Object Oriented Programming in Java

Dott. Ferdinando Primerano - 2024

1 - Introduzione e setup dell’ambiente. Il nostro primo programma

- Programmare: “dare ordini al computer”, in un linguaggio che esso possa comprendere a fondo, e senza ambiguità.

- A tale scopo utilizziamo dei linguaggi formali, diversi da quelli naturali. I linguaggi naturali sono necessariamente ambigui e devono essere interpretati – il computer ha bisogno di comandi precisi.

- Un linguaggio adatto a parlare in maniera precisa col computer viene detto “linguaggio di programmazione”. Il nostro linguaggio di riferimento è Java.

- Parole chiave: Eclipse, Java, concetto di progetto, setup dell’ambiente, strumenti del corso, package, main, concetto di istruzione, chiamata a metodo.

Perché Java

- Attualmente, e da molti anni, il linguaggio più usato. Di ispirazione a molti altri linguaggi e necessario nel bagaglio di un programmatore.

- Un enorme mercato, e in espansione.

- Estremamente complesso, per niente user friendly, molto poco didattico, pensato per progetti reali di grandi dimensioni. A tale scopo semplificheremo la didattica con tool appositi.

Strumenti

Eclipse: Eclipse è un IDE, un ambiente di sviluppo integrato. Versatile, ma specializzato nello sviluppo Java, anche di progetti complessi.

com.generation.library: una libreria proprietaria che semplificherà i compiti di input/output più comuni.

Il nostro progetto didattico

- Un progetto non è mai un singolo file ma un insieme di molti file di codice sorgente (.java) e accessori, e rappresenta uno o più programmi pensati per l'uso finale da parte di un utente o di un altro sviluppatore.

- Per gli scopi del corso, configureremo un progetto di natura didattica di nome OOP (Object Oriented Programming) in Eclipse, lo collegheremo alla libreria citata in precedenza e lo useremo come pilot per le dimostrazioni pratiche di tutti gli argomenti trattati.

Video 1 - Setup dell’ambiente di sviluppo e creazione del progetto OOP

Video 2 - Il nostro primo programma

Analisi del nostro primo programma

- Il nostro primo programma si è limitato a stampare a schermo “Welcome to Java”.

- Abbiamo scelto di creare un nuovo package per separarlo dalla libreria importata in precedenza. Questa è buona prassi quando si lavora, e da ora in avanti le lezioni pratiche verranno salvate nel package com.generation.lessons

- Il programma è stato inscritto in un *metodo* (sottoprogramma) di nome main() all’interno di una classe di nome Lesson001.

Definizione di package

- Un package è un contenitore di file scritti in Java, letteralmente una sottocartella nell'albero del progetto, sotto src in Eclipse.

- Ogni file scritto in Java riporta, come prima riga, l'indicazione del proprio package. Se questo non corrisponde alla cartella del file, Eclipse segnalerà un errore e il file non potrà essere *compilato*.

Compilazione dei file Java

- I file Java vengono *compilati* (in realtà, pseudo compilati), in un linguaggio che la macchina virtuale sottostante possa capire. Noi scriviamo file .java, ma ad essere eseguiti sono dei file .class, il risultato della compilazione.

- Se vedete un qualunque segno rosso in Eclipse, il file presenta degli errori di compilazione (di scrittura) e non potrà essere compilato (trasformato in class) e quindi eseguito, o utilizzato da altri file. Alla visione di un segno rosso di errore, siete obbligati a risolverlo prima di proseguire.

Prima definizione di classe Java

- Una prima definizione, informale e incompleta, per una classe Java è “contenitore di sottoprogrammi” (o *metodi*).

- I sottoprogrammi contenuti in una classe possono essere usati (*richiamati*) anche in altre classi, e da sottoprogrammi della stessa classe, secondo regole di visibilità che vedremo in seguito.

Il metodo main()

- Un *metodo* è un insieme di istruzioni. Una *istruzione* è un comando dato alla macchina.

- Il metodo private static void main(String[] args) è anche detto metodo di avvio. E' il metodo da cui parte l'esecuzione in una data classe. Ogni classe può avere il proprio main, ma non vi è costretta.

- Una classe che contenga un metodo private static void main() viene detta classe avviabile – può fungere da punto di avvio per il programma nel suo senso più lato, coinvolgendo altre classi.

