MODELLAZIONE E GESTIONE DELLA CONOSCENZA

Docenti

Michele Loreti

Lorenzo Rossi

Studente

Giorgio Della Roscia

Università degli studi di Camerino

Informatica per la Comunicazione Digitale (L-31)

Scuola di Scienze e Tecnologie

A.A. 2023/2024

Indice

1 OOP **1**
1.1 Classi . . . 1
1.1.1 Oggetti . . . 1
1.1.1.1 Oggetti immutabili . . . 2
1.1.1.2 Campi . . . 2
1.1.1.3 Metodi . . . 2
1.1.1.3.1 Costruttori . . . 3
1.1.1.3.2 Contratti . . . 3
1.2 Interfacce . . . 3
1.2.1 Metodi delle interfacce . . . 4
1.2.1.1 Polimorfismo . . . 4
1.3 Ereditarietà . . . 4
1.3.1 Overloading . . . 5
1.3.2 Overriding . . . 6
1.3.3 Casting . . . 6
1.3.4 Classi astratte . . . 7
1.4 Programmazione generica . . . 7
1.4.1 Classi generiche . . . 8
1.4.1.1 Metodi generici . . . 9
1.4.2 Wildcards . . . 9
1.5 UML . . . 10
1.6 Interfacce funzionali . . . 11
1.6.1 Lambda expressions . . . 11
1.7 Programmazione concorrente . . . 11

2 XML **13**
2.1 Struttura XML . . . 14
2.1.1 Preambolo XML . . . 14
2.1.2 Attributi e sottoelementi . . . 14
2.1.2.1 Caratteri speciali . . . 15
2.1.3 Namespace . . . 15
2.1.4 CDATA . . . 16
2.2 DTD . . . 16
2.2.1 Elementi . . . 17
2.2.2 Attributi . . . 18
2.2.3 Entità . . . 19
2.3 XSD . . . 19
2.3.1 Elementi XSD semplici . . . 21
2.3.2 Elementi XSD complessi . . . 21
2.4 SAX . . . 22
2.4.1 DOM . . . 23
2.5 XPath . . . 23
2.5.1 Funzioni XPath . . . 25
2.5.2 REGEX . . . 25
2.6 XSLT . . . 26

3 ONTOLOGIE **28**
3.1 Semantic web . . . 29
3.1.1 Significato e comprensione . . . 30
3.1.1.1 Tassonomia e partonomia . . . 30
3.2 RDF . . . 31
3.2.1 Turtle . . . 32
3.2.2 Reificazione . . . 32
3.2.3 RDFS . . . 33
3.3 Model-theoretic semantics . . . 33
3.3.1 Interpretazione . . . 34
3.3.2 Regole di deduzione . . . 35
3.3.2.1 Mondo aperto . . . 35
3.4 OWL . . . 36
3.4.1 Sintassi . . . 36
3.4.2 Costrutti . . . 36
3.4.2.1 Costrutti base . . . 36
3.4.2.1.1 Classi . . . 36
3.4.2.1.2 Proprietà . . . 37
3.4.2.1.3 Individui . . . 37
3.4.2.1.4 Gerarchie . . . 37
3.4.2.2 Costruttori logici . . . 38
3.4.2.3 Vincoli di proprietà . . . 38
3.4.2.4 Relazioni tra proprietà . . . 40
3.5 SPARQL . . . 40
3.5.1 Modificatori . . . 41
3.5.1.1 Funzioni di aggregazione . . . 41
3.5.2 Sub query . . . 41
3.5.3 Path di proprietà . . . 42
3.5.4 SPARQL protocol . . . 42
3.5.5 Query SPARQL . . . 42
3.5.5.1 Funzioni di aggiornamento . . . 42
3.6 Linked Data Mashup . . . 43

4 JAVA **45**
4.1 JavaDoc . . . 45
4.2 Principi SOLID . . . 46
4.3 Code smells . . . 46
4.4 Testing . . . 47
4.4.1 JUnit . . . 47
4.5 Apache Jena . . . 48
4.5.1 RDF Model . . . 48

Sitografia

Bibliografia

Figure

1.1 Simboli sintattici relazioni fra classi 10

2.1 URI 15
2.2 Elementi semplici e complessi 19
2.3 Grafica degli assi di XPath 23
2.4 Modello XLST 26

3.1 Istanza di un'ontologia 29
3.2 Tecnologie del web 29
3.3 Triangolo semiotico 30
3.4 Rapporto ontologia-reasoner-conoscenza 33
3.5 Interpretazioni di preposizioni 34
3.6 Assenza di cardinalità dei vincoli relazionali 35

Tabelle

1.1 Modificatori ai campi di una classe 2
1.2 Livelli di accesso dei modificatori 2
1.3 Notazione di visibilità UML 10

2.1 Attributi del tag di apertura di un file XML 14
2.2 Caratteri speciali XML 15
2.3 Ripetizione degli elementi contenuti 17
2.4 Tipologia attributi degli elementi 18
2.5 Modificatori dei valori degli attributi 18
2.6 Restrizioni imposte agli attributi 21
2.7 Approcci al parsing 22
2.8 Assi di XPath 23
2.9 Sintassi abbreviata degli assi 24
2.10 Operatori dei filtri 24
2.11 Cheatsheet dei metacaratteri 25

3.1 Evoluzione del web 29
3.2 Componenti della semantica RDFS 33
3.3 Differenze gradi OWL 36
3.4 Gerarchie e unicità 38
3.5 Restrizioni e vincoli di proprietà 38
3.6 Relazioni tra proprietà 40
3.7 Operatori SPARQL 40
3.8 Modificatori di risultato delle query SPARQL 41
3.9 Funzioni di aggregazione 41
3.10 Path di proprietà 42

Codici

1.1 Sintassi di una classe . . . 1
1.2 Oggetti immutabili tramite record . . . 2
1.3 Metodo costruttore . . . 3
1.4 Metodi con contratto presenti nella classe `Object` . . . 3
1.5 Classe che estende un'interfaccia . . . 3
1.6 Implementazione di un factory method . . . 4
1.7 Polimorfismo di un metodo . . . 4
1.8 Dichiarazione di una sottoclasse . . . 4
1.9 Costruttore della superclasse nella sottoclasse . . . 5
1.10 Dynamic lookup classi padre e figlia . . . 6
1.11 Dynamic lookup dichiarazione oggetto . . . 6
1.12 Widening . . . 6
1.13 Narrowing . . . 6
1.14 Classe astratta . . . 7
1.15 Interfaccia con implementazione controllata . . . 8
1.16 Classe generica per memorizzare un dizionario . . . 8
1.17 Type erasure . . . 8
1.18 Scambio di elementi in un vettore . . . 9
1.19 Type bounding multiplo . . . 9
1.20 Unbounded wildcards . . . 9
1.21 Subtype wildcards . . . 9
1.22 Supertype wildcards . . . 10
1.23 Espressione lambda . . . 11
2.1 Documento XML errato . . . 13
2.2 Codice XML riportabile come albero . . . 14
2.3 Tag di apertura di un file XML . . . 14
2.4 Attributi XML . . . 14
2.5 Sottoelementi XML . . . 15
2.6 XML CDATA . . . 16
2.7 Documento XML ben indentato . . . 16
2.8 Dichiarazione del DTD interna . . . 16
2.9 Dichiarazione del DTD esterna . . . 16
2.10 Dichiarazione del DTD mista . . . 16
2.11 Documento XML da cui scrivere il DTD . . . 17
2.12 Documento DTD scritto dall'XML . . . 17
2.13 Documento DTD da cui scrivere l'XML . . . 18
2.14 Documento XML scritto dal DTD . . . 18
2.15 Dichiarazione DTD degli attributi di un elemento XML . . . 18
2.16 Dichiarazione di un'entità . . . 19
2.17 Utilizzo di un'entità . . . 19
2.18 Dichiarazione schema XSD . . . 19
2.19 Documento XSD costruito dall'XML . . . 20
2.20 Documento XSD costruito dall'XML . . . 20
2.21 Definizione elemento semplice . . . 21
2.22 Elemento complesso `xs:sequence` . . . 21

2.23 Elemento complesso `xs:all` . . . 22
2.24 Elemento complesso `xs:choice` . . . 22
2.25 Dichiarazione dello stile XLST in XML . . . 26
2.26 Value of in XLST . . . 26
2.27 Foreach in XLST . . . 26
2.28 Sort in XLST . . . 26
2.29 If in XLST . . . 26
2.30 Choose in XLST . . . 26
3.1 Compattazione da RDF a Turtle . . . 32
3.2 Compattazione da RDF a Turtle . . . 32
3.3 Header OWL . . . 36
3.4 Definizione di una object property . . . 37
3.5 Definizione di una datatype property . . . 37
3.6 Gerarchie OWL . . . 37
3.7 OWL disgiunzione fra classi . . . 37
3.8 OWL classi chiuse . . . 38
3.9 Costrutti logici OWL . . . 38
3.10 `hasValue` . . . 39
3.11 `allValuesFrom` . . . 39
3.12 `someValuesFrom` . . . 39
3.13 `cardinality` . . . 39
3.14 Sintassi SPARQL . . . 40
3.15 Ridenominazione tramite `AS` . . . 41
3.16 Sub query . . . 41
4.1 Commenti JavaDoc . . . 45
4.2 Metodo di test . . . 47
4.3 Modello RDF vuoto . . . 48
4.4 Modello RDF da file . . . 49
4.5 Modello RDF da altro modello . . . 49
4.6 Operazioni sui modelli RDF . . . 49

Approfondimenti

Definizione (classe) . . . 1
Definizione (package) . . . 1
Definizione (oggetto) . . . 1
Definizione (invariante di classe) . . . 1
Nota bene (modificabilità oggetto) . . . 2
Definizione (getters) . . . 2
Definizione (setters) . . . 2
Definizione (contratti dei metodi) . . . 3
Esempio (metodi classe Object) . . . 3
Definizione (programmazione difensiva) . . . 3
Definizione (buffer) . . . 5
Nota bene (classe figlia > classe padre) . . . 6
Esercizio (*dynamic lookup*) . . . 6
Definizione (tipo di dato) . . . 7
Definizione (sottotipo di dato) . . . 7
Nota bene (classi sealed) . . . 8
Definizione (classe generica) . . . 8
Esempio (classe generica) . . . 8
Esempio (metodo generico) . . . 9
Definizione (type checking) . . . 9
Definizione (type inference) . . . 9
Definizione (UML class diagram) . . . 10
Definizione (threading) . . . 11
Definizione (future) . . . 11

Esercizio (correzione XML) . . . 13
Esempio (albero XML) . . . 14
Nota bene (commenti pre-dichiarazione) . . . 15
Definizione (*Unified Resource Identifier*) . . . 15
Esercizio (XML da DTD) . . . 17
Esercizio (DTD da XML) . . . 18
Esempio (entità) . . . 19
Nota bene (entità con dichiarazione DTD interna) . . . 19
Esercizio (XSD da XML) . . . 20
Nota bene (datetime) . . . 21
Definizione (parser) . . . 22
Esempio (filtri) . . . 24
Esempio (operatori nei filtri) . . . 24

Definizione (ontologia) . . . 28
Nota bene (ontologia non assiomatizzata) . . . 28
Esempio (ontologia) . . . 28
Esempio (ambiguità della parola *"Polo"*) . . . 30
Definizione (informazione) . . . 30
Definizione (dato) . . . 30
Definizione (tassonomia) . . . 30
Definizione (partonomia) . . . 30

Esempio (RDF triple) . . . 31
Esempio (blank node) . . . 31
Definizione (tripla) . . . 32
Esercizio (Turtle da RDF) . . . 32
Esempio (reificazione) . . . 32
Nota bene (risorse RDFS) . . . 33
Nota bene (implicazione logica transitiva) . . . 33
Esempio (implicazione logica transitiva) . . . 33
Definizione (vocabolario) . . . 34
Nota bene (insiemi delle funzioni) . . . 35
Definizione (assioma) . . . 35
Esempio (assioma) . . . 35
Esempio (istanza non disgiunta) . . . 35
Esempio (gerarchie) . . . 37
Definizione (graph pattern) . . . 40
Esempio (graph pattern) . . . 40
Definizione (path di proprietà) . . . 42
Esempio (SPARQL protocol) . . . 42

Principio (Single responsibility) . . . 46
Esempio (Single responsability principle) . . . 46
Principio (Open-closed) . . . 46
Principio (Liskov substitution) . . . 46
Lemma (Liskov substitution) . . . 46
Definizione (proprietà) . . . 46
Principio (Interface segregation) . . . 46
Principio (Dependency inversion) . . . 46
Definizione (test) . . . 47
Definizione (state testing) . . . 47
Definizione (behavior testing) . . . 47
Definizione (test coverage) . . . 47
Definizione (JUnit Platform) . . . 48
Definizione (JUnit Jupiter) . . . 48
Definizione (JUnit Vintage) . . . 48

Sezione 1

OOP

Il codice nella OOP (*Object Oriented Programming*) deve seguire i seguenti principi:

- *the simpler, the better*;
- riscrittura se errato (refactoring);
- imparare dagli errori commessi in precedenza.

