Programmazione II

Luca De Paulis

7 novembre 2020

INDICE

Parte I

OBJECT-ORIENTED PROGRAMMING

1 OOP IN JAVA

1.1 CLASSI E OGGETTI IN JAVA

Java, come tutti i linguaggi Object-Oriented, si basa fortemente sulle nozioni di *classe* e di *oggetto*.

- Un oggetto è un insieme strutturato di *variabili di istanza*, che rappresentano lo stato interno dell'oggetto, e di *metodi*, che rappresentano le azioni che possiamo compiere sull'oggetto.
- Una classe, invece, è un *template* che possiamo usare per creare oggetti di un determinato tipo: la definizione di una classe specifica
  - tipo e valori iniziali dello stato locale degli oggetti;
  - l'insieme delle operazioni che possono essere eseguite su oggetti che sono istanze di quella classe.

  In particolare ogni definizione di classe implementa uno o più metodi costruttore: essi servono a "costruire" un oggetto di quella determinata classe.

Per implementare il concetto di *information hiding* gli oggetti in Java nascondono all'esterno il loro stato locale, ma hanno un'interfaccia ben definita, data dai loro metodi pubblici, che consentono di agire sull'oggetto solo nei modi concessi dal programmatore.
Gli oggetti sono caratterizzati da:

- uno stato interno
- un nome che individua ogni oggetto
- un ciclo di vita (creazione, riferimento, disattivazione)
- una locazione di memoria
- dei comportamenti, dati dai metodi.

Il meccanismo utilizzato per gli assegnamenti tra oggetti è il cosiddetto *sharing strutturale*: l'assegnamento `obj1 = obj2` (dove `obj1` e `obj2` sono due oggetti) fa in modo che `obj1` sia un *riferimento* all'oggetto riferito da `obj2`. Non viene quindi creata una copia di `obj2`, ma ora due variabili si riferiscono alla stessa locazione di memoria: questo concetto va sotto il nome di *aliasing*.

1.1.1 Definizione di classe

Vediamo ora come si definisce una classe in Java.

```
public class Point {
    private int x, y;

    public Point(int x0, int y0){
        x = x0;
        y = y0;
    }

    public void add(Point p){
```

```
            x += p.x;
            y += p.y;
        }

        public Point sum(Point p){
            Point res = new Point(x + p.x, y + p.y);
            return res;
        }

        public String toString(){
            return "("+ x + ", " + y + ")";
        }
    }
```

- `public` e `private` sono dei *modificatori*: `public` si usa per dire che un metodo, una variabile di istanza o la dichiarazione di una classe devono essere visibili al codice esterno, mentre `private` si usa per nascondere la dichiarazione al resto del codice.

  Generalmente le variabili di istanza devono essere nascoste (per rispettare il principio dell'information hiding), mentre i metodi sono pubblici in modo da consentire lo scambio di dati tra le istanze di una classe e l'esterno.

- La parola chiave `class`, seguita da un nome, è l'inizio della dichiarazione della classe.

- Nella prima parte si dichiarano le *variabili di istanza* dela classe: in questo caso `x` e `y` sono dichiarate private in quanto vogliamo che siano visibili e modificabili solo dalla classe.

- Il primo metodo dichiarato è il *costruttore della classe*: esso ha lo stesso nome della classe e ci consente di creare nuovi oggetti di tipo `Point` inizializzando le variabili di istanza.

- Seguono poi dei *metodi*: essi permettono al resto del codice di operare con oggetti di tipo `Point`, ad esempio sommandoli tra loro o stampandoli a schermo tramite il metodo `toString`.

Un programma Java è mandato in esecuzione invocando un metodo speciale, detto *main*, con la seguente firma:

```
    public static void main(String[] args)
```

Un esempio potrebbe essere il seguente:

```
    public class Main {
    public static void main(String args[]){
        Point a = new Point(1, 0);
        Point b = new Point(0, 2);
        Point c = a.sum(b);
        System.out.println(a.toString() + " + " + b + " = " + c);
    }
}
```

Il metodo `main` crea due oggetti di tipo `Point` tramite la parola chiave `new`: essa manda in esecuzione il costruttore della classe, creando due oggetti che rappresentano rispettivamente il punto $(1,0)$ e il punto $(1,2)$. Successivamente viene invocato il metodo `sum` dell'oggetto `a`: questo metodo restituisce un nuovo oggetto di tipo `Point` che contiene il punto dato dalla "somma" dei punti `a` e `b`. Infine il metodo stampa i tre punti sfruttando il metodo `toString`.

1.2 ABSTRACT STACK MACHINE

Per descrivere il funzionamento di Java sfrutteremo un modello computazionale chiamato *Abstract Stack Machine*: essa ci consente di descrivere i cambiamenti dello stato e le interazioni tra gli oggetti.

La ASM è formata da tre componenti:

- un *workspace*, dove sono memorizzati i programmi in esecuzione e quindi le istruzioni da eseguire;
- uno *stack*, che viene usato per gestire i binding tra variabili e locazioni di memoria che contengono gli oggetti;
- uno *heap* che contiene le locazioni di memoria degli oggetti e viene usato per la gestione della memoria dinamica.

Supponiamo ad esempio di avere una dichiarazione di classe di questo tipo:

```
public class Node{
    private int elt;
    private Node next;

    public Node(int e0, Node n0){
        elt = e0;
        next = n0;
    }
    ...
}
```

Se creiamo un oggetto di tipo `Node`:

```
Node first = new Node(5, null);
```

la ASM si modificherà nel seguente modo:

- nello stack comparirà un nuovo binding: il nome di variabile `first` sarà un riferimento ad una locazione di memoria sullo heap, in cui sarà contenuto un nuovo oggetto di tipo `Node`;
- nello heap verrà allocato lo spazio per il nuovo oggetto: in particolare vi sarà lo spazio per le variabili di istanza (inizializzate a 5 e a `null` rispettivamente) e dello spazio aggiuntivo che analizzeremo in seguito.

È importante rendersi conto che i riferimenti di Java sono simili ai puntatori definiti in C/C++, ma non consentono l'uso dell'aritmetica dei puntatori: sono semplicemente *riferimenti* agli oggetti allocati sullo heap.

1.3 INTERFACCE

Abbiamo visto che un meccanismo per definire nuovi tipi in Java è il meccanismo delle classi. Tuttavia Java definisce anche altri costrutti sintattici per realizzare nuovi tipi di dati, come ad esempio le *interfacce*.

Un'interfaccia serve a definire il tipo di un oggetto in modo dichiarativo: vengono dichiarati i metodi che gli oggetti di quel tipo devono implementare, ma non viene definita la loro implementazione.

