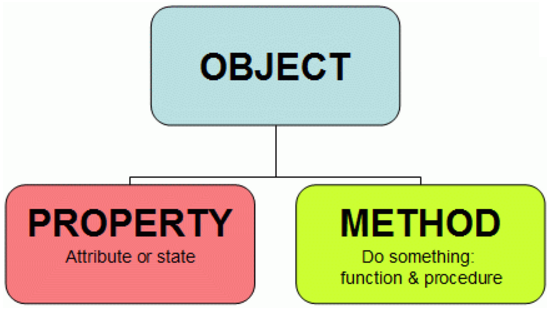
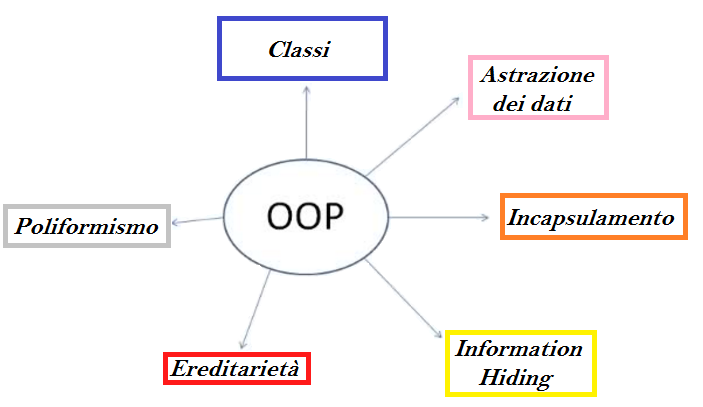
**Java** è un linguaggio di programmazione orientato agli oggetti

Consente lo sviluppo di applicazioni su piattaforme multiple e in reti eterogenee e distribuite

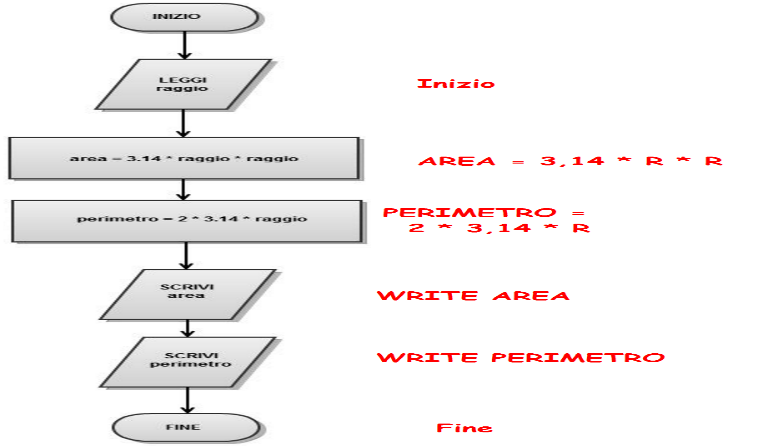
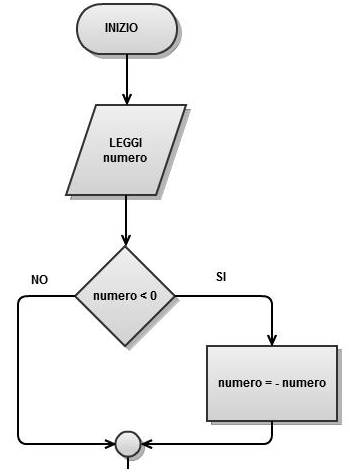
È stato creato per:

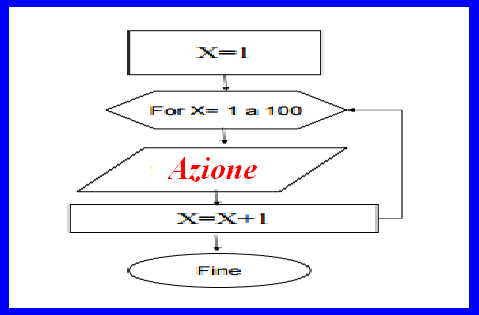
1. Essere orientato agli oggetti
2. Essere indipendente dalla piattaforma, robusto e sicuro
3. Contenere strumenti e librerie per il Networking
4. Essere progettato per eseguire codice da sorgenti remoti in modo sicuro