La programmazione imperativa, ad esecuzione sequenziale, organizza ogni programma in moduli ed è costituito da strutture dati e funzioni. Nella OOP si va a rimuovere tale spartizione: lo stesso oggetto memorizza l'informazione e, al contempo, ne fornisce i rispettivi metodi.

1.1 Classi

> **Definizione** (classe)**.** Prototipo per un insieme di oggetti.

Vengono indicati attributi e metodi che sono messi a disposizione. Un oggetto creato con una classe è detto istanza di essa.

```
1 public class NomeClasse { ... }
```

Codice 1.1: Sintassi di una classe

La convenzione rispetto al nome di una classe richiede:

- inglese
- sostantivi
- PascalCase

In Java, le classi fanno parte di un pacchetto.

> **Definizione** (package)**.** Meccanismo per organizzare classi Java in gruppi logici, principalmente allo scopo di definire namespace distinti per diversi contesti.

Un esempio sono le librerie standard organizzate in base alla loro finalità: `java.math`, `java.io`...

1.1.1 Oggetti

> **Definizione** (oggetto)**.** Componente software caratterizzata da
>
> * uno stato → attributi,
> * un comportamento → metodi.

L'uso dell'OOP pecca in verbosità, ma spicca per riusabilità, manutenibilità ed estendibilità.

> **Definizione** (invariante di classe)**.** Proprietà che lo stato di un'oggetto verifica per tutta la sua esistenza.

1.1.1.1 Oggetti immutabili

Lo stato non può cambiare, dunque i conflitti dovuti da iterazione con altri oggetti diminuisce.

> **Nota bene** (modificabilità oggetto)**.** Solamente l'oggetto stesso può modificare il suo stato! Questo per accertarsi che la struttura delle informazioni interne sia coerente con le assunzioni.

Il costrutto `record` permette di definire oggetti immutabili.

```
1 record Position(int row, int column){};
```

Codice 1.2: Oggetti immutabili tramite record

Si ha una definizione momentanea di una classe `final`. Vengono inoltre generati i getters per accedere ai campi.

1.1.1.2 Campi

Oltre al tipo, al nome, che usa sostantivi in camelCase, e all'inizializzazione si possono avere dei modificatori di visibilità e di accesso:

Tabella 1.1: Modificatori ai campi di una classe

VISIBILITÀ	ACCESSO
`public`	`static`
`protected`	`final`
`private`	

La regola generale è di usare solo campi `private` e `protected`.

Tabella 1.2: Livelli di accesso dei modificatori

MODIFIER	CLASS	PACKAGE	SUBCLASS	WORLD
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	–
no	✓	✓	–	–
`private`	✓	–	–	–

1.1.1.3 Metodi

Il nome, in camelCase, deve essere significativo in modo da esplicare alla lettura la funzione del metodo. Le tre operazioni standard sono:

1. ricezione di parametri;
2. cambiamento dello stato degli oggetti;
3. restituzione di un valore.

Per una buona pulizia del codice ogni metodo non dovrebbe superare le 10/15 righe.

> **Definizione** (getters)**.** Metodi che garantiscono l'accesso in lettura ai campi della classe.

> **Definizione** (setters)**.** Metodi che garantiscono l'accesso in scrittura ai campi della classe.

1.1.1.3.1 Costruttori

Metodo invocato per la creazione di un oggetto dove vengono inizializzati più elementi.

```java
class Persons {
    private String name;
    private String surname;
    public Persons(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }
}
```

Codice 1.3: Metodo costruttore

Se assente, Java ne crea automaticamente uno.

1.1.1.3.2 Contratti

Definizione (contratti dei metodi)**.** Garanzia rispetto a dei parametri soddisfacenti di certe condizioni preimposte.

I contratti vengono inseriti nel JavaDoc dato che non è permesso esprimerli nel codice.

Esempio (metodi classe Object)**.** Ecco tre metodi standard della classe `Object` che si aspettano un determinato tipo di dato.

```java
equals(Object obj) //bool
hashCode() //int
toString() //String
```

Codice 1.4: Metodi con contratto presenti nella classe `Object`

Chi va ad utilizzare gli oggetti si impegna a rispettarne il contratto.

Definizione (programmazione difensiva)**.** Costrutti sintattici atti alla verifica del funzionamento atteso del codice.

Tali costrutti sono: `if-then-else` ed `assert`. La violazione dei contratti comporta la restituzione di un codice di errore o il lancio di un'eccezione.

1.2 Interfacce

Si dispone un meccanismo per la definizione un contratto tra due parti:

a. l'erogatore di un servizio;

b. le classi che lo vanno ad utilizzare.

Vengono messi a disposizione degli ADT (*Abstract Data Type*): List, Map, Queue, Set, Stack, . . . È poi compito delle classi che estendono l'interfaccia implementarli.

```java
public interface Closeable {
    void close();
}

public interface Channel extends Closeable {
    boolean isOpen();
}
```

Codice 1.5: Classe che estende un'interfaccia

Una classe può implementare un qualsiasi numero di interfacce a condizione di sfruttare ogni metodo presente. Ogni campo definito è automaticamente posto come `public static final`.

1.2.1 Metodi delle interfacce

Nelle interfacce, i metodi implementati possono essere:

* *statici*, funzionalità generiche;
* *default*, compatibilità evolutiva;
* *privati*, funzionalità di utilità.

I *factory methods* costruiscono un oggetto senza la necessità di conoscere la classe, come accade invece per i costruttori.

```
1 public interface SolverStrategy {
2     ...
3     static SolverStrategy getSolver(int levels) {
4         return new BoundedSearchStrategy(levels);
5     }
6 }
```

Codice 1.6: Implementazione di un factory method

Per sfruttare i *factory methods* è però necessario conoscere i parametri.

1.2.1.1 Polimorfismo

L'uso delle interfacce è un'astrazione rispetto alla reale implementazione.

```
1  class Animal {
2      public void animalSound() {
3          System.out.println("the animal makes a sound");
4      }
5  }
6
7  class Dog extends Animal {
8      @Override
9      public void animalSound() {
10         System.out.println("the dog says: 'wof wof'");
11     }
12 }
13
14 class Main {
15     public static void main(String[] args) {
16         new Animal().animalSound();
17         new Dog().animalSound();
18     }
19 }
```

Codice 1.7: Polimorfismo di un metodo

Si vanno a ridefinire i metodi dell'interfaccia relativi alla situazione attuale.

1.3 Ereditarietà

Viene permessa la costruzioni di oggetti basandosi su uno standard già definito.

```
1 public class Dog extends Animal { ... }
```

Codice 1.8: Dichiarazione di una sottoclasse

Come visibile, la parola chiave da sfruttare è `extends`. Il costruttore della sottoclasse va a richiamare, immediatamente nella prima riga, quello del padre.

```
1  class Animal {
2      public Animal (String arg) {
3          System.out.println("Animale: " + arg);
4      }
5  }
6
7  class Dog extends Animal {
8      public Dog () {
9          super("cane");
10     }
11 }
```

Codice 1.9: Costruttore della superclasse nella sottoclasse

Definizione (buffer)**.** Memoria che immagazzina i dati ricevuti come array. Si hanno due indici che puntano rispettivamente:

a. primo dato disponibile da leggere;

b. prima posizione vuota da utilizzare.

La caratteristica principale è però quella di poter ridimensionare la dimensione.

Per quanto riguarda gli assegnamenti, i tipi delle variabili possono essere:

* *statico*, si fa riferimento alla dichiarazione;
* *dinamico*, tipo dell'oggetto a cui si fa riferimento.

1.3.1 Overloading

La *signature* di un metodo lo identifica univocamente all'interno della classe:

$$\text{signature} = \overbrace{\text{nome}}^{\text{metodo}} + \overbrace{\text{tipo} + \text{numero}}^{\text{attributi}}$$

Risulta possibile, e talvolta utile, definire più metodi con lo stesso nome, ma signature diversa.

1.3.2 Overriding

In questo caso il metodo della sottoclasse ha la medesima signature di quello presente nella superclasse, viene dunque sovrascritto.

Nota bene (classe figlia > classe padre)**.** La sottoclasse ha più funzionalità della superclasse!

Esercizio (*dynamic lookup*)**.** Date le classi

```
class ClassA {
    public void m1() {
        System.out.println("A1");
        m2();
    }
    public void m2() {
        System.out.println("A2");
    }
}

class ClassB extends ClassA {
    public void m2() {
        System.out.println("B2");
    }
}
```

Codice 1.10: Dynamic lookup classi padre e figlia

Cosa accade eseguendo il codice che segue?

```
ClassA a = new ClassB();
a.m1();
```

Codice 1.11: Dynamic lookup dichiarazione oggetto

.....

Il risultato ottenuto sarà una stampa a schermo di: *"A1"* e *"B2"*. Questo accade poiché l'oggetto `a` è stato dichiarato di tipo `ClassB`.

1.3.3 Casting

Si analizza ora la conversione dei dati da un tipo ad un altro. Esistono due possibili alternative per quanto riguarda il type casting:

* implicito (*widening*), avviene automaticamente quando il passaggio è da piccolo a grande;

```
1 int i = 10;
2 double d = i; //int automatically converts to double
```

Codice 1.12: Widening

L'ordine di successione di grandezza con cui vengono convertiti i dati è dato da

$$\text{byte} \rightarrow \text{short} \rightarrow \text{char} \rightarrow \text{int} \rightarrow \text{long} \rightarrow \text{float} \rightarrow \text{double}$$

* esplicito (*narrowing*), occorre specificarlo manualmente e può comportare perdita di informazioni.

```
1 double d = 10.5;
2 int i = (int) d; //double must be casted to int
```

Codice 1.13: Narrowing

L'ordine di successione di grandezza con cui vengono ridimensionati i dati è

$$\text{double} \rightarrow \text{float} \rightarrow \text{long} \rightarrow \text{int} \rightarrow \text{char} \rightarrow \text{short} \rightarrow \text{byte}$$

Il comando `instanceof` permette di verificare se un oggetto appartiene, o comunque è compatibile, ad una classe.

1.3.4 Classi astratte

Una classe può definire un metodo senza implementazione, forzando così le sottoclassi a fornirne una propria. Sia il metodo che la classe in cui è contenuto vengono detti `abstract`.

```
1  public abstract class Person {
2      private final String name;
3      public Person (String name) {
4          this.name = name;
5      }
6      public final String getName() {
7          return name;
8      }
9      public abstract int getId();
10 }
11 public class Student extends Person {
12     private final int id;
13     public Student (int id, String name) {
14         super(name);
15         this.id = id;
16     }
17     public final int getId() {
18         return id;
19     }
20 }
```

Codice 1.14: Classe astratta

Sia le classi astratte che le interfacce sono uno strumento di astrazione.

1.4 Programmazione generica

Viene garantita la possibilità di descrivere il comportamento di una struttura dati indipendentemente dal suo contenuto.

> **Definizione** (tipo di dato)**.** Insieme dei valori che una variabile, o il risultato di un'espressione, può assumere.

In Java esistono i tipi primitivi (int, double, char, boolean, ...) e quelli derivanti da classi ed interfacce. Un'oggetto è un'istanza di una classe se è stato costruito da un suo costruttore, o da quello di una sua sottoclasse.

> **Definizione** (sottotipo di dato)**.** Il tipo T_1 è sottotipo di T_2 $(T_1 < T_2)$ quando possiamo usare T_1 al posto di T_2 senza incorrere in errori di tipo.

Gli errori di tipo sono causati da:

- eccezione;
- perdita d'informazione;
- uso di un metodo non presente o inaccessibile;
- uso di un attributo non presente o inaccessibile.

Da Java 17, per limitare le classi che implementano una determinata interfaccia, o che estendono una classe, è possibile utilizzare il modificatore `sealed`.

```
1 public sealed interface InterfaceA permits ClassA, ClassB { ...
```

Codice 1.15: Interfaccia con implementazione controllata

In questo caso solo le classi *ClassA* e *ClassB* potranno implementare l'interfaccia *InterfaceA*.

Nota bene (classi sealed). Una classe `permitted` dovrà essere:

* `sealed`, con la lista delle classi `permits`;
* `non-sealed`, per consentire ulteriori estensioni;
* `final`, per impedire che altre classi la possano estendere.

1.4.1 Classi generiche

Definizione (classe generica). Classe con uno o più parametri di tipo.

La loro principale utilità sorge nella progettazione di strutture dati.