Ad esempio, consideriamo gli oggetti che vengono rappresentati graficamente su uno schermo bidimensionale: ognuno di essi ha una posizione rispetto all'asse delle ascisse e delle ordinate; inoltre ognuno di essi può essere spostato sullo schermo muovendo le sue coordinate in un altro punto dello schermo. Per rappresentare ciò in Java dichiariamo un'interfaccia:

```
public interface Displaceable{
    public int getX();
    public int getY();
    public void mode(int dx, int dy);
}
```

Questa interfaccia dichiara tre metodi:

- il metodo `getX`, che (quando implementato) dovrà restituire la posizione sulle ascisse del nostro oggetto;
- il metodo `getY`, che (quando implementato) dovrà restituire la posizione sulle ordinate del nostro oggetto;
- il metodo `mode`, che prende in input due interi (`dx` e `dy`) e avrà lo scopo di spostare l'oggetto dalla posizione corrente, aggiungendo `dx` alle ascisse e `dy` alle ordinate.

Le interfacce sono utili in quanto possono essere *implementate dalle classi*:

```
public class Point implements Displaceable {
    private int x, y;

    public Point(int x0, int y0){
        x = x0;
        y = y0;
    }

    public int getX(){
        return x;
    }

    public int getY(){
        return y;
    }

    public void move(int dx, int dy){
        x += dx;
        y += dy;
    }
}
```

La classe `Point` implementa i metodi definiti dall'interfaccia `Displaceable`:

- la parola chiave **`implements`** serve a specificare che la classe sta implementando un'interfaccia;
- ogni classe può implementare più di un interfaccia;
- quando una classe implementa un'interfaccia *deve* fornire un'implementazione di ogni metodo definito nell'interfaccia.

Ovviamente una classe può implementare più metodi di quelli definiti da un'interfaccia, come si può vedere nel prossimo esempio:

```
public class ColorPoint implements Displaceable {
    private Point p;
    private Color c;

    public ColorPoint(int x0, int y0, Color c0){
        p = new Point(x0, y0);
        c = c0;
```

```
        }

        public int getX(){
            return p.getX();
        }

        public int getY(){
            return p.getY();
        }

        public Color getColor(){
            return c;
        }

        public void move(int dx, int dy){
            p.move(dx, dy);
        }
    }
```

La classe `ColorPoint` ha più metodi di quelli definiti nell'interfaccia `Displaceable`; inoltre notiamo che essa delega il compito di restituire la posizione sulle ascisse/ordinate e il compito di spostare il punto alla classe `Point`: ciò è una buona pratica di programmazione in quanto consente il riuso del codice. Infatti se volessimo estendere le funzionalità di `mode` in questo caso possiamo semplicemente modificare l'implementazione contenuta nella classe `Point`: la classe `ColorPoint` sarebbe automaticamente aggiornata e non dovremmo cambiare anche il suo codice.

1.3.1 Interfacce come tipi di variabili

Possiamo usare le interfacce per dichiarare nuove variabili:

```
Displaceable d;
d = new Point(3, 1);
d.move(1, -1);
d = new ColorPoint(0, 0, new Color("white"));
```

La variabile `d` è di tipo `Displaceable`: possiamo assegnare ad essa una qualsiasi istanza di una classe che implementa `Displaceable`, come ad esempio `Point` oppure `ColorPoint`.

Questo fenomeno è chiamato *sub-typing*: la variabile `d` è di tipo `Displaceable`, dunque può essere legata ad un qualsiasi oggetto che implementi l'interfaccia `Displaceable`. Più in generale, un tipo `A` è detto *sottotipo* di un altro tipo `B` se `A` soddisfa tutti gli obblighi richiesti da `B`.

1.3.2 Interfacce multiple

Come abbiamo detto prima, un oggetto può implementare più di un'interfaccia. Ad esempio dichiariamo questa nuova interfaccia:

```
public interface Area{
    public void getArea();
}
```

Una classe che implementa sia `Displaceable` che `Area` può essere la seguente:

```
public class Circle implements Area, Displaceable{
    private Point center;
```

```
    private int rad;

    public Circle(int x0, int y0, int r0){
        center = new Point(x0, y0);
        rad = r0;
    }

    public double getArea(){
        return Math.pi * rad * rad;
    }

    public int getX(){
        return center.getX();
    }

    public int getY(){
        return center.getY();
    }

    public void move(int dx, int dy){
        center.move(dx, dy);
    }
}
```

1.4 EREDITARIETÀ

Consideriamo il seguente codice:

```
public class PaintedCircle {
    private Point center;
    private int radius;
    private Color fillColor;
    private Color borderColor;
    private double borderThickness;

    public void fillWith(Color c){
        fillColor = c;
    }

    public void setBorderThickness(double t){
        borderThickness = t;
    }

    public void setBorderColor(Color c){
        borderColor = c;
    }
    ...
}

public class PaintedTriangle {
    private Point v1, v2, v3;
    private Color fillColor;
    private Color borderColor;
    private double borderThickness;

    public void fillWith(Color c){
        fillColor = c;
    }

    public void setBorderThickness(double t){
```

```
            borderThickness = t;
        }

        public void setBorderColor(Color c){
            borderColor = c;
        }
        ...
    }
```

Il codice delle due classi è corretto e funziona, ma non è buon codice: se volessimo modificare il funzionamento dei bordi, ad esempio aggiungendo la possibilità di avere bordi tratteggiati, dovremmo modificarlo per ognuna delle due classi `PaintedCircle` e `PaintedTriangle`. Per questo si ricorre al meccanismo dell'*ereditarietà tra classi*.

L'ereditarietà ci consente di

- riusare il codice e creare una gerarchia tra le classi, in modo che
  - le classi in alto nella gerarchia rappresentino tipi più generalizzati/astratti;
  - le classi in basso nella gerarchia rappresentino tipi più specifici/concreti;
- sfruttare il meccanismo del sub-typing per usare tipi più specifici dove sono previsti i loro supertipi.

Il secondo punto viene generalmente chiamato *principio di sostituzione*: se `B` è un sottotipo di `A`, allora gli oggetti di tipo `B` possono essere usati dove sono previsti oggetti di tipo `A`. In particolare

- un'istanza del sottotipo (`B`) soddisfa sempre le proprietà del supertipo (`A`);
- un'istanza del sottotipo può avere più vincoli del supertipo.

1.4.1 Ereditarietà in Java

L'ereditarietà tra due classi `B` e `A` viene realizzata in Java tramite la parola chiave **`extends`**:

```
class B extends A {...}
```

Al contrario delle interfacce, una classe in Java può estendere una sola classe: questo meccanismo è chiamato *ereditarietà singola* (al contrario dell'ereditarietà multipla, come ad esempio accade in C++).

Facciamo un esempio di ereditarietà:

```
public class D {
    private int x, y;

    public int addBoth(){
        return x + y;
    }
}

public class C extends D {
    private int z;

    public int addThree(){
        return addBoth() + z;
    }
}
```

Notiamo che:

- la classe `C` eredita tutte le variabili di istanza di `D` implicitamente; tuttavia, dato che le variabili di istanza di `D` sono dichiarate private, gli oggetti di tipo `C` non possono accedervi (anche se sono presenti!);
- la classe `C` eredita tutti i metodi di `D` e quindi (se sono dichiarati **`public`**) può usarli all'interno dei suoi metodi.