* Permette una più facile gestione e manutenzione di progetti di grandi dimensioni

STRUTTURE ALGORITMI:

 struttura di sequenza; struttura selettiva:

struttura iterativa:

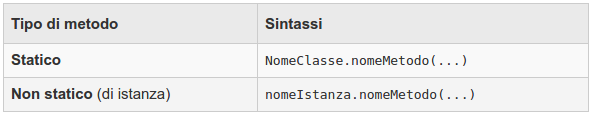
VARIABILE: Una variabile rappresenta una locazione di memoria all’interno della quale viene salvato un dato di un determinato **tipo**.

Locali: Le variabili locali sono delle variabili la cui dichiarazione avviene all’interno di un metodo.

Istanza: Le variabili di istanza sono delle variabili la cui dichiarazione avviene all’interno del corpo di una classe ma all’esterno dei metodi.

Di Classe: Le variabili di classe sono delle variabili la cui dichiarazione avviene all’interno del corpo di una classe attraverso la keyword **static**

METODO: Un metodo non è altro che un blocco di codice che svolge un determinato task. Rappresenta un sottoprogramma che può essere richiamato o riutilizzato in altre parti del programma evitandoci di dover riscrivere l’intero insieme di istruzioni.



È dunque necessario un metodo iniziale per far partire l’esecuzione del programma. Si utilizza metodo main.(public static void main(String[] args) {})

ARRAY

Un **array** in Java è un **contenitore** che consente di gestire una sequenza di lunghezza fissa di elementi dello stesso tipo. nome = new Tipo[n]; oppure int[] arrayInteri = {1, 2, 3, 4, 5, 6, 7, 8};

CLASSI WRAPPER

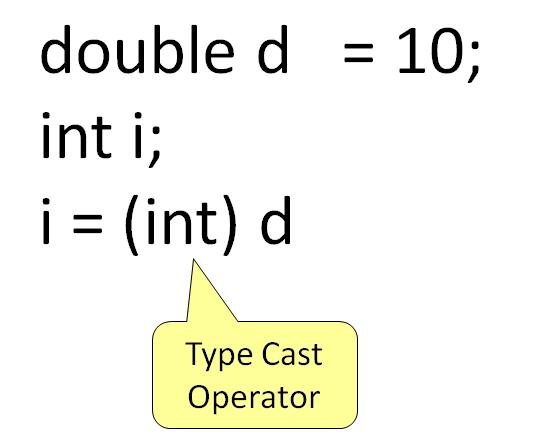
Una classe wrapper incapsula una variabile di tipo primitivo e la “trasforma” in un oggetto equivalente dandoci la possibilità di trattarlo come tale.

LIBRERIE

Java possiede un vasto insieme di librerie di classi standard, che costituiscono uno dei punti di forza del linguaggio. Queste sono organizzate in vari package (letteralmente pacchetti, fisicamente cartelle) che raccolgono le classi a seconda delle funzionalità che mettono a disposizione. Dobbiamo prima importarle però.

CAST

Il cast è l’operazione che consente di **trasformare** una variabile da un tipo di dato ad un altro.



VISIBILITÀ

I modificatori di visibilità regolano la possibilità di accedere ad una Classe, ad un metodo o ad un attributo, da parte di una Classe o di un metodo, esterno od interno alla Classe stessa

* Public → visibile da qualsiasi parte del programma
* Private → visibile solo dall'interno della classe stessa
* Protected → visibile solo dalle classi dello stesso package e dalle sottoclassi (classi derivate)
* Default → visibile dallo stesso package e dalle sottoclassi se sono nello stesso package. È il modificatore assegnato di default se non ne viene specificato nessuno dei precedenti.

OOP

La programmazione orientata agli oggetti è un paradigma di programmazione che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi.

È fondamentale approfondire i concetti di **classi**, **oggetti** e **costruttori** e i **principi** sui quali tale tipo di programmazione si basa.