Esempio (classe generica). Possiamo ad esempio considerare la seguente classe che è utilizzata per memorizzare delle coppie chiave/valore:

```
public class Entry<K, V> {
    private K key;
    private V value;
    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() { return this.key; }
    public V getValue() { return this.value; }
}
```

Codice 1.16: Classe generica per memorizzare un dizionario

Al momento dell'istanziazione della classe, occorre specificare il tipo che si intende usare. Per quanto riguarda la successiva compilazione, tutte le informazioni sul tipo generico sono perse e le classi otterranno tipi *raw*.

```
public class Entry {
    private Object key;
    private Object value;
    public Entry (Object key, Object value) {
        this.key = key;
        this.value = value;
    }
    public Object getKey() { return this.key; }
    public Object getValye() { return this.value; }
}
```

Codice 1.17: Type erasure

Il type erasure è un meccanismo del compilatore introdotto per garantire la compatibilità retroattiva.

1.4.1.1 Metodi generici

Come le classi generiche, dipendono da un parametro di tipo.

Esempio (metodo generico)**.** Prendiamo un metodo che esegue lo scambio fra due generici elementi contenuti in un array.

```
public class Arrays {
    public static <T> void swap (T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

Codice 1.18: Scambio di elementi in un vettore

Come visibile, il parametro di tipo <T> va posto dopo il modificatore, ma prima del tipo di ritorno.

Per il richiamo dei metodi può essere omessa la dichiarazione del tipo dato che viene colta dal contesto.

Definizione (type checking)**.** Tutte le variabili utilizzate devono essere tipate e ne viene controllata la coerenza.

Definizione (type inference)**.** Il tipo viene determinato dal contesto di utilizzo.

Come le classi, anche i tipi possono estenderne altri: questo fenomeno viene chiamato *type bounding*. È inoltre consentito effettuare una molteplice estensione.

```
1 <T extends Runnable & AutoCloseable>
```

Codice 1.19: Type bounding multiplo

1.4.2 Wildcards

Le wildcards fungono da placeholder per dei tipi lista di elementi che estendono una classe base data. Se nell'utilizzo delle wildcard non ci è necessario conoscere il tipo ricevuto si possono sfruttare le *unbounded wildcards*.

```
1 public static boolean hasNull (ArrayList<?> elements) {
2     for (int i = 0; i < elements.size(); ++i) {
3         if (elements.get(i) == null) {
4             return true;
5         }
6     }
7     return false;
8 }
```

Codice 1.20: Unbounded wildcards

Altrimenti, se il tipo aspettato è sottotipo si hanno le *subtype wildcards*.

```
1 public void printNames (ArrayList<? extends Employee> staff) {
2     for (int i = 0; i < staff.size(); ++i) {
3         System.out.println(staff.get(i).getName());
4     }
5 }
```

Codice 1.21: Subtype wildcards

Infine, se il tipo aspettato è supertipo si hanno le *supertype wildcards*.

```
1 public void addManager (ArrayList<? super Manager> staff) {
2     staff.add(new Manager(...));
3 }
```

Codice 1.22: Supertype wildcards

1.5 UML

L'*Unified Modeling Language* fornisce una standardizzazione nel contesto del software engineering. Il nostro interesse si focalizza però su una sola funzionalità presente al suo interno: i class diagram[1]. Essi fungono da notazione grafica per costruire e rappresentare software OOP.

> **Definizione** (UML class diagram)**.** Diagramma che descrive la struttura di un sistema mostrandone:
>
> – le classi e le interfacce;
>
> – i loro rispettivi attributi;
>
> – le operazioni, ovvero i metodi;
>
> – le relazioni tra classi ed interfacce.

Con la UML class notation un singolo diagramma è utilizzato per fornire tutte le informazioni di una classe. Gli attributi e i metodi possono essere omessi, a seconda del livello d'astrazione:

* concettuale, rappresentazione dei punti del dominio;
* specifica, definizione delle interfacce che compongono il sistema;
* implementazione, descrizione delle classi e le relazioni fra queste.

Tabella 1.3: Notazione di visibilità UML

SIMBOLO	SIGNIFICATO
	`default`
+	`public`
−	`private`
#	`protected`

Le interfacce e le classi astratte vengono rappresentate rispettivamente tramite una 'I' in alto a destra ed una 'a' in alto a sinistra. Per quanto riguarda le relazioni fra classi, si hanno tre legami:

i. ereditarietà; ii. implementazione; iii. dipendenza.

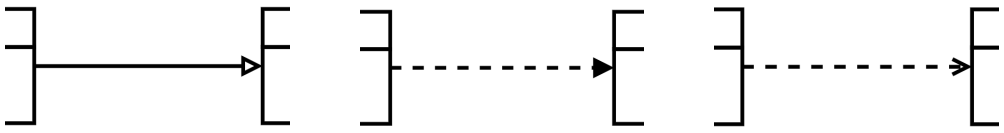


Figura 1.1: Simboli sintattici relazioni fra classi

La dipendenza, o uso, si presenta qualora:

a. un campo è istanza;

b. un metodo ha una classe come parametro.

[1] Cam24.

1.6 Interfacce funzionali

La caratteristica di questa tipologia di interfacce riguarda l'avere un solo metodo astratto. Nella dichiarazione occorre aggiungere il tag `@FunctionalInterface`. Possono essere rimpiazzate da *lambda expressions* e riferimenti a metodi. Le varie sintassi possibili per questi ultimi sono:

* `NomeClasse::nomeMetodo`, dove `nomeMetodo` è un metodo di istanza;
  Se tipo `nomeMetodo` $\text{type}_1 \cdot \ldots \cdot \text{type}_k \to \text{type}$
  allora tipo `NomeClasse::nomeMetodo` $\text{NomeClasse} \cdot \text{type}_1 \cdot \ldots \cdot \text{type}_k \to \text{type}$

* `NomeClasse::nomeMetodo`, dove `nomeMetodo` è un metodo statico;
  Se tipo `nomeMetodo` $\text{type}_1 \cdot \ldots \cdot \text{type}_k \to \text{type}$
  allora tipo `NomeClasse::nomeMetodo` $\text{type}_1 \cdot \ldots \cdot \text{type}_k \to \text{type}$

* `obj::nomeMetodo`, dove `obj` è un riferimento ad un metodo;
  Se tipo `nomeMetodo` $\text{type}_1 \cdot \ldots \cdot \text{type}_k \to \text{type}$
  allora tipo `obj::nomeMetodo` $\text{type}_1 \cdot \ldots \cdot \text{type}_k \to \text{type}$

* `NomeClasse::new`.
  Se tipo `NomeClasse` $\text{type}_1 \cdot \ldots \cdot \text{type}_k$
  allora tipo `NomeClasse::new` $\text{type}_1 \cdot \ldots \cdot \text{type}_k \to \text{NomeClasse}$

1.6.1 Lambda expressions

Si tratta essenzialmente di una funzione.

```
1 (arg1, ..., argn) → expr
2 (arg1, ..., argn) → { ... }
```

Codice 1.23: Espressione lambda

Sono utilizzabili laddove un'interfaccia funzionale è attesa; il compilatore controlla la compatibilità.

$$\text{type}_1 \cdot \text{type}_2 \cdot \ldots \cdot \text{type}_n \to \text{type}$$

Si ha la lista dei tipi di parametro ed infine il tipo di ritorno.

1.7 Programmazione concorrente

Il primo passo per sviluppare programmi concorrenti è quello di suddividere l'attività in dei *task*, questi possono essere eseguiti:

a. in un *thread*;
b. tramite un *executor*.

Definizione (threading)**.** Creazione e gestione di unità multiple di esecuzione all'interno di un singolo processo.

* **Binari**, programmi inattivi compilati in un formato accessibile da determinate architetture.
* **Processi**, astrazione che rappresenta binari in esecuzione.
* **Thread**, unità di esecuzione di un processo.

Se un processo contiene un solo thread è detto *single threaded*, altrimenti *multi threaded*.

Definizione (future)**.** Oggetto che rappresenta una computazione il cui risultato sarà disponibile in un momento futuro.

Sezione 2

XML

I documenti XML (*eXtensible Markup Language*) scambiati hanno queste regole sintattiche:

1. case sensitive;
2. annidamento ad albero;
3. tag come stringa senza spazi;
4. tag `xml` riservato per l'apertura del file;
5. elementi non vuoti aventi tag di inizio e fine;
6. inizio tag con lettera minuscola o underscore;
7. i caratteri possono essere alfanumerici e speciali;
8. nessun limite ai tag o alle informazioni contenute;

† indentatura consigliata, ma pur sempre opzionale.

Esercizio (correzione XML)**.** Dato un file XML, verificare la presenza di eventuali errori.

```
<!-- Ship Manifest -->
<?xml version="1.0" >
<fleet>
<Country>UK</country>
<type>Destroyer<\type>
<19 year of production>1934</19 year of production>
</fleat>
```

Codice 2.1: Documento XML errato

Sono presenti numerose imprecisioni, esattamente sette:

1. commento prima del tag di apertura;
2. mancata chiusura del tag header;
3. unmatching case del tag `<country>`;
4. backslash anziché slash nel tag `<type>`;
5. spazi interni al tag `<19 year of production>`;
6. typo nel tag `<fleet>`;

† mancata indentazione.

Nel web è usato HTML (*HyperText Markup Language*), una tipizzazione di XML.

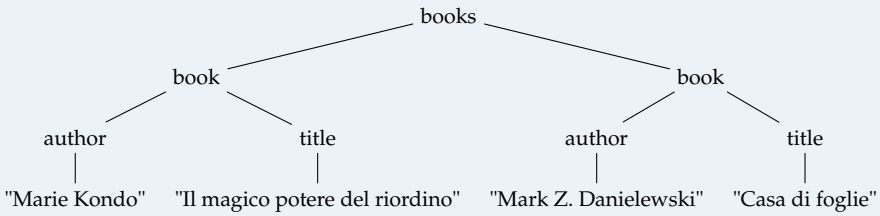
2.1 Struttura XML

Il metaformato XML è rappresentabile come un albero poiché ogni elemento, al di fuori della radice, ha un padre.

Esempio (albero XML)**.** Dato un qualunque codice XML, è possibile associarvi un albero.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<books>
    <book>
        <author>Marie Kondo</author>
        <title>Il magico potere del riordino</title>
    </book>
    <book>
        <author>Mark Z. Danielewski</author>
        <title>Casa di foglie</title>
    </book>
</books>
```

Codice 2.2: Codice XML riportabile come albero



2.1.1 Preambolo XML

Il preambolo contiene un solo tag di apertura e va posto nella prima riga del file.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

Codice 2.3: Tag di apertura di un file XML

Vediamo nel dettaglio gli attributi presenti nel tag di apertura.

Tabella 2.1: Attributi del tag di apertura di un file XML

ATTRIBUTO	INSERIMENTO	DEFAULT	ALTERNATIVA
`version`	obbligatorio	1.0	–
`encoding`	opzionale	UTF-8	UTF-16
`standalone`	opzionale	no	yes

2.1.2 Attributi e sottoelementi

Nel documento sono presenti attributi e sottoelementi per fornire informazioni. I primi sono eretti dalla coppia nome-valore, mentre i secondi si appoggiano su tag interni.

```xml
<automobile marca="Fiat" modello="Punto"></automobile>
<automobile marca="Ford" modello="Puma"></automobile>
<automobile marca="Audi" modello="A3"></automobile>
```

Codice 2.4: Attributi XML

```
<automobile>
    <marca>Fiat</marca>
    <modello>Punto</modello>
</automobile>
<automobile>
    <marca>Ford</marca>
    <modello>Puma</modello>
</automobile>
<automobile>
    <marca>Audi</marca>
    <modello>A3</modello>
</automobile>
```

Codice 2.5: Sottoelementi XML

Per quanto riguarda i commenti, la sintassi è quella che viene poi ripresa anche da HTML: `<!--comment-->`. Essendo un file prettamente data-oriented, non vanno abusati.

Nota bene (commenti pre-dichiarazione)**.** È obbligatorio non posizionare commenti antecedentemente al tag di inizio documento XML!

2.1.2.1 Caratteri speciali

Esistono inoltre dei particolari caratteri già utilizzati per specifiche funzioni legate al linguaggio; in tal caso si sfrutta un escape char: l'&.

Tabella 2.2: Caratteri speciali XML

CHAR	FUNZIONE
&	escape char
"	apici valori
<	apertura tag
>	chiusura tag

ALT	FUNZIONE
&amp;	and
&quot;	testo
&l;	minore
&gt;	maggiore

2.1.3 Namespace

La semplicità di XML implica che, se due applicazioni definiscono lo stesso elemento con significati differenti non c'è distinzione. Per ovviare si aggiunge un prefisso al nome tag per renderlo univoco:

```
prefisso:nometag
```

Tale prevenzione non è sufficiente poiché documenti diversi possono avere stesso namespace. Si risolve con l'implementazione dell'URI, il quale identifica univocamente l'elemento.