MODIFICATORE `protected` Per fare in modo che le variabili di istanza siano nascoste all'esterno ma visibili alle sottoclassi il Java mette a disposizione la parola chiave **`protected`**: se dichiarassimo protette (invece che private) le variabili della classe `D` riusciremmo ad accedervi e a modificarle dalla classe `C`.

La scelta **`private`**/**`protected`** dipende dalla situazione e non vi sono regole generali per capire quando è meglio usare l'una oppure l'altra.

IL METODO `super` Il metodo costruttore della classe padre non viene ereditato direttamente: per chiamarlo è necessario usare il metodo **`super`**, come mostrato dal seguente esempio:

```
public class D {
    private int x, y;

    public D(int initX, int initY){
        x = initX;
        y = initY;
    }
    ...
}

public class C extends D {
    private int z;

    public C(int initX, int initY, int initZ){
        super(initX, initY);
        z = initZ;
    }
}
```

THIS La parola chiave **`this`** ci permette di riferirci all'oggetto corrente. Anche se solitamente possiamo usare i metodi e le variabili dell'oggetto corrente senza farli precedere dal nome dell'oggetto e dall'operatore punto, in alcuni casi può essere utile.

Uno di questi casi è quando vogliamo disambiguare i nomi delle variabili di istanza con i nomi delle variabili in ingresso al metodo:

```
public class Point {
    private int x, y;
    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

Nel metodo costruttore `x` e `y` si riferiscono ai parametri formali del metodo, mentre **`this`**`.x` e **`this`**`.y` si riferiscono alle variabili di istanza dell'oggetto.

Un altro uso per **`this`** è come costruttore implicito:

```
public class Rectangle{
    private int x, y;
    private int width, height;

    public Rectangle(int x0, int y0, int w, int h){
        x = x0;
        y = y0;
        width = w;
        height = h;
    }

    public Rectangle(int w, int h){
        this(0, 0, w, h);
    }

    public Rectangle(){
        this(0, 0, 0, 0);
    }
}
```

In questo caso la classe `Rectangle` ha tre diversi costruttori: il primo prende 4 parametri (posizione sull'asse x, sull'asse y, larghezza e altezza), il secondo ne prende solo 2 (larghezza e altezza) e infine il terzo non prende parametri. Il costruttore implicito `this` usa il codice del costruttore con 4 parametri per definire i costruttori con 2 e 0 parametri, ponendo a 0 i termini non specificati.

Upcasting e downcasting

L'ereditarietà in Java ci consente di usare oggetti sottotipo al posto di oggetti supertipo, in modo da avere codice più modulare. Possiamo inoltre trasformare un oggetto di un supertipo in un oggetto di un sottotipo e viceversa, tramite l'*upcasting* e il *downcasting*.

L'upcasting serve a trasformare una variabile di un sottotipo in una variabile di un supertipo:

```
public class Vehicle {...}
public class Car extends Vehicle {...}

...
Vehicle v = (Vehicle) new Car();
```

Invece il downcasting serve per trasformare una variabile di un supertipo e specializzarla in un sottotipo:

```
Car c = (Car) new Vehicle();
```

L'upcasting può essere anche implicito, mentre il downcasting va sempre esplicitato e può avvenire solo tra una classe e una sua super-classe.

Ogni classe estende `Object`

Esiste una classe in Java che fa da "antenato" a tutte le altre classi: la classe `Object`. Infatti tutte le classi estendono `Object` implicitamente, poiché in `Object` sono definiti alcuni metodi che devono essere comuni a tutti gli oggetti.

METODO `equals` Il metodo `equals` viene usato per controllare la *deep equality* tra due oggetti: si usa `equals` quando si vuole verificare non se due variabili sono riferimenti allo stesso oggetto (*shallow equality*), ma quando si vuole verificare che due oggetti (che occupano due posizioni in memoria diverse) hanno lo stesso *stato interno*.

Siccome il concetto di *deep equality* dipende dal tipo di oggetto che stiamo considerando, il metodo `equals` va ridefinito per ogni nuova classe: la definizione di default controlla solo la *shallow equality* tra due oggetti (tramite l'operatore ==).

METODO `tostring` Il metodo `toString` permette di rappresentare lo stato interno di un oggetto sotto forma di stringa. Anche in questo caso il metodo va ridefinito per ogni classe.

METODO `clone` Il metodo `clone` permette di creare un nuovo oggetto con lo stesso stato interno dell'oggetto corrente (anche se in alcuni casi può creare una situazione di condivisione di dati). Questo metodo viene ereditato solo se la classe implementa l'interfaccia `Cloneable`.

1.5 TIPI DINAMICI E STATICI

Consideriamo il seguente esempio:

```
public class Vehicle {...}
public class Car extends Vehicle {...}

...
Vehicle v = (Vehicle) new Car();
```

Qual è il tipo di `v`?

- Da una parte `v` è una variabile di tipo `Vehicle`.
- D'altro canto l'oggetto il cui riferimento è contenuto in `v` è di tipo `Car`, un sottotipo di `Vehicle`.

Definiamo quindi due *tipi* associati ad ogni variabile:

- Il *tipo statico*, che è dato da tutte le informazioni di carattere sintattico (in particolare, dal tipo della variabile). Nel nostro caso il tipo statico di `v` è `Vehicle`.
- Il *tipo dinamico* che rappresenta il tipo dell'oggetto istanziato sullo heap riferito dalla variabile in questione. Nel nostro caso, il tipo dinamico è `Car`.

Facciamo un esempio:

```
public class Father {
    ...
    public void printA(){
        System.out.println("A");
    }
    public void printB(){
        System.out.println("B");
    }
}
public class Son extends Father {
    ...
```

```
        public void printB(){
            System.out.println("B from Son");
        }
        public void printC(){
            System.out.println("C");
        }
    }

    ...
    Father obj = new Son();
    obj.printA();
    obj.printB();
    obj.printC();
```

Il metodo `printA()` stampa "A" a schermo, come dovrebbe. I metodi successivi si comportano in maniera più particolare:

- Il metodo `printB()` stampa "B from Son": siccome il tipo dinamico della variabile `obj` è `Son`, dunque il metodo `printB()` usa l'implementazione fornita dalla classe `Son` e non dalla classe padre.
- Il metodo `printC()` dà un errore di compilazione: siccome il tipo statico della variabile `obj` è `Father` il compilatore non riconosce il metodo `printC()` e quindi dà errore, anche se il tipo dinamico lo consentirebbe.

Il motivo di questo comportamento è il seguente: siccome una variabile di tipo `Father` può contenere un riferimento ad oggetti di un qualunque suo sottotipo, a priori gli unici metodi che siamo certi che essa può eseguire sono quelli definiti da `Father`. Infatti tutti i metodi della classe padre devono essere ereditati o ridefiniti dalle classi che ereditano da essa, mentre i metodi definiti dai sottotipi non sono sempre accessibili (come nel caso di `printC()`).

Invece il punto dei sottotipi è quello di specializzare le classi supertipo, dunque se un metodo della classe supertipo viene ridefinito, la versione invocata sarà sempre la più specializzata possibile (come nel caso di `printB()`).

1.6 ECCEZIONI IN JAVA

In alcuni casi vogliamo segnalare il fatto che un metodo non può restituire il risultato corretto perché si è verificato un errore (ad esempio, si vuole leggere un file che non esiste). Il Java fornisce un potente strumento sintattico per gestire questi casi: le cosiddette *eccezioni*.

Le eccezioni sono particolari oggetti che servono a segnalare una situazione anomala: esse sono uno strumento più potente di scegliere "valori particolari" da restituire in caso di errore in quanto

- possiamo avere eccezioni specializzate per ogni tipo di errore;
- c'è *separations of concerns*: abbiamo un costrutto sintattico che si occupa di gestire gli errori.

Per sollevare un'eccezione bisogna usare la parola chiave `throw`:

```
    throw new MyException();
```

Esso richiede come argomento un qualunque oggetto che sia un sottotipo della classe `Throwable`. Questa classe ha una famiglia di sottoclassi molto ramificata, su cui torneremo più avanti per una distinzione particolare.

Gestione delle eccezioni

Per gestire le eccezioni il Java mette a disposizione il costrutto `try - catch - finally`.

- Nella clausola `try` si inseriscono le istruzioni che potrebbero sollevare un'eccezione. Ad esempio

```
try {
    int[] arr = new int[3];
    System.out.println(arr[4].toString);
}
```

  Siccome l'array ha solo 3 elementi, tentando di accedere alla posizione di indice 4 viene sollevata una `IndexOutOfBoundsException`, che sarà catturata da un eventuale blocco `catch`.

- Nelle clausole `catch` si specifica un'eccezione che può essere sollevata dal codice contenuto nella `try`: nel caso questa eccezione venisse sollevata, il codice all'interno della specifica clausola `catch` viene eseguito. Ad esempio:

```
try {
    int[] arr = new int[3];
    System.out.println(arr[4].toString);
} catch (ArrayOutOfBoundsException e) {
    System.out.println(e);
} catch (NullPointerException e) {
    System.out.println(e);
}
```

  In questo caso, dopo che `IndexOutOfBoundsException` viene sollevata, essa viene catturata dal primo blocco `catch` che quindi esegue il suo codice e fa proseguire l'esecuzione del programma alla fine del blocco `try - catch`. Se l'eccezione sollevata non compare in nessuno dei blocchi `catch`, l'esecuzione del metodo si interrompe con un fallimento e l'eccezione viene passata al metodo chiamante.

- Il blocco `finally` viene eseguito alla fine dell'esecuzione `try - catch`, in *qualsiasi* caso. Infatti anche se nessuno dei blocchi `catch` riesce a catturare l'eccezione sollevata nel blocco `try`, il metodo esegue il blocco `finally` prima di restituire il controllo al chiamante. La clauolsa `finally` è utile per eseguire del codice di *clean-up* a prescindere dal fatto che sia stata sollevata un'eccezione o meno.

Osserviamo che il blocco `catch` non può catturare solamente l'eccezione che viene specificata come argomento, ma anche tutti i suoi sottotipi: ad esempio in

```
try {
    int[] arr = new int[3];
    System.out.println(arr[4].toString);
} catch (Throwable e) {
    System.out.println(e);
}
```

la `catch` cattura qualsiasi tipo di eccezione, in quanto tutte sono sottotipo di `throwable`.

Inoltre possiamo avere più blocchi `try - catch` annidati.

```
try {
    try {
```

```
            x = Array.searchSorted(v, y);
        } catch (NullPointerException e) {
            throw new NotFoundException();
        }
    } catch (NotFoundException e) {
        System.out.println(e);
    }
```

In questo caso il blocco `catch` esterno può catturare sia la `NotFoundException` sollevata dal blocco `catch` interno, sia una possibile `NotFoundException` sollevata dal metodo `Array.searchSorted`.

1.6.1 Eccezioni checked e unchecked

Come abbiamo detto prima la classe `Throwable` ha diverse sottoclassi. Esse si dividono in due categorie: le classi che generano eccezioni *checked* e quelle che generano eccezioni *unchecked*.

ECCEZIONI *CHECKED* Le eccezioni *checked* sono tutte le eccezioni che sono sottotipo della classe `Exception`. Esse sono delle eccezioni particolari in quanto devono essere elencate nelle firme dei metodi che possono sollevarle, come in

```
public void myMethod() throws MyCheckedException { ... }
```

Inoltre le eccezioni checked non possono essere propagate, ma devono essere gestite (tramite `try - catch`) appena vengono sollevate (a meno che il metodo corrente non le dichiari nella sua firma).

Queste condizioni vengono controllate dal compilatore, per cui le eccezioni checked sono le eccezioni che non "distruggono il flusso del programma": basta catturare l'eccezione e si può continuare con l'esecuzione.

ECCEZIONI *UNCHECKED* Un'eccezione *unchecked* è un'eccezione che estende `RuntimeException`: esse rappresentano tutti gli errori che possono accadere a runtime, come ad esempio la possibilità di avere un riferimento a `null` (codificato dalla `NullPointerException`), oppure di essere andati fuori dagli indici permessi in un array (errore codificato dalla `IndexOutOfBoundsException`), o tante altre.

Al contrario delle eccezioni checked, le unchecked non devono necessariamente essere enumerate nella firma di un metodo (anche se è buona pratica farlo ugualmente), né devono essere obbligatoriamente catturate da una specifica clausola `try - catch`.

Le eccezioni checked sono più sicure e robuste delle unchecked, poiché il compilatore si assicura che vengano elencate e gestite; tuttavia quando siamo ragionevolmente sicuri che l'eccezione non verrà sollevata può essere più utile usare un'eccezione unchecked. Al contempo, le eccezioni unchecked possono essere difficili da notare, in quanto non vengono dichiarate esplicitamente e il programmatore non sa quale metodo le ha sollevate: in questo caso l'unica cosa da fare è separare il blocco `try` in più blocchi, in modo da sapere effettivamente quale metodo solleva la specifica eccezione.

1.6.2 Definire nuove eccezioni

Per definire una nuova eccezione bisogna innanzitutto decidere se la si vuole definire checked o unchecked. Nel primo caso, essa deve essere un sottotipo di `Exception`, e deve contenere al suo intero solamente uno o più costruttori:

```
public MyCheckedException extends Exception {
    public MyCheckedException(){
        super();
    }

    public MyCheckedException(String e){
        super(e);
    }
}
```

Il discorso è identico per eccezioni unchecked, con la differenza che la classe deve estendere `RuntimeException`:

```
public MyUnCheckedException extends RuntimeException {
    public MyUnCheckedException(){
        super();
    }

    public MyUnCheckedException(String e){
        super(e);
    }
}
```

1.6.3 Introduzione alla Programmazione Difensiva

Lo stile di programmazione che usa le eccezioni (insieme, come vedremo nel prossimo capitolo, ai contratti d'uso) per evitare situazioni anomale viene chiamato *defensive programming* o programmazione difensiva. In questo stile di programmazione il programmatore descrive quali sono gli input ammessi per ogni metodo definito, e in caso di input non ammessi solleva un'eccezione per segnalare all'utente che il metodo non può svolgere il suo compito correttamente. Parleremo più approfonditamente di defensive programming più avanti.

2 | ADT E SPECIFICHE

2.1 DESIGN-BY-CONTRACT

Il concetto di *design-by-contract* è stato introdotto per permettere di costruire progetti robusti e facilmente estendibili. Secondo questo modello di progettazione è necessario creare una *barriera di astrazione* tra colui che progetta il software e il cliente tramite un *contratto d'uso*.

Il contratto è formato da due parti:

- una *precondizione* (clausola *requires*), che serve a colui che progetta il software per dire quali devono essere i vincoli sui dati prima della chiamata di un metodo;
- una *postcondizione* (clausola *effects*), che indica l'effetto del metodo nei casi in cui la precondizione è soddisfatta: il metodo deve sottostare precisamente alla postcondizione, ad esempio modificando le giuste variabili, oppure sollevando le eccezioni specificate, eccetera.

Facciamo un esempio:

```
// @requires: num >= 0
// @effects: returns the square root of the number num
public static double sqrt(double num);
```

In questo caso la precondizione è che il parametro in ingresso `num` sia non-negativo (se ciò non fosse l'operazione di radice quadrata non avrebbe senso); la postcondizione invece è che il metodo (se la precondizione è verificata) restituisce effettivamente la radice quadrata del numero.

Il contratto d'uso serve ad astrarre l'uso di un metodo dalla sua implementazione: non serve conoscere il codice sorgente di un metodo per poterlo usare. Nel caso del metodo `sqrt` non importa sapere *in che modo* esso calcola la radice quadrata: il contratto d'uso ci dice che il parametro in ingresso deve essere non-negativo e ci assicura che in questo caso il risultato sia davvero la radice quadrata del numero, a prescindere da quale sia l'algoritmo usato per calcolarla.

Questo concetto si rivela molto utile quando si vogliono fare modifiche incrementali ai metodi: se volessimo migliorare l'implementazione di un metodo non dobbiamo preoccuparci di come esso viene usato, ma dobbiamo assicurarci solamente di mantenere inalterate la precondizione e la postcondizione. In questo modo il cliente non vede alcuna differenza nell'uso del metodo precedente con quello nuovo e non possono esserci errori dovuti all'implementazione di nuove funzionalità, in quanto tutte le richieste e tutte le funzionalità vanno inserite nelle clausole *requires* e *effects*.

2.2 DEFENSIVE PROGRAMMING

Le precondizioni e le postcondizioni viste nella sezione precedente possono essere pensate come formule logiche della Logica del Primo Ordine. La relazione tra le due (ovvero che se vale la precondizione allora dobbiamo assicurarci che, alla fine del metodo, valga la postcondizione) può essere realizzata tramite una semplice implicazione:

$$\text{Pre} \implies \text{Post}.$$

Tuttavia un'implicazione è sempre vera quando l'antecedente (in questo caso `Pre`) è falso, dunque il contratto ci permette di fare qualsiasi cosa quando l'antecedente è falso. Ad esempio il seguente programma rispetta il contratto d'uso dato:

```
// @requires: num ⩾ 0
// @effects: returns the 4th root of the number num
public static double fourthRoot(double num) {
    if(num < 0) {       // anything's allowed
        return 254;
    } else {
        return sqrt(sqrt(num));
    }
}
```

Infatti, siccome abbiamo specificato chiaramente che il programma funziona correttamente solamente quando `num` è non-negativo, ci aspettiamo di non trovarci mai nel primo caso, per cui potremmo direttamente eliminare il costrutto `if`.

Tuttavia una buona pratica di programmazione è controllare due volte la precondizione: la prima tramite la clausola requires, la seconda attraverso l'uso delle eccezioni.

```
// @requires: num ⩾ 0
// @effects: returns the 4th root of the number num
public static double fourthRoot(double num)
            throws IllegalArgumentException {
    if(num < 0) {
        throw new IllegalArgumentException();
    } else {
        return sqrt(sqrt(num));
    }
}
```

In questo caso siamo certi che il programma non può comportarsi in modi diversi da come lo vogliamo poiché, oltre ad aver esplicitato le precondizioni, abbiamo anche sollevato un'eccezione adatta in caso il metodo fosse stato chiamato con i parametri errati.

Questo stile di programmazione si chiama *programmazione difensiva* (oppure *defensive programming*), e serve a creare programmi facili da usare, facili da mantenere e al contempo sicuri (poiché blocchiamo qualsiasi possibile input che non rispetta le precondizioni tramite le eccezioni).

2.3 ABSTRACT DATA TYPES

Abbiamo visto che il concetto di specifica ci permette di astrarre l'uso di un metodo dalla sua implementazione: possiamo fare la stessa cosa con le strutture dati, che sono la base di ogni progetto. Infatti può capitare spesso che una scelta di struttura dati fatta ad inizio progetto possa rivelarsi non ottimale in seguito a causa della sua implementazione: vorremmo quindi un modo per astrarre dall'organizzazione e dal significato specifico dei dati e pensare solo in termini delle operazioni fornite.

Un esempio di tipo di dato astratto può essere una classe in Java da un punto di vista esterno: per operare sulla classe possiamo solamente usare i metodi che ci vengono forniti dalla classe, insieme alle loro specifiche; se essi in futuro dovessero essere migliorati (ad esempio dal punto di vista della performance) il cliente non avrà modo di rendersene conto.

I metodi di un ADT devono essere nascosti all'utente se non per la loro specifica: ognuno di essi dovrà avere delle precondizioni e delle postcondizioni che permettono a chi usa il metodo di sfruttarlo anche senza conoscerne l'implementazione. I metodi di un ADT possono essere divisi in 4 categorie a seconda del loro scopo:

- i *creators*, che servono a creare nuove istanze dell'ADT (in Java sono i costruttori);
- i *producers*, che sono metodi che restituiscono nuovi dati;
- i *mutators*, che modificano i dati esistenti;
- gli *observers*, che servono a dare informazioni relative allo stato interno degli oggetti, facendo attenzione a non violare la barriera di rappresentazione.

Infine gli ADT possono essere divisi in due gruppi: quelli *mutable*, che permettono operazioni che ne modifichino lo stato interno, e quelli *immutable*, che non hanno *mutators*.

Facciamo due esempi.

2.3.1 ADT Poly

CLAUSOLE OVERVIEW E TYPICAL OBJECT Supponiamo di voler creare un ADT che rappresenti un polinomio. Il primo passo è specificare *cosa rappresenta* il nostro ADT e qual è un elemento tipico di questo tipo di dato astratto.

```
/**
 * A Poly is an immutable polynomial with integer coefficient.
 * A typical element is
 *          c_0 + c_1 x + c_2 x^2 + ...
 */
public class Poly { ...
```

La clausola *overview* stabilisce se il tipo di dato astratto è mutable o immutable e ne definisce il modello astratto tramite il *typical element*.

CREATORS I metodi *creators* servono a creare una nuova istanza del tipo di dato astratto.

