: I nomi dei metodi in Java seguono la convenzione **camelCase**, dove la prima parola inizia con una lettera minuscola e le parole successive con una maiuscola (es. `myMethod`, `calculateSum`).
- **Parola chiave `this`**: All'interno di un metodo, la parola chiave `this` si riferisce all'istanza corrente dell'oggetto. È spesso usata per disambiguare tra un campo di istanza e un parametro di un metodo con lo stesso nome (da `02-objects_slides.pdf`, Pag. 22, 34).

Esempio (da `02-objects_slides.pdf`, Pag. 13):

```
class Point {
    double x;
    double y;

    // Metodo per inizializzare le coordinate del punto
    void build(double x, double y) {
        this.x = x; // 'this.x' si riferisce al campo della classe
        this.y = y; // 'y' si riferisce al parametro del metodo
    }

    // Metodo per stampare le coordinate del punto
    void print() {
        System.out.println("(" + this.x + ", " + this.y + ")");
    }

    // Metodo per calcolare la distanza dall'origine al quadrato
    double getNormSquared() {
        return this.x * this.x + this.y * this.y;
```

```
    }

    // Metodo per confrontare due oggetti Point
    boolean equal(Point other) {
        return this.x == other.x && this.y == other.y;
    }
}
```

2.4.4 Creazione e Uso di Oggetti (Istanze di Classi)

Per utilizzare una classe, è necessario crearne un'istanza (un oggetto) utilizzando l'operatore `new`. Una volta creato l'oggetto, è possibile accedere ai suoi campi (se pubblici) e invocare i suoi metodi (da `02-objects_slides.pdf`, Pag. 14).

Esempio (da `02-objects_slides.pdf`, Pag. 14):

```
public class UsePoint {
    public static void main(String[] s) {
        Point p = new Point(); // Crea un nuovo oggetto Point
        p.build(10, 20);      // Inizializza lo stato dell'oggetto p
        p.print();          // Invoca il metodo print() sull'oggetto p
    }
}
```

Le classi possono essere compilate separatamente. Se per compilare `UsePoint.java` è necessaria la classe `Point`, il file `Point.java` deve essere compilato prima. In alternativa, si possono compilare entrambi i file insieme, fornendoli entrambi al compilatore (es. `javac UsePoint.java Point.java`).

2.5 Accenno a Package e Librerie

Java organizza le classi in **package** per prevenire conflitti di nomi e per raggruppare classi correlate.

- Un package è una **raccolta di classi e interfacce correlate**.
- La dichiarazione del package deve essere la prima istruzione non commentata in un file sorgente Java (es. `package mypackage;`).
- Per utilizzare una classe da un altro package, è necessario importarla usando la parola chiave `import` (es. `import java.util.Scanner;`). Se non si importa, è necessario usare il nome completo della classe (es. `java.util.Scanner`).
- Il package `java.lang` è importato implicitamente in ogni programma Java e contiene classi fondamentali come `String` e `System`.

Le **librerie** Java sono collezioni di **classi e package predefiniti** che offrono funzionalità comuni e potenti, evitando agli sviluppatori di dover riscrivere codice da zero. Esempi includono:

- `java.util`: Contiene classi per collezioni (liste, mappe), utilità di data/ora, scanner per input, ecc.
- `java.io`: Classi per operazioni di input/output.
- `java.net`: Classi per la programmazione di rete.

2.6 Stampe a Video e Primo Semplice Programma Java

Per visualizzare output sulla console in Java, si utilizza il metodo `println()` della classe `System.out`.

- `System`: È una classe del package `java.lang` (quindi non richiede import).
- `out`: È un campo statico della classe `System`, di tipo `PrintStream`.
- `println()`: È un metodo del `PrintStream` che stampa una riga di testo e poi va a capo. Esistono anche `print()` (stampa senza andare a capo) e `printf()` (per stampe formattate).

Esempio (da `02-objects_slides.pdf`, Pag. 29):

```
System.out.println("Hello World!"); // Stampa "Hello World!" e va a capo
System.out.print("Questo è un ");   // Stampa "Questo è un "
System.out.println("test.");       // Stampa "test." e va a capo
```

Output:

```
Hello World!
Questo è un test.
```

2.6.1 Struttura di un Programma Java Eseguibile

Ogni programma Java che deve essere eseguito autonomamente deve avere un metodo `main`. Questo metodo è il punto di ingresso del programma (da `02-objects_slides.pdf`, Pag. 30).

- `public`: Il metodo è accessibile da qualsiasi altra classe.
- `static`: Il metodo appartiene alla classe stessa, non a un'istanza specifica dell'oggetto. Questo significa che può essere chiamato senza creare un oggetto della classe.
- `void`: Il metodo non restituisce alcun valore.
- `main`: Il nome del metodo, riconosciuto dalla JVM come punto di partenza.
- `String[] args`: Un array di stringhe che permette di passare argomenti dalla riga di comando al programma.

Esempio di programma Java minimo (da `02-objects_slides.pdf`, Pag. 30):

```
public class MyFirstProgram {
    public static void main(String[] args) {
        System.out.println("Il mio primo programma Java!");
    }
}
```