Definizione (*Unified Resource Identifier*)**.** Unione dell'URL (*Uniform Resource Locator*) e dell'URN (*Uniform Resource Name*) il cui ruolo è quello di identificare la risorsa.

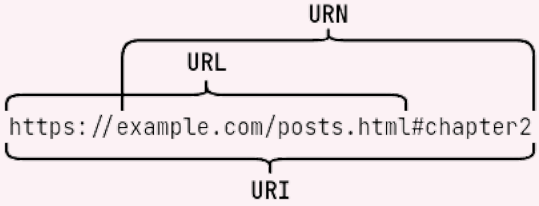


Figura 2.1: URI

2.1.4 CDATA

Per inserire del testo in un documento, utile nel caso sia anch'esso codice XML, esiste la possibilità di creare una sezione CDATA (*Character DATA*).

```
<codice>
    <![CDATA[
        <libro>
            <capitolo>
            </capitolo>
        </libro>
    ]]>
</codice>
```

Codice 2.6: XML CDATA

In questo caso non è richiesto l'utilizzo di escape char per l'inserimento di determinati componenti.

2.2 DTD

La *Document Type Definition* è uno schema della struttura di un generico file XML. Viene fornito, seppur non obbligatorio, per aiutare il parser nell'elaborazione.

DTD + indentation = well-formed XML

```
<?xml version="1.0"?>
<a>
    <b></b>
    <c>
        <d></d>
        <e></e>
    </c>
</a>
```

Codice 2.7: Documento XML ben indentato

La dichiarazione di un DTD definisce:

a. la lista degli attributi;
b. gli elementi strutturali;
c. il modello di contenuto.

Essa può esser posta in diverse maniere:

* internamente

```
<!DOCTYPE rootTag [
    <!ELEMENT ...>
]>
```

Codice 2.8: Dichiarazione del DTD interna

* esternamente

```
<!DOCTYPE rootTag SYSTEM "document.dtd">
```

Codice 2.9: Dichiarazione del DTD esterna

* mista

```
<!DOCTYPE rootTag SYSTEM "document.dtd" [
    <!ELEMENT ...>
]>
```

Codice 2.10: Dichiarazione del DTD mista

2.2.1 Elementi

Il DTD definisce la forma dei dati che possono occorrere all'interno dei tag XML:

- *any content*, `<!ELEMENT memo ANY>`;
- *empty content*, `<!ELEMENT br EMPTY>`;
- *simple content*, `<!ELEMENT message (#PCDATA)>`;
- *element content*, `<!ELEMENT note (to, from, title, message)>`;

Tabella 2.3: Ripetizione degli elementi contenuti

SIMBOLO	CARDINALITÀ	SINTASSI
	unico	`<!ELEMENT a (b)>`
,	obbligatori	`<!ELEMENT a (b, c, d)>`
|	esclusivi	`<!ELEMENT a (b | c | d)>`
?	opzionale	`<!ELEMENT a (b?)>`
+	uno o più	`<!ELEMENT a (b+)>`
*	zero o più	`<!ELEMENT a (b*)>`

  Per due, o più, elementi la sintassi è la seguente: `<!ELEMENT a (b, b+)>`. Per creare una sequenza illimitata si usa invece `<!ELEMENT a (b | c)*>`.

- *mixed content*, aggiunge tag al testo unendo CDATA ed elementi.

Esercizio (XML da DTD). Definire un documento XML coerente al seguente DTD.

```
<!DOCTYPE books [
    <!ELEMENT books (book+)>
    <!ELEMENT book (title,author,year,publisher,isbn,price)>
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT author (#PCDATA)>
    <!ELEMENT year (#PCDATA)>
    <!ELEMENT publisher (#PCDATA)>
    <!ELEMENT isbn (#PCDATA)>
    <!ELEMENT price (#PCDATA)>
]>
```

Codice 2.11: Documento XML da cui scrivere il DTD

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<books>
    <book>
        <title>Il lupo della steppa</title>
        <author>Hermann Hesse</author>
        <year>1927</year>
        <publisher>Mondadori</publisher>
        <isbn>9788804668060</isbn>
        <price>12</price>
    </book>
</books>
```

Codice 2.12: Documento DTD scritto dall'XML

Vediamo ora invece l'esempio opposto.

Esercizio (DTD da XML). Definire un documento DTD corretto per il seguente XML.

```
<movie>
    <title>Le Iene</title>
    <director>Q. Tarantino</director>
    <actor>H. Keitel</actor>
</movie>
```

Codice 2.13: Documento DTD da cui scrivere l'XML

.....

```
<!DOCTYPE movie [
    <!ELEMENT movie (title, director, actor)>
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT director (#PCDATA)>
    <!ELEMENT actor (#PCDATA)>
]>
```

Codice 2.14: Documento XML scritto dal DTD

2.2.2 Attributi

I singoli tag possono avere diversi attributi e vengono specificati nel DTD.

```
1 <!ATTLIST elementName attributeName attributeType attributeValue>
```

Codice 2.15: Dichiarazione DTD degli attributi di un elemento XML

Il tipo è ristretto a tre alternative:

Tabella 2.4: Tipologia attributi degli elementi

TIPO	VALORE	SINTASSI
CDATA	stringa	`<!ATTLIST message lang CDATA "Italiano">`
enum	lista	`<!ATTLIST person salutation (Mr | Mrs) "Mrs">`
ID	univoco	`<!ATTLIST user login ID #REQUIRED>`

Esistono inoltre dei vincoli per questi attributi:

Tabella 2.5: Modificatori dei valori degli attributi

VALORE	DESCRIZIONE
"value"	default
`#REQUIRED`	obbligatorio
`#IMPLIED`	opzionale
`#FIXED` "value"	fissato

2.2.3 Entità

Un solo DTD viene solitamente riferito da numerosi file XML in modo da garantire uno standard. Per evitare di riscrivere lo stesso testo vengono introdotte le entità che fungono da simil-costanti.

```
<!ENTITY entityName entityValue>
```

Codice 2.16: Dichiarazione di un'entità

Per riprendere quel dato in un campo testuale si usa la seguente sintassi: `<nome>&entityName;</nome>`. Internamente al DTD la dicitura è invece `<nome>%entityName;</nome>`.

Esempio (entità). Si riprende il nome dell'autore.

```
<!ENTITY autore "Lorenzo Rossi">
<document>
    <title>Introduzione ad XML</title>
    <author>&autore;</author>
</document>
```

Codice 2.17: Utilizzo di un'entità

Nota bene (entità con dichiarazione DTD interna). Il richiamo ad un'entità dichiarata all'inizio dell'XML tramite DTD interna non funziona! Ciò avviene siccome il parser non ha tempo di creare il riferimento.

2.3 XSD

L'*XML Schema Definition* sostituisce DTD grazie alla standardizzazione W3C e al maggior controllo; ad esempio, in DTD, il tag `<year>1990</year>` è valido, ma lo è altrettanto `<year>hi</year>`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns:xs="http://www.w3.rg/2001/XMLSchema"></xs:schema>
```

Codice 2.18: Dichiarazione schema XSD

La radice del file con estensione .xsd è `<schema>`.

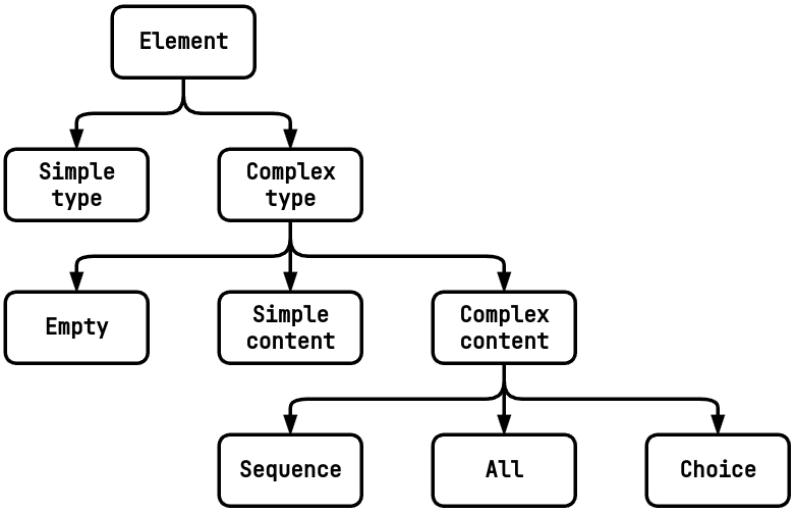


Figura 2.2: Elementi semplici e complessi

Esercizio (XSD da XML). Dato il seguente codice XML, definire un XSD valido che lo accetti.

```xml
<?xml version="1.0"?>
<catenaMontuosa>
    <monte>
        <nome>Monte Bianco</nome>
        <regione>Valle d'Aosta</regione>
        <altezza unitaMisura="metri">4818</altezza>
    </monte>
    <monte>
        <nome>Gransasso</nome>
        <regione>Abruzzo</regione>
        <altezza unitaMisura="metri">2912</altezza>
    </monte>
</catenaMontuosa>
```

Codice 2.19: Documento XSD costruito dall'XML

.....

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="catenaMontuosa">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="monte" minOccurs="1" maxOccurs="unbounded"/>
            <xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="monte">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="nome"/>
                <xs:element ref="regione" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element ref="altezza" minOccurs="0" maxOccurs="1"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="nome" type="xs:string"/>
    <xs:element name="regione" type="xs:string"/>
    <xs:element name="altezza">
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base="xs:integer">
                    <xs:attribute name="unitaMisura" type="xs:string"/>
                </xs:extension>
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Codice 2.20: Documento XSD costruito dall'XML

2.3.1 Elementi XSD semplici

Gli elementi basici, ed i rispettivi attributi, sono definiti fornendone nome e tipo.

```
<xs:element name="name" type="type"/>
<xs:attribute name="name" type="type"/>
```

Codice 2.21: Definizione elemento semplice

Gli attributi possono essere `default`, `fixed` o `use`: per definire l'opzionalità, o meno, del campo. Il tipo viene scelto tra varie alternative, le più comuni:

- `xs:date`
- `xs:string`
- `xs:decimal`
- `xs:time`
- `xs:boolean`
- `xs:integer`

Nota bene (datetime). La combinazione di `xs:date` e `xs:time`, `xs:dateTime`, richiede la seguente struttura: CCYY-MM-DDThh:mm:ss.

È inoltre possibile, come anticipato, impostare dei controlli sul dato contenuto nell'attributo.

Tabella 2.6: Restrizioni imposte agli attributi

NUMERI	STRINGHE
`minInclusive`	`length`
`minExclusive`	`minLength`
`maxInclusive`	`maxLength`
`maxExclusive`	`pattern`
`totalDigits`	`whiteSpace`

Il facets `whiteSpace` può essere:

* *preserve*, lascia gli spazi;
* *replace*, rimuove gli spazi;
* *collapse*, trim ai lati e riduzione degli spazi multipli ad uno solo.

2.3.2 Elementi XSD complessi

* `xs:sequence`
  Consente di specificare l'ordine di apparizione degli elementi.

```
<xs:element name="person">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="firstName" type="xs:string"/>
            <xs:element name="lastName" type="xs:string"/>
            <xs:element name="birthday" type="xs:date"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Codice 2.22: Elemento complesso `xs:sequence`

* `xs:all`
  Permette l'apparizione in qualsiasi ordine.

```
<xs:element name="person">
    <xs:complexType>
        <xs:all>
            <xs:element name="firstName" type="xs:string"/>
            <xs:element name="lastName" type="xs:string"/>
            <xs:element name="birthday" type="xs:date"/>
        </xs:all>
    </xs:complexType>
</xs:element>
```

Codice 2.23: Elemento complesso `xs:all`

* `xs:choice`
  Definisce un insieme di elementi.