```
/**
 * @effects: makes a new Poly = 0
 */
public Poly();

/**
 * @throws: NegExponentException if n < 0
 * @effects: makes a new Poly = cx^n
 */
public Poly(int c, int n);
```

Siccome i creators servono a creare nuovi oggetti, l'unica clausola indispensabile è la clausola effects.

OBSERVERS I metodi *observers* servono a reperire parte dell'informazione contenuta nello stato interno di un'istanza di un ADT.

```
/**
 * @returns: the degree of this,
```

```
     *      the greatest exponent with a
     *      non-zero coefficient.
     *      Returns 0 if this = 0.
     */
    public int degree() { }

    /**
     * @throws: NegExponentException if n < 0
     * @returns: the coefficient of the term
     *      of this whose exponent is n.
     */
    public int coeff(int n) { }
```

È importante che gli observers non modifichino lo stato astratto, né violino la barriera di astrazione esponendo dati all'esterno, ad esempio restituendo il riferimento ad un oggetto contenuto nello stato interno. Inoltre notiamo che nella descrizione degli observers si usa sempre la rappresentazione astratta dell'oggetto fornita tramite la clausola overview.

PRODUCERS I *producers* servono a produrre nuovi oggetti del tipo dell'ADT a partire da oggetti già esistenti. Essi non devono avere effetti laterali, ovvero non devono modificare lo stato astratto degli oggetti su cui sono chiamati, ma devono solamente produrre nuovi dati.

```
    /**
     * @returns: this + p
     */
    public Poly add(Poly p) { }

    /**
     * @returns: this * p
     */
    public Poly mult(Poly p) { }
```

2.4 IMPLEMENTARE ADT

Abbiamo visto come la specifica e i concetti del design-by-contract ci consentano progettare tipi di dati robusti. Per implementare un tipo di dato astratto abbiamo bisogno di altri due importanti strumenti: l'*invariante di rappresentazione* e la *funzione di astrazione*.

INVARIANTE DI RAPPRESENTAZIONE L'invariante di rappresentazione (abbreviato ad RI, per *rapresentation invariant*) è una funzione che prende un oggetto e ritorna un valore di verità (vero o falso): esso serve per stabilire se l'istanza considerata è ben formata, ovvero rispetta delle condizioni particolari. L'invariante di rappresentazione è una guida per chi implementa, modifica o verifica il funzionamento del tipo di dato astratto: nessun oggetto deve violare l'invariante di rappresentazione.

Ad esempio, se volessimo codificare una struttura dati che rappresenta un insieme matematico, potremmo richiedere che non vi siano mai due valori uguali nell'insieme: l'invariante di rappresentazione conterrà questa clausola e quindi le istanze ben formate della classe saranno soltanto quelle che non contengono duplicati.

FUNZIONE DI ASTRAZIONE La funzione di astrazione (o AF, dall'inglese *abstraction function*) è una funzione che serve ad interpretare astrattamente un'istanza del tipo di dato astratto: essa è definita solo sugli oggetti che

rispettano l'invariante di rappresentazione e ci permette di pensare agli oggetti come se fossero la loro rappresentazione astratta.

Ad esempio, nel caso degli insiemi la funzione di astrazione prende un oggetto della classe `Set` e restituisce l'insieme $\{a_1, a_2, \ldots, a_n\}$ che contiene tutti i dati dell'oggetto originale.

La funzione di astrazione ci consente di pensare alle varie operazioni eseguibili sulla classe `Set` come se fossero effettivamente insiemi matematici.

Facciamo un esempio.

```
public interface CharSet{
    // Overview: CharSet is a finite and
    // modifiable set of characters.
    // A typical element is
    //          {c1, ..., cn}.

    //@effects: creates a new and empty CharSet
    public CharSet() {...}

    //@modifies: this
    //@effects: this_pre = this_post ∪ {c}
    public void insert(Character c) {...}

    //@modifies: this
    //@effects: this_pre = this_post \ {c}
    public void delete(Character c) {...}

    //@effects: returns true if and only if c is an element of this
    public void member(Character c) {...}

    //@effects: return cardinality of this
    public int size() {...}
}
```

Consideriamo ora la seguente implementazione:

```
    public class ArrayListCharSet{
        private List<Character> elems = new ArrayList<Character>();

        public void insert(Character c) { elems.add(c); }
        public void delete(Character c) {
            int i = elems.indexOf(c);
            if (i > -1) elems.remove(i);
        }
        public void member(Character c) {
            return elems.contains(c);
        }
        public int size() {
            return elems.size();
        }
    }
```

Parte II

LINGUAGGI DI PROGRAMMAZIONE

3 SINTASSI DI OCAML

3.1 VALORI ED ESPRESSIONI

OCaml è un linguaggio multiparadigma derivato dal linguaggio funzionale CaML e dai suoi antenati nella famiglia di ML. Essendo alla base un linguaggio funzionale, un programma OCaml è un'*espressione* che può essere valutata ad un *valore*, ovvero ad un'espressione che non deve essere valutata ulteriormente.

Useremo la notazione <exp> $\implies v$ oppure la notazione $\text{Eval}(\text{exp}) = v$ per dire che l'espressione exp viene valutata al valore v.

Il primo metodo per creare espressioni è il costrutto `let`. Ad esempio l'espressione

```
let x = 42;;
```

ci permette di *legare* al nome della variabile `x` il valore `42`. Questo costrutto ci consente tuttavia di definire espressioni più complesse:

```
let x = 7*3
    in x*x;;
```

Questa espressione lega alla variabile `x` il valore `21`, cioè il valore a cui l'espressione `7*3` viene valutata; inoltre, questo legame è *locale* all'espressione che segue la parola chiave `in`. Il valore dell'espressione è quindi `441`.

3.2 COSTRUTTO `let`

Il costrutto `let` ci permette inoltre di introdurre una nozione di *scope*: nel seguente codice la variabile `x` è definita e visibile dentro il `let` più esterno, mentre la variabile `y` esiste ed è visibile solo all'interno del secondo `let`.