**CLASSE**

*“Una classe è esattamente il “blueprint” (il prototipo) di un oggetto, in cui vengono definite tutte le proprietà che l’oggetto possederà.”*

Dichiarando le variabili di istanza (noti anche come attributi o fields) di una classe andiamo a modellare la forma degli oggetti che andremo a creare, definendone quindi lo stato.

Implementando i metodi di istanza, invece, andiamo a descrivere il comportamento degli oggetti definendo le operazioni che tali entità potranno eseguire o che potranno essere eseguite su di esse.

**OGGETTI**

In Java per “oggetto” si intende l’istanza particolare di una determinata classe.

* Per poter istanziare un nuovo oggetto della relativa classe, ci avvaliamo del suo costruttore.
* Il valore delle variabili di istanza è specifico per la singola istanza (ogni oggetto rappresenta una specifica copia del suo prototipo).
* Tutte le istanze di una determinata classe hanno gli stessi metodi.
* La relazione tra classi e oggetti è tale che una classe definisce il tipo (non primitivo) i cui valori sono degli oggetti.

**COSTRUTTORE**

Il **costruttore** è quel metodo di una classe il cui compito è quello di creare nuove istanze.

Esempio Persona persona = new Persona();

**I PRINCIPI**

i tre principi fondamentali della programmazione ad oggetti sono **ereditarietà**, **polimorfismo** e **incapsulamento**.

**L'incapsulamento** rappresenta il principio in base al quale una classe può mascherare la sua struttura interna e proibire ad altri oggetti di accedere ai suoi dati

Non abbiamo modo di accedere in maniera diretta allo stato interno di un oggetto e l’unica possibilità di ottenere o di modificare i valori dei suoi attributi (variabili di istanza) è rappresentata da un insieme di elementi public quali i cosiddetti **getters and setters**. this ci riferiamo alla particolare istanza che richiama il metodo.

Il principio dell’**ereditarietà** si basa sul fatto di poter definire un legame di dipendenza di tipo gerarchico tra classi diverse.

Una classe deriva da un'altra se da essa ne **eredita** stato e comportamento.

La classe "figlia" si dice classe **derivata**, mentre la classe "padre" prende il nome di **superclasse**. public class Studente extends Persona { … }

Tramite la keyword **super** possiamo riferirci alla superclasse per richiamare un costruttore

Il polimorfismo rappresenta il principio in funzione del quale diverse classi derivate possono implementare uno stesso comportamento definito nella classe base in modo differente.