```
<xs:element name="person">
    <xs:complexType>
        <xs:choice>
            <xs:element name="firstName" type="xs:string"/>
            <xs:element name="surName" type="xs:string"/>
        </xs:choice>
    </xs:complexType>
</xs:element>
```

Codice 2.24: Elemento complesso `xs:choice`

2.4 SAX

Il *Simple Api for Xml* permette di far scaturire degli eventi da catturare.

> **Definizione** (parser)**.** Software che valuta la sintassi di uno script quando viene eseguito.

Il processo del parser lancia gli eventi, ed un thread dell'handler resta costantemente in ascolto. Le principali funzioni sono:

a. recuperare il documento XML;

b. caricare i dati in memoria;

c. presentare all'applicazione un'interfaccia di alto livello.

Essenzialmente funge da layer fra l'XML e le applicazioni. Esistono due approcci al parsing:

* *ad eventi*, scansionamento totale e scagionamento di eventi tramite callback;
* *a modello*, passaggio dell'intero contenuto dell'xml che viene replicato in memoria.

Il secondo metodo, maggiormente utilizzato, permette di conoscere la struttura ad albero.

Tabella 2.7: Approcci al parsing

AD EVENTI	A MODELLO
leggero	struttura
verboso	completo

Data la mancanza di conservazione della struttura, SAX non è in grado di costruire file XML.

2.4.1 DOM

Il *Document Object Model* è uno sviluppo di SAX che fornisce delle funzionalità basilari:

- parsing
- browsing
- update
- create
- add
- remove

La struttura di un DOM verte sul concetto di *node*; ogni elemento, attributo e campo è definito tale. Si viene a creare la conversione da file XML, e relativi tag, ad albero, e relativi nodi.

2.5 XPath

Linguaggio di ricerca su XML che opera sulla struttura logica, e non sintattica, del documento. Il file è visualizzato come un albero con nodi generici: elementi, attributi e contenuto.

† Location path: specifica come spostarsi fra i nodi.

‡ Location step, formato da:

i. asse;
Insieme di nodi.

Tabella 2.8: Assi di XPath

NOME	SELEZIONE	NOME	SELEZIONE
self	corrente	*descendant*	figli fino alle foglie
child	figli	*descendant-or-self*	descendant e corrente
parent	padre	*following*	destri (−descendant)
attribute	attributi	*following-sibling*	fratelli destri
ancestor	avi fino alla radice	*preceding*	sinistri (−ancestor)
ancestor-or-self	ancestor e corrente	*preceding-sibling*	fratelli sinistri

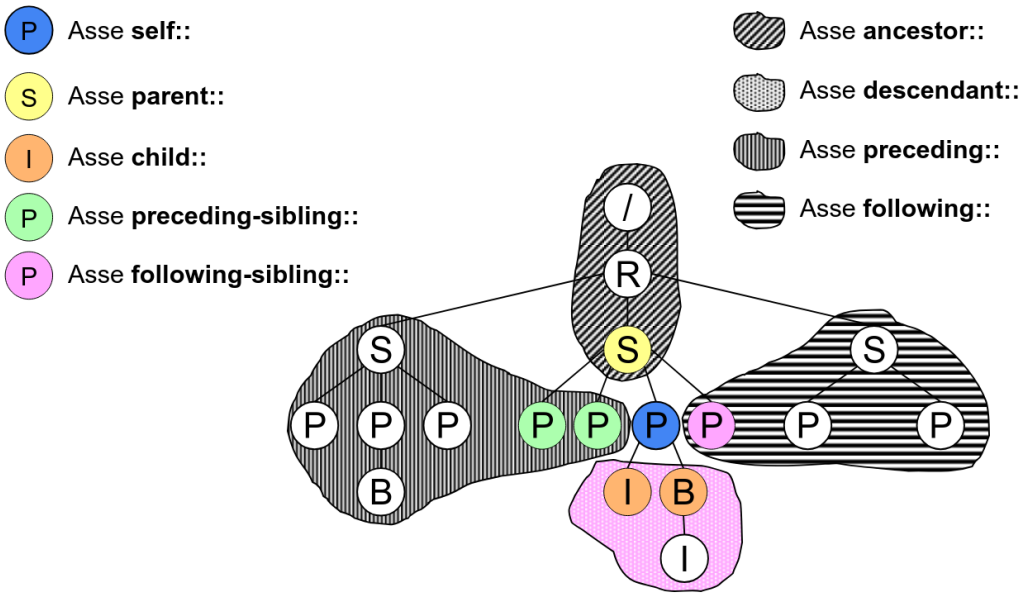


Figura 2.3: Grafica degli assi di XPath

Si sfruttano inoltre delle apposite abbreviazioni per indicare l'asse interessato.

Tabella 2.9: Sintassi abbreviata degli assi

STANDARD	SHORT
child	x
attribute	@a
descendant	/

STANDARD	SHORT
descendants	//
self	.
parent	..

ii. eventuali predicati.
Si possono aggiungere, opzionalmente, dei predicati.

axis:test[pred1][pred2]. . . [predN]

Esempio (filtri)**.** Ecco una query con relativa descrizione.

```
/collection/song[@album=//album[title="Lesbianitj"]/@ID]/title
```

Si ricercano i campi title delle song contenute in collection che abbiano come attributo album la stringa ricercata.

Gli operatori dei filtri sono: confronto, logici, insiemistici, raggruppamento e matematici.

Tabella 2.10: Operatori dei filtri

CONFRONTO
=
≠
<
≤
>
≥

MATEMATICI
+
−
*
/
%
()

LOGICI
∨
∧
¬

INSIEMISTICI
∪
∩
\

Esempio (operatori nei filtri)**.** Ecco una query con uso di operatori.

```
/letters/letter[sender!="Bob" and receiver="Alice"]/body
```

Si effettua una ricerca nelle lettere presenti impostando dei requisiti a livello di filtraggio. I corpi ottenuti dal risultato della query saranno stati ricevuti da Alice e scritti da una persona che non sia Bob.

2.5.1 Funzioni XPath

Sono definiti diversi metodi relativi a differenti tipi di oggetto.

```
Funzioni
├── Nodi
│   ├── count(...)
│   ├── last()
│   ├── position()
│   ├── document(...)
│   └── id("...")
├── Stringhe
│   ├── string(...)
│   ├── concat(s1, s2, ...)
│   ├── substring(s, a, b)
│   ├── substring-after(s, t)
│   ├── substring-before(s, t)
│   ├── contains(s, t)
│   ├── starts-with(s, t)
│   └── string-length(s)
├── Boolean
│   ├── boolean(o)
│   ├── true()
│   ├── false()
│   └── not(e)
└── Numeriche
    ├── number(o)
    ├── ceiling(x)
    ├── floor(x)
    ├── round(x)
    └── sum(x)
```

2.5.2 REGEX

Le *REGular EXpression* sono particolari sequenze di testo che schematizzano la ricerca di stringhe.

Tabella 2.11: Cheatsheet dei metacaratteri

REGEX	FILTRO
.	tutti i caratteri
\ w, \ d, \ s	parola, cifra, spazio
[abc]	qualunque
$[^{abc}]$	non
[a − z]	caratteri ASCII da 'a' a 'z'
a*, a+, a?	zero o più, almeno uno, opzionale
a{3}, a{7, }	esattamente tre, da sette in su

È poi consentito l'uso dei soliti operatori logici.

2.6 XSLT

L'*eXtensible Stylesheet Language for Transformation* trasforma i contenuti XML con approccio: cerca-trova-trasforma. Si instaurano delle query di ricerca e successive istruzioni di modifica.

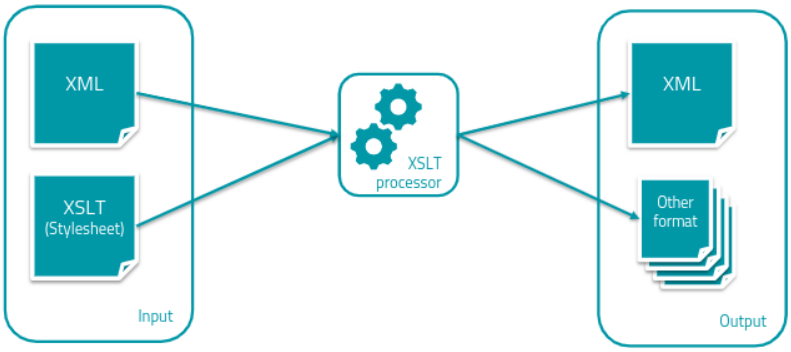


Figura 2.4: Modello XLST

Il documento ha radice `<Stylesheet>` e la sua dichiarazione in XML avverrà dopo il tag di apertura.

```
1 <?xml version="1.0"?>
2 <?xml-stylesheet type="text/xsl" href="prima.xsl">
```

Codice 2.25: Dichiarazione dello stile XLST in XML

È inoltre possibile definire un comando `template` che sfrutta l'attributo `match` per definire i nodi a cui sottoporlo: `<xsl:template match="/">`.

* lettura

```
1 <xsl:value-of select="/path/to/node">
```

Codice 2.26: Value of in XLST

* iterazioni

```
1 <xsl:for-each select="path/to/node">
```

Codice 2.27: Foreach in XLST

* ordinamento

```
1 <xsl:sort select="path/to/node">
```

Codice 2.28: Sort in XLST

* condizione

```
1 <xsl:if test="espressione">...</xsl:if>
```

Codice 2.29: If in XLST

* scelta

```
1 <xsl:choose>
2     <xsl:when test="espressione">...</xsl:when>
3     <xsl:otherwise>...</xsl:otherwise>
4 </xsl:choose>
```

Codice 2.30: Choose in XLST

Sezione 3

ONTOLOGIE

Si mira a rappresentare un dominio d'interesse. I punti cardinali da garantire sono:

- *formalità*, machine understandable;
- *concettualità*, si riferiscono ad un modello astratto;
- *esplicitezza*, basate su strutture di immediata applicazione;
- *condivisione*, catturano conoscenza riconosciuta da un gruppo.

Definizione (ontologia)**.** Collezione di termini, e relazioni fra loro, definibile come una tripla $O = (C,\ R,\ A)$ tale che:

* C = insieme di concetti;
* R = insieme di relazioni concettuali | $\forall r \in R \rightarrow R \subseteq C \times C$;
* A = insieme di assiomi.

Nota bene (ontologia non assiomatizzata)**.** Se $A = \emptyset$ l'ontologia non è assiomatizzata.

Esempio (ontologia)**.** Si osservi l'ontologia $O' = (C',\ R',\ A')$ dove:

- C' = {Entità, Oggetto, Persona, Meccanico, Automobile, Motore}
- R' = {è-un, ha-un, ripara}
  - è-un
    - (Oggetto, Entità)
    - (Persona, Entità)
    - (Meccanico, Persona)
    - (Automobile, Oggetto)
    - (Motore, Oggetto)
  - ha-un
    - (Automobile, Motore)
  - ripara
    - (Meccanico, Automobile)
- A' = {"$\forall a \in$ Automobile $\exists m \in$ Meccanico | ripara$(m,\ a)$"}

Ci sono poi le istanze che descrivono un'applicazione dell'ontologia.

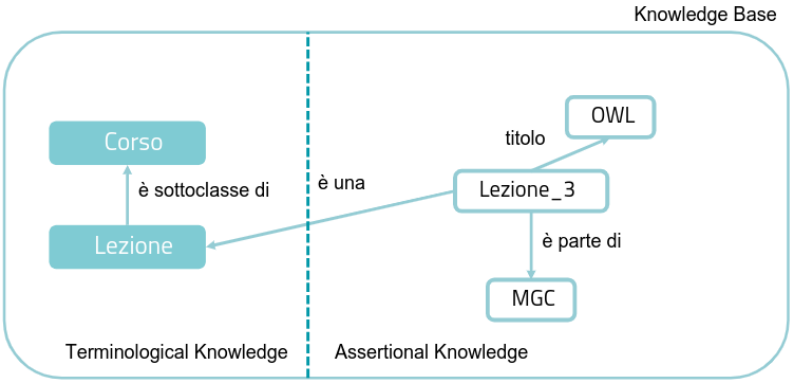


Figura 3.1: Istanza di un'ontologia

Per aiutarci nella realizzazione di ontologie utilizzeremo il tool Protégé[1].

3.1 Semantic web

Internet ha vissuto tre generazioni: *computer-centered*, *document-centered* e *data-centered*.