```
let x = 42
    in x + (let y = "3110"
                in int_of_string y);;
```

Il valore di questa espressione è `3152`, in quanto il valore dell'espressione `let` interna è il numero intero `3110`, mentre il valore dell'espressione più esterna è dato dalla valutazione di `x + 3110`, che restituisce `3152`.

Regola di valutazione del costrutto `let`

Studiamo ora la regola formale di valutazione del `let`:

$$\frac{\text{Eval}(\texttt{e1}) = v' \quad \text{subst}(\texttt{e2}, \texttt{x}, v') = \texttt{e2}' \quad \text{Eval}(\texttt{e2}') = v}{\text{Eval}(\texttt{let x = e1 in e2}) = v}.$$

Questa regola ci dice che la valutazione dell'espressione `let x = e1 in e2` è v se:

- la valutazione di `e1` è un valore v';

- sostituendo il valore v' al posto di tutte le occorrenze libere di `x` nell'espressione `e2` otteniamo l'espressione `e2`$'$;
- la valutazione di `e2`$'$ è esattamente il valore v.

Esempio 3.2.1. Usiamo la regola di inferenza per calcolare il valore dell'espressione

```
let x = 2 + (5 * 3) in x + 10;;
```

(1) Valutiamo l'espressione `2 + (5 * 3)`, il cui valore è 17.
(2) Sostituiamo 17 al posto di ogni occorrenza libera di `x` nell'espressione `x + 10`, ottenendo l'espressione `17 + 10`.
(3) Valutiamo quest'ultima espressione, ottenendo 27, che è il valore dell'espressione iniziale.

Esempio 3.2.2. Usiamo la regola di inferenza per calcolare il valore dell'espressione

```
let x = 12 - 3
    in let y = 2 + x
           in x == y;;
```

(1) Valutiamo l'espressione `12 - 3`, il cui valore è 9.
(2) Sostituiamo 9 al posto di ogni occorrenza libera di `x` nell'espressione `let y = 2 + x in x == y`, ottenendo l'espressione `let y = 2 + 9 in 9 == y`.
(3) Valutiamo quest'ultima espressione: siccome anch'essa è un costrutto `let` bisogna applicare nuovamente la regola di inferenza:
    (i) Valutiamo l'espressione `2 + 9`, il cui valore è 11.
    (ii) Sostituiamo 11 al posto di ogni occorrenza libera di `y` nell'espressione `9 == y`, ottenendo l'espressione `9 == 11`.
    (iii) Valutiamo l'ultima espressione: siccome $9 \neq 11$ il risultato dell'espressione è `false`.

Segue quindi che il codice iniziale ha valore `false`.

3.3 FUNZIONI

Come in ogni linguaggio funzionale in OCaml le funzioni sono elementi del primo ordine: sono dunque espressioni, esattamente come ogni costrutto del linguaggio.

SINTASSI La sintassi per dichiarare una funzione è la seguente:

```
let <fun_name> (<par_1> : <type_1>) ... (<par_n> : type_n) :
↪  ret_type =
    <fun_body>.
```

Per fare un esempio:

```
let f (x : int) : int =
    let y = x * 10
        in y * y;;
```

Una funzione è quindi formata da

- un nome di funzione, in questo caso `f`;
- una lista di parametri, in questo caso il singolo parametro `x`. Notiamo che abbiamo esplicitato il tipo di `x` tramite la notazione `(x : int)`, anche se può essere sottointeso;
- un tipo di ritorno, in questo caso `int`, anch'esso opzionale;
- un corpo di funzione, in questo caso dato dal `let` interno.

INFERENZA DI TIPI L'inferenza dei tipi dell'interprete OCaml ci permette di dichiarare funzioni senza esplicitare i tipi degli argomenti e/o del risultato, come nel seguente caso:

```
let f x = x + 3;;
```

Siccome l'operatore di somma è specifico per il tipo `int` sia l'argomento della funzione che il suo risultato dovranno essere di tipo intero: il tipo di questa funzione sarà quindi denotato con `int -> int`.

CHIAMATA DI FUNZIONE Per chiamare una funzione non è necessario usare le parentesi: se dichiarassi la funzione `plus` in questo modo

```
let plus x y = x + y;;
```

l'espressione `plus 2 3` sarebbe valutata a `5`.

Possiamo anche dichiarare delle funzioni interne al corpo di un'espressione `let`:

```
let square x = x*x
    in square 3 + square 4;;
```

Questa espressione ha valore `25` poiché l'interprete calcola i valori di `square 3` e `square 4` utilizzando la formula data dal `let`, ovvero `let square x = x * x`.

FUNZIONE RICORSIVA Per dichiarare funzioni ricorsive bisogna usare la parola chiave `rec`. Ad esempio la funzione fattoriale può essere implementata in questo modo:

```
let rec fact x =
    if x = 0
        then 1
        else x * fact (x-1);;
```

Ancora una volta il meccanismo di inferenza dei tipi assegna alla funzione `fact` il tipo `int -> int`.

CURRYING Consideriamo nuovamente la funzione `plus` definita sopra. Se chiedessimo all'interprete OCaml di ricavarne il tipo, la risposta sarebbe che la funzione `plus` ha tipo `int -> int -> int`, ovvero la funzione prende un intero e restituisce un'altra funzione `int -> int`. Possiamo quindi sfruttare questo fatto per ottenere una funzione applicata parzialmente:

```
let incr = plus 1;;
```

Il tipo della funzione `incr` è `int -> int` (ovvero prende un intero e restituisce un intero), e semplicemente questa funzione si comporta come una `plus` con il primo parametro fissato ad `1`.

FUNZIONI ANONIME Possiamo dichiarare funzioni con una sintassi diversa usando la parola chiave `fun` oppure `function`.

```
let f =
    function x y = x * y;;
```

La funzione `f` è semplicemente una funzione che prende due parametri interi (chiamati `x` e `y`) e ne restituisce il prodotto, dunque è equivalente a

```
let f x y = x * y;;
```

Regola di valutazione di una funzione

La regola formale di valutazione di una chiamata di funzione è la seguente:

$$\frac{\text{Eval}(\texttt{f}) = \texttt{fun x\_1 ... x\_n = e} \quad (\forall i : \text{Eval}(\texttt{e\_i}) = v_i) \quad \text{subst}(\texttt{e}, \texttt{x\_1}, \ldots, \texttt{x\_n}, v_1, \ldots, v_n) = \texttt{e}' \quad \text{Eval}(\texttt{e}') = v}{\text{Eval}(\texttt{f e\_1 ... e\_n}) = v}.$$

La regola ci dice quindi che la chiamata di `f e_1 ... e_n` viene valutata ad un valore v se:

- `f` viene valutata ad una funzione (`fun`) che prende n parametri (`x_1 ... x_n`) e restituisce un'espressione che dipende da questi parametri (`e`);
- i parametri attuali `e_1 ... e_n` vengono valutati a dei valori $v_1, \ldots, v_n$;
- sostituendo all'interno dell'espressione `e` ogni parametro formale `x_i` con il suo valore v_i si ottiene una nuova espressione `e'`;
- la valutazione di `e'` dà come risultato v.

Esempio 3.3.1. Consideriamo la funzione `plus` definita sopra e la chiamata `plus 5 4`. Allora:

- `plus` viene valutata ad una funzione che prende due parametri e ne restituisce la somma, ovvero:
$$\text{Eval}(\texttt{plus}) = \texttt{fun x y = x + y}.$$
- I parametri vengono valutati rispettivamente a 5 e a 4.
- Sostituendo i valori 5 e 4 nell'espressione `x + y` (al posto di `x` e `y` rispettivamente) otteniamo l'espressione `5 + 4`.
- Valutando quest'ultima espressione otteniamo 9, cioè il valore restituito dalla funzione.

FUNZIONI DI ORDINE SUPERIORE Siccome le funzioni in OCaml sono oggetti del primo ordine possiamo usarle come parametri di altre funzioni, come in questo esempio:

```
let compose (f : int -> int) (g : int -> int) (x : int) : int =
    g(f x);;
```

La funzione `compose` prende tre parametri:

- una *funzione* `f` da interi in interi;
- una *funzione* `g` da interi in interi;
- un valore intero `x`.

Il risultato della funzione `compose` è la composizione matematica delle funzioni `f` e `g`.

Un altro possibile esempio è il seguente:

```
let rec n_times (f : int -> int) (n : int) : (int -> int) =
    if n = 0 then (fun x -> x)
             else compose f (n_times f (n-1));;
```

Questa funzione restituisce la composizione di una funzione `f` con se stessa per `n` volte usando la ricorsione.

POLIMORFISMO Se chiedessimo all'interprete di inferire il tipo della funzione `compose` definita sopra, rimuovendo le annotazioni di tipo, la risposta sarebbe:

```
compose : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
```

Le variabili `'a`, `'b` e `'c` sono *variabili di tipo*: ciò significa che la funzione `compose` non accetta solo parametri di tipo `int`, ma accetta tipi qualsiasi fintanto che tutti i tipi legati ad `'a` siano uguali, tutti i tipi legati a `'b` siano uguali e stessa cosa anche per `'c`.

3.4 LISTE

Una lista in OCaml è una collezione di valori dello stesso tipo. Le seguenti sono tutte liste:

```
let l1 = [1; 2; 3];;
let l2 = 1 :: 2 :: 3 :: [];;
let l3 = [];;
let l4 = 'a' :: 'b' :: ['c', 'd'];;
```

Il simbolo `[]` rappresenta una lista vuota, pertanto la lista `l3` ha tipo polimorfo `'a list`, mentre le altre liste hanno tipo `int list` oppure `char list`.

L'operatore `::` (chiamato *cons*) ci permette di aggiungere un elemento in testa ad una lista (a patto che il tipo dell'elemento sia uguale al tipo della lista). In realtà la sintassi con le parentesi quadre (ad esempio nel caso della prima lista) è del tutto equivalente alla sintassi con l'operatore cons (come nel caso della seconda lista), dunque `l1 = l2`.

Pattern matching

Per scrivere comodamente funzioni su liste è possibile usare il *pattern matching*:

```
let empty lst =
    match lst with
    | []    -> true
    | x::xs -> false;;
```

L'espressione `match lst with` ci permette di confrontare il parametro `lst` con alcuni "pattern", in modo da poter dare la risposta a seconda del pattern a cui la lista viene associata. Quindi se `lst` viene associata alla lista vuota `[]` significa che `lst` è vuota, dunque la funzione `empty` restituisce `true`; invece se `lst` viene associata ad una lista della forma `x::xs` significa che `lst` contiene almeno un elemento (cioè `x`), dunque non è vuota e pertanto restituiamo `false`.

Osserviamo che

- il pattern `[]` viene associato soltanto ad una lista vuota;
- il pattern `x::xs` viene associato ad una lista con *almeno* un elemento (`xs` rappresenta il resto della lista, che può essere vuoto oppure contenere altri elementi);
- il pattern `x::[]` viene associato ad una lista con *esattamente* un elemento;
- il pattern `x::y::xs` viene associato ad una lista con *almeno* due elementi;

- il pattern `x::y::[]` viene associato ad una lista con *esattamente* due elementi;

e così via.

Option types

Alcune funzioni su liste non possono essere applicate ad ogni tipo di lista, ma solo a liste non vuote. Un esempio è la funzione che calcola il massimo di una lista:

```
let rec max_list lst =
    match lst with
    | []    -> ???
    | [x]   -> x
    | x::xs -> max x (max_list xs);;
```

Per risolvere questi problemi (senza ricorrere all'uso del `null`) OCaml implementa gli *option types*:

```
let rec max_list lst =
    match lst with
    | [] -> None
    | x::xs -> match (max_list xs) with
               | None   -> x
               | Some v -> Some (max x v);;
```

Studiamo più nel dettaglio il funzionamento di `max_list`:

- se `lst` è `[]` allora il risultato è un valore particolare, chiamato `None` per indicare l'assenza di valore;
- se `lst` ha un elemento in testa `x` e una coda `xs`, allora la funzione
  - calcola ricorsivamente il valore della coda e lo usa come parametro del pattern matching;
  - se il risultato di `max_list xs` è `None`, allora la lista `xs` è vuota e il valore massimo della lista `x::xs` è `x`;
  - altrimenti se il valore massimo della lista `xs` è `Some v`, allora il valore massimo della lista `x::xs` è il massimo tra `x` e `v`.

Notiamo che se il risultato non è `None` allora deve essere racchiuso dal costruttore di tipo `Some`: infatti la funzione deve restituire un valore di un tipo preciso e non può restituire talvolta option types e talvolta tipi standard.

Il tipo di questa funzione è quindi `max_list : 'a list -> 'a option`, dove l'`option` indica il fatto che possiamo restituire `None`.

3.5 NUOVI TIPI DI DATO

OCaml ci permette di dichiarare nuovi tipi di dato tramite la parola chiave `type`:

```
type day =
    | Monday
    | Tuesday
    | Wednesday
    | Thursday
    | Friday
    | Saturday
    | Sunday;;
```

Questo costrutto dichiara un nuovo tipo (`day`) che può assumere i valori `Monday`, `Tuesday`, ..., `Sunday`.

Per dichiarare funzioni che operano su valori di tipo `day` possiamo usare il pattern matching:

```
let string_of_day d =
    match d with
    | Monday    -> "Monday"
    | Tuesday   -> "Tuesday"
    | Wednesday -> "Wednesday"
    | Thursday  -> "Thursday"
    | Friday    -> "Friday"
    | Saturday  -> "Saturday"
    | Sunday    -> "Sunday";;
```

I nuovi tipi di dato possono anche "trasportare valori" di tipi standard:

```
type foo =
    | Nothing
    | Int of int
    | Pair of int * int
    | String of string;;
```

Dunque i seguenti sono tutti valori di tipo `foo`:

```
Nothing;;
Int 3;;
Pair (2, 7);;
String "ciao";;
```

Il sistema di creazione di tipi di OCaml è molto potente in quanto ci consente di creare tipi ricorsivi:

```
type bool_expr =
    | True
    | False
    | And of bool_expr * bool_expr
    | Or of bool_expr * bool_expr
    | Not of bool_expr;;
```

Il tipo `bool_expr` è quindi un tipo usato per rappresentare espressioni booleane con gli operatori And, Or e Not: ad esempio i seguenti valori sono di tipo `bool_expr`.

```
True;;
And (False, True);;
Or (Not (And (True, True)), Or (Not True, False));;
```

Scriviamo ora una funzione per valutare un'espressione booleana di tipo `bool_expr`:

```
let rec bool_eval expr =
    match expr with
    | True         -> true
    | False        -> false
    | Or (e1, e2)  -> bool_eval e1 || bool_eval e2
    | And (e1, e2) -> bool_eval e1 && bool_eval e2
    | Not e1       -> not (bool_eval e1);;
```