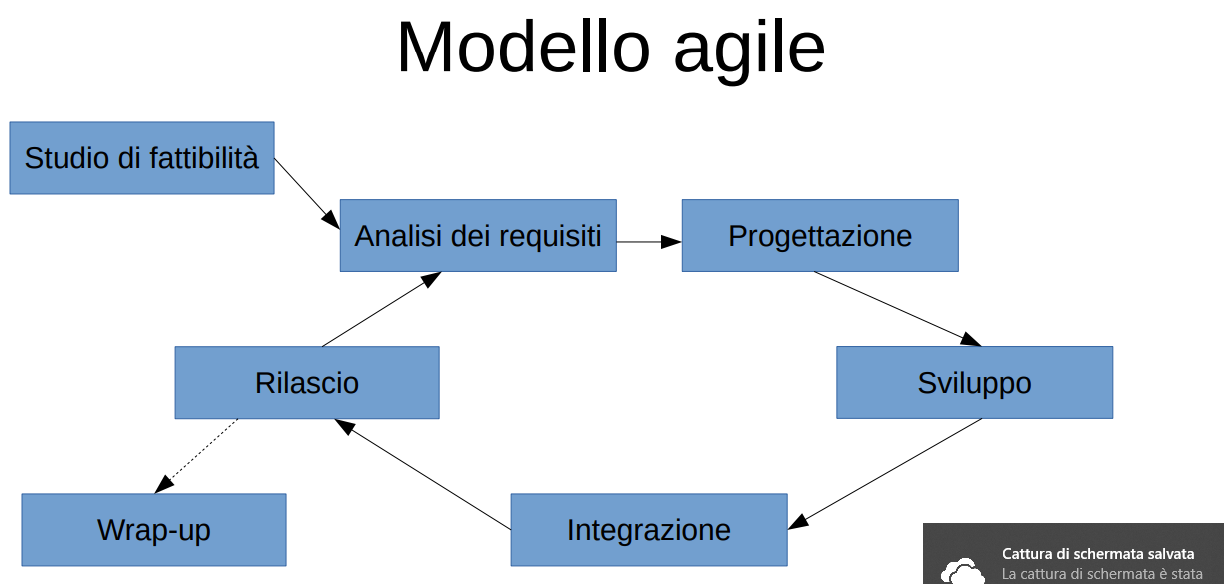
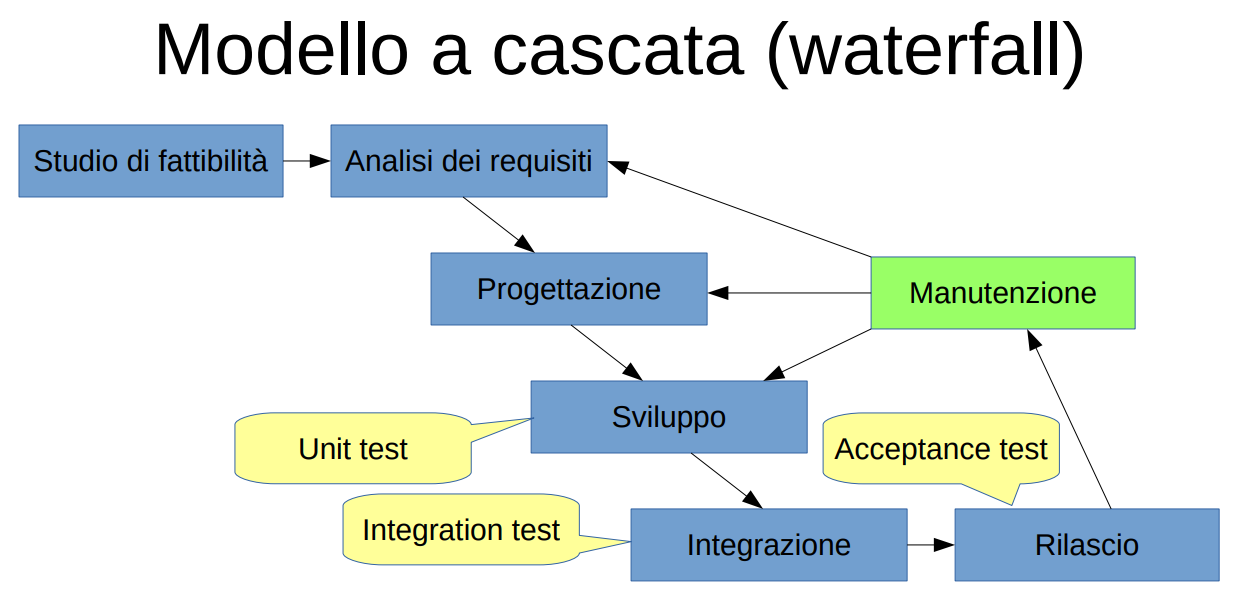
Dal punto di vista implementativo il polimorfismo per i metodi si ottiene utilizzando l'overload e l'override dei metodi stessi.

**SOFTWARE ENGINEERING**

Con **software** **engineering** (ingegneria del software) ci riferiamo a quella disciplina che si occupa dei processi produttivi e delle metodologie di sviluppo finalizzate alla realizzazione di sistemi software.

CICLO VITA:

* Studio di fattibilità
* Analisi dei requisiti
* Proggettazione dell’architettura
* Programmazione e debugging
* Testing
* Rilascio
* Manutenzione



**Codifica e Debugging**

La **codifica** è il processo con cui si traducono i progetti e i documenti dettagliati dei moduli descritti precedentemente in linguaggio di programmazione.