```
Web
├─ URL (Uniform Resource Locator)
├─ HTTP (HyperText Transfer Protocol)
└─ HTML (HyperText Markup Language)
```

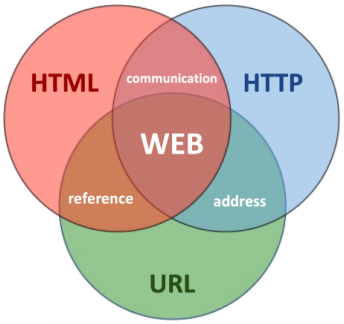


Figura 3.2: Tecnologie del web

Con l'avanzare degli anni aumenta il carico di dati ed anche la produttività delle ricerche.

Tabella 3.1: Evoluzione del web

ANNO	ERA	ROUTE	RICERCA
1979	PC	desktop	files
1980	web 1.0	WWW	directories
1998	web 2.0	social	keywords
2009	web 3.0	semantic	tags
2018	web 4.0	intellingent	agents

I motori di ricerca interpretano i dati, tramite parole chiavi, ma, a differenza degli umani, non hanno conoscenza implicita della realtà.

[1] Bio16.

3.1.1 Significato e comprensione

La ricerca per keywords risulterà sempre meno efficiente all'aumentare della mole di dati; il motivo risiede nelle ambiguità del linguaggio naturale.

Esempio (ambiguità della parola *"Polo"*)**.** Il termine *"Polo"* nella lingua italiana ha una moltitudine di significati e accezioni:

- sport;
- Nord;
- celeste;
- elettrico;
- estremità;
- maglietta;
- caramella;
- magnetico;
- automobile.

Per questo motivo è necessario contestualizzare il dato in modo da comprenderne il significato.

Definizione (informazione)**.** Elemento che, sviluppando il dato, consente di avere conoscenza esatta dei fatti.

> **Definizione** (dato)**.** Elemento alla base di tutto, su cui poi si costruisce la conoscenza.

Le informazioni trasmesse vengono interpretate dal destinatario secondo uno specifico linguaggio:

* *sintassi*, formato corretto della comunicazione;
* *semantica*, significato da interpretare al fine di comprendere l'informazione.

Si cerca di rappresentare la conoscenza per permettere alle macchine di simulare il ragionamento:

- modelli di conoscenza;
- teoria di ragionamento;
- scambio di informazioni.

Quando dialoghiamo, stiamo comunicando concetti astratti tramite l'uso di stringhe.

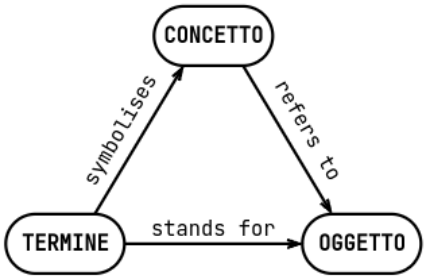


Figura 3.3: Triangolo semiotico

Il semantic web si pone come obiettivo quello di descrivere il significato, ovvero la semantica, dei contenuti in modo tale che possano essere interpretati dagli elaboratori.

3.1.1.1 Tassonomia e partonomia

Definizione (tassonomia)**.** Disciplina che si occupa della classificazione gerarchica di elementi. Data la natura ordinata, si appoggia spesso alla rappresentazione ad albero.

Definizione (partonomia)**.** Regola gerarchica basata sulle relazioni di appartenenza. La struttura grafica questa volta è un grafo deradicalizzato.

Tassonomia: "è un" Partonomia: "parte di"

3.2 RDF

Il *Resource Description Framework* permette di rappresentare la conoscenza tramite una tripla:

* **R**, chiave univoca identificabile tramite URI;
* **D**, rappresenta proprietà e relazione tra risorse;
* **F**, combinazione di protocolli web, basato su modelli formali.

L'obiettivo è quello di rendere frasi in linguaggio naturale *machine-processable*.

Esempio (RDF triple)**.** Data la frase *«VW Polo è stata commercializzata nel 1975»*, la relativa RDF triple corrisponde a

TRIPLA	LOGICA	FRASE	TIPO
R	soggetto	VW Polo	URI
D	predicato	è stata commercializzata nel	URI
F	oggetto	1975	URI/literal

Lo statement generato è dunque:

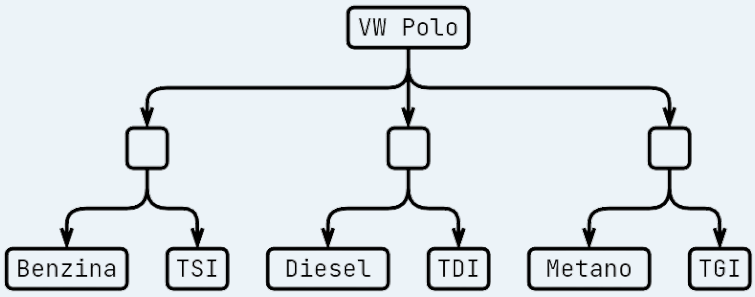
<VW Polo> <è stata commercializzata nel> <1975> .

Per rappresentare relazioni con valori multipli possono essere creati dei blank node.

Esempio (blank node)**.** La Volkswagen Polo ha diverse alimentazioni:

- Benzina, TSI
- Diesel, TDI
- Metano, TGI

Con i blank nodes ci è possibile effettuare una categorizzazione sui molteplici modelli dell'autovettura.



Si usano strutture dati per enumerare le risorse:

* *open list*, estendibile; Un singolo blank node contiene i puntatori agli elementi presenti nella lista. Il tipo di enumerazione è definito dal tag:
  - `Bag`, not ordered
  - `Seq`, ordered
  - `Alt`, alternatives
* *closed list*, non estendibile. Si genera una sequenza di blank nodes lunga quanto il numero di elementi a cui rimandano. La keyword in questo caso è semplicemente `List`.

3.2.1 Turtle

Esiste una sintassi che consente di abbreviare RDF in una successione di triple.

Definizione (tripla)**.** Sequenza soggetto-predicato-oggetto che permette di esprimere affermazioni su risorse.

* **Soggetto**: risorsa che si sta descrivendo.
* **Predicato**: proprietà o relazione che collega soggetto e oggetto.
* **Oggetto**: valore del predicato per il soggetto.

Esercizio (Turtle da RDF)**.** Dato il seguente estratto RDF, mutarlo in sintassi Turtle.

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:geography="http://www.example.org/geography#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <rdf:Description rdf:about="http://www.example.org/geography#Brussels">
    <geography:capitalOf rdf:resource="http://www.example.org/geography#Belgium"/>
    <geography:name xml:lang="fr">Bruxelles</geography:name>
    <rdf:type rdf:resource="http://www.example.org/geography#City"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.example.org/geography#Belgium">
    <rdf:type rdf:resource="http://www.example.org/geography#Country"/>
  </rdf:Description>
</rdf:RDF>
```

Codice 3.1: Compattazione da RDF a Turtle

```
<!-- prefix rdfs, geography, rdf -->
http://www.example.org/geography#Brussels
  geography:capitalOf http://www.example.org/geography#Belgium ;
  geography:name#lang=fr Bruxelles ;
  a http://www.example.org/geography#City .
http://www.example.org/geography#Belgium a http://www.example.org/geography#Country .
```

Codice 3.2: Compattazione da RDF a Turtle

3.2.2 Reificazione

RDF permette di creare statement che riguardano altri statement.

Esempio (reificazione)**.** Data la seguente frase

> *«Luca sostiene che il frate ha ucciso l'avvocato»*

si trova un primo statement in *"il frate ha ucciso l'avvocato"*, che a sua volta è incluso in un secondo che coincide con la frase stessa.

> *«Luca sostiene che «il frate» «ha ucciso» «l'avvocato» .»*

Si evidenzia con la sottolineatura un primo statement e con le virgolette caporali un secondo.

3.2.3 RDFS

RDF Schema permette di modellare una conoscenza terminologica. Consente dunque di creare delle classi ed istanziarle: `rdfs:Class`.

Nota bene (risorse RDFS)**.** In RDFs tutto è gestito come risorsa!

La gerarchizzazione di classi e proprietà è effettuata con: `rdfs:subClassOf` e `rdfs:subPropertyOf`.

Nota bene (implicazione logica transitiva)**.** La gerarchizzazione è transitiva.

Esempio (implicazione logica transitiva)**.** Se sono date

```
:Autovettura rdfs:subClassOf :Veicolo .
  :Veicolo rdfs:subClassOf :Oggetto .
```

allora è logico che

```
:Autovettura rdfs:subClassOf :Oggetto .
```

Altre proprietà sono:

- `rdfs:seeAlso`
- `rdfs:isDefinedBy`
- `rdfs:comment`
- `rdfs:label`

Le implicazioni logiche viste poco fa vengono effettuate da un reasoner.

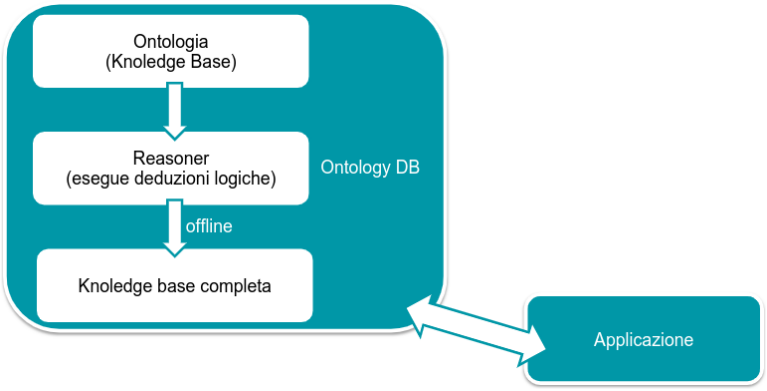


Figura 3.4: Rapporto ontologia-reasoner-conoscenza

3.3 Model-theoretic semantics

La semantica formale è necessaria per effettuare deduzioni, ossia implicazioni logiche, valide.

Tabella 3.2: Componenti della semantica RDFS

INGREDIENTE	SINTASSI
preposizioni	$\mathbb{P}$
relazioni d'implicazione	$\models \subseteq 2^{\mathbb{P}\times\mathbb{P}}$
logica	$L = (\mathbb{P}, \models)$
interpretazione	$I \models p \; \forall p \in \mathbb{P}$

I modelli definiscono se una preposizione sia vera oppure falsa.

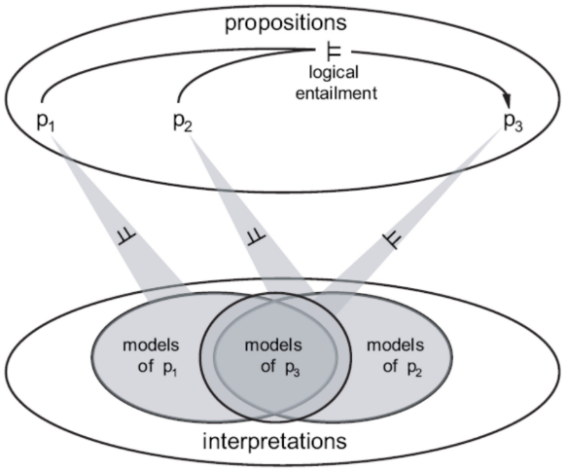


Figura 3.5: Interpretazioni di preposizioni

3.3.1 Interpretazione

> **Definizione** (vocabolario)**.** Insieme di termini, ossia URI e literal, del modello che definiscono concetti e relazioni usati per descrivere e rappresentare un dominio d'interesse.

Esistono numerosi insiemi di vocabolari pubblici[2] ed utilizzabili.

* **Interpretazione semplice**
  Un'interpretazione semplice I di un determinato vocabolario V consiste in:
  - I_P insieme delle proprietà;
  - I_R insieme non vuoto di risorse;
  - I_L funzione i letterali tipati di V nel set IR di risorse;
  - LV sottoinsieme di I_R contenente I letterali di V non tipati;
  - I_S funzione che mappa gli URI di V alle risorse di I_R e alle proprietà di I_P;
  - I_{EXT} funzione che assegna a ciascuna proprietà un insieme di coppie di elementi di I_R.

  Data un'interpretazione, una tripla può essere valutata come booleana.
* **Interpretazione RDF**
  x è una proprietà se connessa a `rdf:Property` tramite `rdf:type`.
* **Interpretazione RDFS**
  Devono esser soddisfatti i seguenti criteri:
  - ogni risorsa è di tipo `rdfs:Resource`;
  - ogni letterale è di tipo `rdfs:Literal`;
  - x è connesso dalla proprietà `rdfs:domain` a y e x collega u e v, allora u è di tipo y;
  - x connesso dalla proprietà `rdfs:range` a y e x collega u e v, allora v è di tipo y;
  - `rdfs:subPropertyOf` connette ogni proprietà a sé stessa;
  - se `rdfs:subPropertyOf` connette x a y e y a z, allora collega anche x a z;

[2] Gro24.

- se `rdfs:subPropertyOf` connette x ad y, allora ogni coppia di risorse della prima apparterrà anche alla seconda;
- ogni classe x è sottoclasse di `rdfs:Resource`;
- se x è `rdfs:subClassOf` di y, allora entrambe sono classi;
- `rdfs:subClassOf` connette ogni classe con sé stessa;
- come per `rdfs:subPropertyOf`, se `rdfs:subClassOf` connette x a y e y a z, allora collega anche x a z.

```
interpretazione RDFS
└─ interpretazione RDF
   └─ interpretazione semplice
```

Nota bene (insiemi delle funzioni)**.** L'insieme di partenza di una funzione prende il nome di dominio, mentre quello di arrivo viene detto range.

3.3.2 Regole di deduzione

Le regole di implicazione definiscono una semantica operazionale.

$$\boxed{\frac{a > b \qquad b > c}{a > c} = \frac{\text{premessa}}{\text{implicazione}}}$$

Definizione (assioma)**.** Verità ammessa senza discussione, evidente di per sé.

Esempio (assioma)**.** Viene posto un assioma come regola di deduzione.

$$\frac{}{u\ a\ x\ .}$$

Il blank imposto nell'implicazione indica l'always true.

3.3.2.1 Mondo aperto

L'assunzione del mondo aperto limita RDF rendendolo obsoleto. Principalmente, non viene permesso di esprimere l'appartenenza di un'istanza ad una classe in modo disgiuntivo fra due classi.

Esempio (istanza non disgiunta)**.** Prendiamo un asino: animale erbivoro. In RDF non è possibile esprimere una negazione o rappresentare la disgiunzione da altri insiemi.

animal:Asino rdf:type animale:Vegetariano .

animal:Asino rdf:type animale:Carnivoro .

Le due preposizioni presentate non generano una contraddizione.

Inoltre, è impossibile dichiarare la cardinalità dei vincoli.

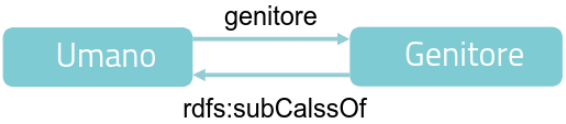


Figura 3.6: Assenza di cardinalità dei vincoli relazionali

3.4 OWL

L'*Ontology Web Language* è uno standard W3C per modellare ontologie. Viene data all'utente la possibilità di scegliere fra diversi gradi di espressività:

```
FOL
└─ OWL Full
   └─ OWL DL
      └─ OWL Lite
```

Dove FOL è l'acronimo di *First Order Logic*.

Tabella 3.3: Differenze gradi OWL

OWL		
Full	DL (*Description Logic*)	Lite
indecidibile	decidibile	decidibile
espressi1vo	supporto	poco espressivo
complicato	complessità $n \cdot e$	complessità e

3.4.1 Sintassi

Viene basato su RDF, dunque una tipizzazione di XML. Un'ontologia OWL è espressa in termini di classi e proprietà.

```
@prefix rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" .
@prefix xsd="http://www.w3.org/2001/XMLSchema#" .
@prefix rdfs="http://www.w3.org/2000/01/rdf-schema#" .
@prefix owl="http://www.w3.org/2002/07/owl#"" .
```

Codice 3.3: Header OWL

In OWL DL si ha la logica descrittiva SROIQ(D):

- **S** (*nominalS*);
- **O** (*Object properties*);
- **Q** (*Quantifiers*);
- **R** (*Roles*);
- **I** (*Inverse properties*);
- **D** (*Data types*).

3.4.2 Costrutti

OWL offre un ricco vocabolario per la definizione e la gestione di ontologie, permettendo di descrivere concetti, relazioni e istanze in modo rigoroso e strutturato. Dispone quindi di costrutti base per fornire struttura alle entità, e logici per esprimere relazioni complesse fra le prime.

3.4.2.1 Costrutti base

I tre pilastri di OWL sono: classi, proprietà ed individui. Questi sono dichiarati come istanze di classe RDFS.

3.4.2.1.1 Classi

Esistono due moduli predefiniti in OWL:

i. `owl:Thing`, contiene tutti gli individui $T \equiv C \cup \neg C$

ii. `owl:Nothing`, non contiene nulla $\bot \equiv C \cap \neg C$

La sintassi per dichiarare una classe è `:Book a owl:Class .`

3.4.2.1.2 Proprietà

Esistono due tipi di proprietà:

* *object properties*, hanno classi come range;

```
:autore a owl:ObjectProperty ;
rdfs:domain :Libro ;
rdfs:range :Scrittore .
```

Codice 3.4: Definizione di una object property

* *datatype properties*, hanno datatype come range.

```
:annoDiPubblicazione a owl:DatatypeProperty ;
rdfs:domain :Libro ;
rdfs:range xsd:integer .
```

Codice 3.5: Definizione di una datatype property

3.4.2.1.3 Individui

Definiti tramite l'appartenenza ad una classe: `:Scheletri a :Libro` . Sono le istanze d'una classe precedentemente dichiarata, vi è inoltre la possibilità di creare individui slegati dalla classe:

```
:Ornitorinco a owl:NamedIndividual .
```

Chiaramente è implicita l'appartenenza a `owl:NamedIndividual` e `owl:Thing`.

3.4.2.1.4 Gerarchie

Legano le risorse alla classe.

Esempio (gerarchie)**.** Data la seguente dichiarazione OWL

```
:Fumetto a owl:Class ;
    rdfs:subClassOf :Libro .
:Libro a owl:Class ;
    rdfs:subClassOf :Manoscritto .
:Manoscritto a owl:Class .
```

Codice 3.6: Gerarchie OWL

si può inferire che

$$\text{Fumetto} \subseteq \text{Libro} \qquad \text{Libro} \subseteq \text{Manoscritto}$$

Oltre a ciò, è facile dedurre che Fumetto $\subseteq$ Manoscritto.

È inoltre fornita una scorciatoia per definire insiemi di classi reciprocamente disgiunte.

```
[] a owl:AllDisjointClasses ;
    owl:members (
    :Autovettura
    :Cane
    :Frutta ) .
```

Codice 3.7: OWL disgiunzione fra classi

Lo stesso avviene per le proprietà con `owl:AllDisjointProperties`.

Tabella 3.4: Gerarchie e unicità

SINTASSI	SIGNIFICATO
`owl:AllDisjointClasses`	tutte classi disgiunte
`owl:AllDisjointProperties`	tutte proprietà disgiunte
`owl:disjointWith`	disgiunto con
`owl:equivalentClass`	stessa classe
`owl:sameAs`	stesso di
`owl:differentFrom`	diverso da
`owl:AllDifferent`	tutte diverse

Ci sono poi le classi chiuse, che presentano una scelta esclusiva.

```
<owl:Class rdf:about="Amico">
    <owl:oneOf rdf:parseType="Collection">
        <Person rdf:about="Carlo"/>
        <Person rdf:about="Franco"/>
    </owl:oneOf>
</owl:Class>
```

Codice 3.8: OWL classi chiuse

3.4.2.2 Costruttori logici

OWL permette di definire classi a partire da espressioni logiche su altre classi:

- `owl:intersectionOf`
- `owl:unionOf`
- `owl:complementOf`

Questi costrutti permettono di costruire classi complesse.

```
:Venduto rdf:type owl:Class ;
    owl:intersectionOf (
        :Acquistato
        :InMagazzino
    ) .
:Persona rdf:type owl:Class ;
    owl:unionOf (
        :Minorenne
        :Maggiorenne
    ) .
:Vegetale rdf:type owl:Class ; owl:complementOf :Animale .
```

Codice 3.9: Costrutti logici OWL

3.4.2.3 Vincoli di proprietà

Un altro strumento per descrivere classi complesse sono le restrizioni che si vanno ad imporre.

Tabella 3.5: Restrizioni e vincoli di proprietà

VALORI	CARDINALITÀ
`owl:hasValue`	`owl:cardinality`
`owl:allValuesFrom`	`owl:minCardinality`
`owl:someValuesFrom`	`owl:maxCardinality`

* Vincoli di proprietà con costanti

```
:LibriDiHermannHesse a owl:Class ;
    rdfs:subClassOf [
        a owl:Restriction ;
        owl:onProperty :autore ;
        owl:hasValue :HermannHesse
    ] .
```

Codice 3.10: `hasValue`

Estrazione di una sottoclasse da una condizione imposta.

* Vincoli di proprietà con legami stretti

```
:Libro a owl:Class ;
    rdfs:subClassOf [
        a owl:Restriction ;
        owl:onProperty :autore ;
        owl:allValuesFrom :Scrittore
    ] .
```

Codice 3.11: `allValuesFrom`

Tutti i risultati della ricerca hanno una correlazione.

* Vincoli di proprietà con legami deboli

```
:Lettore a owl:Class ;
    rdfs:subClassOf [
        a owl:Restriction ;
        owl:onProperty :legge ;
        owl:someValuesFrom :Libro
    ] .
```

Codice 3.12: `someValuesFrom`

Almeno un elemento soddisfa il controllo.

* Vincoli di proprietà con restrizioni di cardinalità

```
:Trilogia a owl:Class ;
    rdfs:subClassOf [
        a owl:Restriction ;
        owl:onProperty :numeroVolumi ;
        owl:cardinality 3
    ] .
```

Codice 3.13: `cardinality`

Dichiarazione stretta correlata alla numerazione.

3.4.2.4 Relazioni tra proprietà

Le proprietà sono collegamenti fra individui e consentono la creazione di un modello di conoscenza.

Tabella 3.6: Relazioni tra proprietà

NOME	SINTASSI	ESEMPIO
gerarchia	`rdfs:subPropertyOf`	isMadeOf $\subseteq$ consistsOf
inversione	`owl:inverseOf`	reads $\equiv^{-1}$ isReadBy
equivalenza	`owl:equivalentProperty`	composedOf $\equiv$ consistsOf
transitività	`owl:TransitiveProperty`	$\text{in}(a,\ b)\ \&\ \text{in}(b,\ c) \implies \text{in}(a,\ c)$
simmetria	`owl:SymmetricProperty`	$\text{near}(a,\ b) \implies \text{near}(b,\ a)$
funzionale	`owl:FunctionalProperty`	$\text{hasOnly}(a,\ b)\ \&\ \text{hasOnly}(a,\ c) \implies b = c$
funzionale^{-1}	`owl:InverseFunctionalProperty`	$\text{author}(b,\ a)\ \&\ \text{author}(c,\ a) \implies b = c$
asimmetria	`owL:AsymmetricProperty`	$\text{son}(a,\ b) \implies \neg\text{son}(b,\ a)$
riflessiva	`owL:ReflexiveProperty`	divider(17, 17)
irriflessiva	`owL:IrriflexiveProperty`	$\text{son}(a,\ b) \implies a \neq b$

3.5 SPARQL

Il *Simple Protocol and RDF Query Language* è utile per la creazione di query su RDF e OWL. Il tutto è basato su RDF Turtle e graph pattern matching.

Definizione (graph pattern)**.** Tripla RDF contenente variabili in posizioni arbitrarie.

Esempio (graph pattern)**.** Ricerca per nazione e rispettiva capitale.

```
?country dbo:capital ?capital .
```

Tabella 3.7: Operatori SPARQL

UNARI	
`!A`	`isBLANK(A)`
`+A`	`isLiteral(A)`
`-A`	`STR(A)`
`BOUND(A)`	`LANG(A)`
`isURI(A)`	`DATATYPE(A)`

BINARI		
=	$\leq$	+
!=	$\geq$	−
<	&&	*
>	||	/

DBpedia offre un editor di query[3] relativo al proprio database.

```
SELECT ?author_name ?title
FROM <http://dbpedia.org>
WHERE {
    ?author rdfs:label ?author_name .
}
```

Codice 3.14: Sintassi SPARQL

[3] Vir24.

3.5.1 Modificatori

Come in SQL (*Structured Query Language*), il risultato delle query è alterabile.

Tabella 3.8: Modificatori di risultato delle query SPARQL

KEYWORD	KEYWORD	KEYWORD
`ORDER BY ASC (?var)`	`FILTER(?var > 100)`	`REGEX(String, Pattern)`
`ORDER BY DESC (?var)`	`FILTER (LANG(?var) = "de")`	`sameTERM(A, B)`
`LIMIT 10`	`FILTER(!BOUND(?var))`	`langMATCHES(A, B)`
`OFFSET 5`	`MINUS(triple)`	`UNION triple . FILTER()`

```
SELECT ?author ?work xsd:integer(?date) AS ?year
FROM <http://dbpedia.org>
WHERE {
    ?author rdf:type dbo:Writer .
    ?author dbo:notableWork ?work .
    ?work dbp:releaseDate ?date .
} ORDER BY DESC (?year)
LIMIT 15
```

Codice 3.15: Ridenominazione tramite `AS`

3.5.1.1 Funzioni di aggregazione

Vi è innanzitutto `COUNT(?var)`, la quale riporta il numero di occorrenze riscontrate.