Il **debugging** è il processo in cui si individuano e correggono gli

errori di programma. Possiamo riscontrare diversi tipi di errori:

* **Errori di sintassi**: un’istruzione non segue le corrette regole della sintassi, il che la rende irriconoscibile dal compilatore (segnalati dall’IDE).
* **Errori di runtime**: si verificano soltanto durante l’esecuzione del programma con particolari insiemi di dati che danno origine a operazioni non lecite, come ad esempio la divisione per zero.
* **Errori logici**: i più difficili da rintracciare ed eliminare, causati da scelte fatte

nell’algoritmo usato per risolvere il problema.

In questa fase vengono definiti una serie di casi di prova e vengono eseguiti con specifici dati. I dati di prova, se scelti con cura per percorrere tutti i diversi percorsi logici all’interno di un programma, possono aiutare a scoprire errori.

Vengono, dunque effettuati tutti i test necessari a garantire il corretto funzionamento di ogni funzionalità presente all’interno del software.

Il software dopo aver superato la fase di **testing (JUNIT)** viene quindi **rilasciato** e quindi **manutenuto**.

**ECCEZIONE**

Le eccezioni sono un modo chiaro e strutturato per **controllare** gli errori. Quando si verifica una situazione di errore 🡪 viene lanciata una eccezione (*java.lang.Exception*), che se viene in seguito catturata permette di gestire l’errore ed evitare che il programma stesso termini.

In Java le eccezioni sono degli oggetti che estendono la classe **Exception** che a sua volte estende **Throwable**.

Per **creare** una eccezione personalizzata è sufficiente creare una classe che estenda *Exception*.

Le eccezioni che creiamo estendendo Exception possono essere sollevate all’interno di un metodo tramite l’istruzione **throw**.

Il metodo che, invece, potrebbe sollevare un’eccezione durante la sua esecuzione deve segnalare quest’eventualità tramite la clausola **throws**. Tramite questo operatore possiamo rimandare la gestione dell’eccezione direttamente al metodo chiamante.

Nota: L’eccezione lanciata da noi si dice eccezione **controllata**, questa deve essere obbligatoriamente gestita, mentre quelle lanciate da Java non per forza devono essere catturate e gestite.

Abbiamo visto, quindi, che le eccezioni vengono lanciate quando viene invocato un metodo dichiarato con la clausola **throws** o che contiene un’istruzione **throw**.

Per gestire queste eccezioni abbiamo bisogno di un costrutto chiamato **blocco try/catch**.

All’interno del **try** andiamo ad includere le righe di codice che potrebbero dar vita ad un’eccezione.

Nel **catch**, invece, catturiamo l’eccezione e indichiamo le istruzioni che verranno eseguite per gestirla.

Possiamo infine aggiungere al blocco la clausola **finally** che conterrà delle istruzioni che verranno eseguite sia in caso di normale esecuzione del programma sia in caso di eccezione.

**CLASSI ASTRATTE**

Le classi astratte (abstract class) sono delle che contengono del codice comune a più classi che ne deriveranno. Non possono essere istanziate quindi può essere utilizzata esclusivamente come classe base. Lo scopo e l’utilità delle classi astratte è di **gestire il comportamento** di base **delle classi che la derivano**

public abstract class Persona {

public abstract void stampaNome();

}

**INTERFACCE**

Un’interfaccia rappresenta una sorta di “promessa” che una classe si impegna a mantenere. La promessa è quella di **implementare** determinati metodi di cui viene resa nota soltanto la definizione.

Le interfacce non possono essere istanziate, non contengono ne costruttori ne variabili di istanza (solo public static).

Allo stesso modo delle classi, è possibile utilizzare l’ereditarietà anche per le interfacce.

Infine, **una classe può implementare più di una interfaccia**.

CONFRONTO

Si usa una **classe astratta** per condividere codice in comune tra più classi : se più classi hanno in comune sia lo stato che il comportamento oppure se si vogliono dichiarare variabili comuni che non siano necessariamente campi static e final.

Si decide di utilizzare un’**interfaccia** se ci si trova nella situazione in cui alcune classi (anche se strutturate in gerarchie differenti) si trovano a condividere dei metodi, se si vuole specificare il comportamento di un certo tipo di dato (ma non implementarne il comportamento) o se si vuole avere la possibilità di sfruttare la “***multiple inheritance***”, quindi ereditare da più di un'interfaccia.