Tabella 3.9: Funzioni di aggregazione

KEYWORD	FUNZIONE	KEYWORD	FUNZIONE
`SUM`	somma	`MAX`	maggiore
`AVG`	media	`SAMPLE`	random
`MIN`	minore	`GROUP_CONCAT`	concatenamento

3.5.2 Sub query

Sempre riprendendo il formato di SQL, è possibile includere dell query internamente a delle altre.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?author ?influencer ?work
FROM <http://dbpedia.org>
WHERE {
    {
        SELECT ?author ?influencer
        FROM <http://dbpedia.org>
        WHERE {
            ?author rdf:type dbo:Writer ;
                    dbo:influencedBy ?influencer .
        }
    }
    ?influencer dbo:notableWork ?work .
}
```

Codice 3.16: Sub query

3.5.3 Path di proprietà

Definizione (path di proprietà)**.** Possibile percorso tra due nodi RDF.

Tabella 3.10: Path di proprietà

NOME	SPARQL
inversa	`?x ^foaf:knows ?y .`
sequenza	`?x foaf:knows/foaf:knows ?y .`
arbitraria	`?x foaf:knows+ ?y .`
negata	`?x !foaf:knows ?y .`
composta	`?x !(foaf:knows/foaf:name) ?y .`

3.5.4 SPARQL protocol

Oltre a modello di interrogazione, agisce anche da protocollo on top: spedisce query su RDF tramite HTTP. Un URI SPARQL è composto da tre parti:

a. URL di un endpoint SPARQL;

b. grafo RDF da interrogare;

c. query SPARQL.

Esempio (SPARQL protocol)**.** Ecco una query SPARQL spedita on top ad HTTP.

https://dbpedia.org/sparql?named-graph-uri=http://dbpedia.org&query=SELECT ?author ?influencer ?work FROM <http://dbpedia.org> WHERE SELECT ?author ?influencer FROM <http://dbpedia.org> WHERE ?author rdf:type dbo:Writer ; dbo:influencedBy ?influencer . ?influencer dbo:notableWork ?work .

3.5.5 Query SPARQL

Oltre alla solita `SELECT` sono presenti altre forme che non restituiscono le variabili associate alla richiesta. Si tratta di altri tipi di interrogazioni:

* `CONSTRUCT`, restituisce un grafo RDF dalle variabili associate al modello;
* `ASK`, restituisce un booleano che indica la possibile corrispondenza;
* `DESCRIBE`, restituisce un grafo RDF che descrive le risorse.

3.5.5.1 Funzioni di aggiornamento

In aggiunta alle query è possibile andare a manipolare direttamente la struttura del modello:

- `INSERT DATA triple`
- `DELETE DATA triple`
- `INSERT template WHERE pattern`
- `DELETE template WHERE pattern`
- `LOAD <uri> INTO GRAPH <uri>`
- `CLEAR GRAPH <uri>`
- `CREATE GRAPH <uri>`
- `DROP GRAPH <uri>`

3.6 Linked Data Mashup

Le ontologie creano la topologia dei Linked Data. Lo SKOS (*Simple Knowledge Organization System*) è basato su RDFS e mette a disposizione le seguenti proprietà:

- `skos:Concept`
- `skos:narrower`
- `skos:broader`
- `skos:related`
- `skos:exactMatch`
- `skos:narrowMatch`
- `skos:broadMatch`
- `skos:relatedMatch`

Molti dati necessari alla creazione di una qualsiasi ontologia sono già presenti nel web[4], questi sono identificati tramite URI ed accessibili con HTTP. Il processo di pubblicazione online di un dataset personale vede i seguenti passaggi:

$$\text{data type} \rightarrow \text{data preparation} \rightarrow \text{data storage} \rightarrow \text{publishment}$$

Le pratiche consigliate per questo tipo di procedimenti sono:

1. spiegare il processo di creazione;
2. selezionare dataset riutilizzabili;
3. modellare linked data indipendenti dalla loro applicazione;
4. scegliere una licenza open source;
5. nominare gli URI ponderatamente;
6. descrivere i concetti con vocabolari definiti;
7. usare rappresentazioni standard;
8. fornire modi per i motori di ricerca di accedere ai dati;
9. annunciare cambiamenti del dataset su domini autorevoli;
10. assicurarsi che il dataset rimanga disponibile.

[4] Dat07.

Sezione 4

JAVA

Esistono delle norme generali relative ad ogni codice scritto.

4.1 JavaDoc

Strumento per generare, a partire dal codice di un progetto, la relativa documentazione in formato HMTL. Gli IDE sfruttano quest'ultima per mostrare il funzionamento del programma.

```
1 // inline comment
2
3 /*
4  * multiline comment
5  */
6
7 /**
8  * javadoc comment
9  */
```

Codice 4.1: Commenti JavaDoc

Come riportato, si tratta essenzialmente di commenti, preferibilmente in lingua inglese, arricchiti da alcuni tag specifici riconosciuti:

- `@author`
- `@code`
- `@link`
- `@param`
- `@return`
- `@see`
- `@since`
- `@throws`

Java fornisce la documentazione[1] rispetto sé stesso. In IntelliJ IDEA, l'IDE (*Integrated Development Environment*) usato nel corso, esiste una funzionalità per la generazione della documentazione

`'Tools'` → `'Generate JavaDoc'`

[1] Jav23.

4.2 Principi SOLID

Il termine SOLID è in realtà un acronimo dato dall'unione di cinque principi:

Principio (Single responsibility)**.** Una classe deve avere una singola responsabilità! Molti la possono usare, ma solo un'istituzione potrà modificarla.

> **Esempio** (Single responsability principle)**.** Una classe che gestisce dei dati, ma ha anche una funzione integrata per stampare a video con `System.out.println` assume una doppia responsabilità. Occorre dunque separare le attività in due classi distinte.

Principio (Open-closed)**.** Le componenti software sono aperte alle estensioni, ma chiuse alle modifiche!

* Un modulo è **aperto** se permette le estensioni.
* Un modulo è **chiuso** se è disponibile per l'uso di altre componenti senza la necessità di conoscere l'esatta implementazione.

Principio (Liskov substitution)**.** In un programma è sempre possibile sostituire le istanze di una classe con le istanze di una sua sottoclasse senza alterare il comportamento del programma!

> **Lemma** (Liskov substitution)**.** Sia $q(x)$ una proprietà verificata da tutti gli oggetti $x \in T$. Allora $q(y)$ sarà verificata da tutti gli oggetti $y \in S$, dove S è sottotipo di T.
>
> > **Definizione** (proprietà)**.** Campo accessibile tramite un metodo getter.

Principio (Interface segregation)**.** Meglio avere diverse interfacce specifiche al posto di una singola interfaccia generale! Nessuna classe dovrebbe essere forzata ad essere dipendente di metodi che non vengono utilizzati.

Principio (Dependency inversion)**.** Le dipendenze devono essere sempre verso le astrazioni, mai verso le concretizzazioni! Le interfacce non dipendono dalle classi concrete, entrambe devono infatti dipendere dalle astrazioni.

4.3 Code smells

L'intento è quello di fornire una base stabile al codice in modo da prevenire problemi futuri. Di seguito una lista dei più comuni code smells:

- **codice morto**, non viene eseguito;
- **eccessivi commenti**, metodo poco chiaro;
- **metodi con troppi parametri**, non più di 3/4;
- **metodi lunghi**, mai più di 10/15 righe di codice;
- **codice duplicato**, stessa funzione, doppio codice;
- **data clumps**, variabili distinte dipendenti tra loro;
- **classi grandi**, possibilità di responsabilità molteplici;

- **campi temporanei**, uso di appoggi spesso vuoti e ignorati;
- **invidia**, metodo che usa dati di un altro oggetto anziché i propri;
- **eredità rifiutata**, utilizzo parziale di campi e metodi della superclasse;
- **gerarchie parallele**, l'estensione di una classe ne costringe anche un'altra;
- **classi con stessa interfaccia**, due classi con medesima funzione duplicata;
- **tipi primitivi**, occorre sfruttare i tipi evoluti per le informazioni strutturate;
- **intimità non appropriata**, classe che utilizza campi o metodi di un'altra classe;
- **modifiche a cascata**, alterazione di metodi a partire da un cambiamento d'una singola classe.

4.4 Testing

Un test è tale se riproducibile, verificabile e misurabile.

> **Definizione** (test)**.** Codice che verifica il corretto funzionamento di un altro.
>
> > **Definizione** (state testing)**.** Lo stato del codice dopo l'esecuzione è quello atteso.
> >
> > **Definizione** (behavior testing)**.** Gli eventi o le istruzioni eseguite sono quelle attese.

Utili per verificare la correttezza logica e comportamentale in ogni anfratto di codice scritto. Esistono diversi livelli di testing:

i. **unit testing**, test del singolo metodo della specifica classe;
ii. **integration testing**, integrazione corretta tra le diverse componenti;
iii. **system testing**, il sistema nella sua interezza soddisfa le funzionalità richieste.

> **Definizione** (test coverage)**.** Percentuale di linee di codice testate.

Durante la fase di testing sarebbe opportuno creare dei mock, ambienti dimostrativi.

4.4.1 JUnit

Framework di testing opensource[2] che permette la creazione di classi parallele a quelle sviluppate per l'inserimento dei test. Per identificarle al meglio si adopera il suffisso -Test.

```
1 @Test
2 public void m5() {
3     list.add("test");
4     assertFalse(list.isEmpty());
5     assertEquals(1, list.size());
6 }
```

Codice 4.2: Metodo di test

[2] Tea23.

Per verificare il corretto esito si hanno gli assert methods.

> **Definizione** (JUnit Platform)**.** Layer base che permette l'esecuzione dei test.
>
> > **Definizione** (JUnit Jupiter)**.** Ambiente di test eseguito dalla JUnit Platform.
> >
> > **Definizione** (JUnit Vintage)**.** Spazio legacy che supporta i vecchi test.

In JUnit le annotazioni per indicare i test sono:

- `@Test`, standard;
- `@Disabled`, disabilitato;
- `@AfterEach`, eseguito dopo;
- `@BeforeEach`, eseguito prima;
- `@RepeatedTest(n)`, eseguito n volte;
- `@Tag("<TagName>")`, associa un'etichetta;
- `@AfterAll`, eseguito dopo di tutti gli altri;
- `@BeforeAll`, eseguito prima di tutti gli altri.

Un insieme di test, test suite, può essere selezionato tramite `@SelectPackages` o `@SelectClasses`.

4.5 Apache Jena

Framework open source[3] per il Semantic Web che mette a disposizione delle API[4] per l'estrazione e la scrittura di dati su ontologie. Esistono diversi moduli erogati:

- ARQ (SPARQL), motore di query;
- RDF API, creazione e lettura di grafi RDF;
- Fuseki, esposizione delle ontologie tramite HTTP;
- Inference API, gestione del contenuto delle ontologie;
- Ontology API, interazione con ontologie RDFS ed OWL;
- TDB (*Triple DataBase*), salvataggio in un database di tuple.

Le classi principali sono:

* *Model*, grafi RDF;
* *InfModel*, grafi RDF con inferenza;
* *OntModel* (extends *InfModel*), interpreta OWL-DL.

Jena da accesso a vari oggetti per la gestione dei dati:

```
Model, raccolta di statement RDF
└─ Statement, tripla soggetto-predicato-oggetto
   └─ componenti, Resource Literal Property RDFNode ...
```

4.5.1 RDF Model

Un modello RDF può essere vuoto.

```
1 Model family = ModelFactory.createDefaultModel();
```

Codice 4.3: Modello RDF vuoto

Inoltre, è possibile estrarre il modello direttamente da file.

[3] Fou11a.
[4] Fou11b.

```
1 Model europe = RDFDataMgr.loadModel(filepath+name);
```

Codice 4.4: Modello RDF da file

In aggiunta a queste due possibilità, si può aggiungere dati da un altro modello.

```
1 model.add(Model othermodel);
2 model.add(RDFDataMgr.loadModel(filepath+name));
3 RDFDataMgr.read(model, filepath+name );
```

Codice 4.5: Modello RDF da altro modello

Inoltre è possibile effettuare delle operazioni sui modelli.

```
1 model1.setNsPrefix("mon", "http://bla/bla#");
2 model2.setNsPrefix("", "foo://bla/bla#");
3
4 model1.write(System.out, "TURTLE");
5 model2.write(System.out, "RDF/XML-ABBREV");
6
7 Model mUnion = model1.union(model2);
8 Model mIntersection = model1.intersection(model2);
9 Model mDifference = model1.difference(model2);
```

Codice 4.6: Operazioni sui modelli RDF

Sitografia

[Bio16] Stanford Center for Biomedical Informatics Research. Protégé. 2016. URL: https://protege.stanford.edu.

[Cam24] Università di Camerino. Visual Paradigm Enterprise. 2024. URL: https://ap.visual-paradigm.com/universita-di-camerino.

[Dat07] Insight Centre for Data Analytics. The Linked Open Data Cloud. 2007. URL: https://lod-cloud.net.

[Fou11a] The Apache Software Foundation. Apache Jena. 2011. URL: https://jena.apache.org.

[Fou11b] The Apache Software Foundation. Apache Jena API. 2011. URL: https://jena.apache.org/documentation/javadoc/jena/org.apache.jena.core/org/apache/jena/package-summary.html.

[Gro24] Ontology Engineering Group. Linked Open Vocabularies. 2024. URL: https://lov.okfn.org/dataset/lov.

[Jav23] Java. Java Doc. 2023. URL: https://docs.oracle.com/en/java/javase/21/docs/api/index.html.

[Tea23] The JUnit Team. JUnit 5. 2023. URL: https://junit.org/junit5.

[Vir24] OpenLink Software Virtuoso. SPARQL Query Editor. 2024. URL: https://dbpedia.org/sparql.

Bibliografia

[HKR10] Hitzler, Krotzsch, and Rudolph. Foundations of Semantic Web Technologies. 1st ed. Taylor & Francis group, 2010. ISBN: 9780429143472.