Chiamata a metodo

- La chiamata a metodo è un tipo particolare di istruzione.

- Ne abbiamo già avuto un esempio: Console.print("Welcome to Java");. Il metodo main() della classe Lesson001 ha *richiamato* il metodo print della classe Console.

- "Welcome to Java" è il valore del parametro x passato al metodo Console.print. Lo approfondiremo in seguito, ma possiamo dire che il testo "Welcome to Java" è l'input del metodo Console.print. Console.print lo riceve e lo gestisce in un qualche modo - lo stampa.

Conclusioni

- Abbiamo visto come creare un progetto e collegarlo a una libreria, e come scrivere un primo programma sotto forma di classe con un proprio metodo main()

- Abbiamo scritto i nostri metodi in classi, e le classi sono state divise in package.

- Abbiamo visto che i metodi in Java possono richiamarsi a vicenda.

Esercizi

- Creare una classe di nome Lesson002 all’interno del package com.generation.lessons. Assicurarsi di avere attivato la spunta relativa al metodo main()

- All’interno del metodo main() della classe Lesson002 utilizzare il metodo Console.print() per stampare sulla prima riga il proprio nome e cognome, sulla seconda la propria età e città. Ricordarsi che i testi (le stringhe) vanno inseriti fra virgolette ("").

2 - Input, output e calcolo

- Possiamo definire una vasta classe di programmi come *funzioni(trasformazioni)* che trasformano gli input in output.

- Definiamo input le informazioni che il programma prende dal mondo esterno (tastiera, scanner, internet, un file, un database…)

- Definiamo output le informazioni che vanno dal programma al mondo esterno (lo schermo, un file, un database, internet, una stampante…)

Concetto di variabile

- Possiamo definire una variabile un contenitore o un collegamento per un valore in memoria (RAM).

- Per lavorare su un valore il programma deve averlo in memoria, vale a dire, in una variabile. La variabile è *allocata* nella RAM, che non è infinita. Notiamo che si parla della memoria di lavoro del computer, non di hard disk, ssd o affini. E' anche detta memoria volatile - i dati in essa presenti tipicamente non sussistono dopo che il programma è stato stoppato.

- Fare input significa portare valori in RAM, fare output significa copiarli, in qualche forma, fuori dalla RAM.

- Ogni variabile ha un *tipo*, che definisce il tipo di valore che potrà ospitare e, indirettamente, il suo consumo di memoria. In termini matematici diremo che il tipo definisce il dominio dei valori per la variabile.

Dichiarazione di una variabile

- Se il nostro programma ha bisogno di lavorare su di un dato dobbiamo fornirgli uno spazio dove salvarlo - una variabile. Per utilizzare una variabile dobbiamo *prima* dichiararla.

- Ipotizzando di avere bisogno di un numero intero, potremo scrivere nel codice: **int a;**

- Si tratta di un altro tipo di istruzione, nota come “dichiarazione di una variabile”. Possiamo tradurlo come “dichiara” (crea) una variabile a di tipo int. La variabile (scatola, collegamento) **a** potrà contenere solo numeri interi. Diremo che a è il nome della variabile (o la variabile stessa, per brevità), mentre int ne è il tipo.

Primo cenno sui tipi

Il concetto di tipo è complesso e articolato e ci torneremo in seguito, ma ad ora è utile citarne quattro:

- **int side;**
  Abbiamo dichiarato una variabile di nome side atta a contenere numeri interi.

- **double side;**
  Come sopra, ma in questo caso la variabile potrà contenere anche numeri con la virgola.

- **String name;**
  Abbiamo dichiarato una variabile di nome name che potrà contenere una stringa, vale a dire un testo.

- **boolean adult;**
  La variabile adult potrà avere solo uno fra due valori - true o false. Le approfondiremo in seguito.

Esempio di uso di una variabile

```
1 int a;
2 a = 5;
3 Console.print(a);
```

I numeri di riga servono solo a chiarire a cosa ci stiamo riferendo. Abbiamo tre tipologie diverse di istruzione:

- riga 1: dichiarazione di una variabile di tipo intero di nome a
- riga 2: assegnamento di una valore alla variabile (riempimento della variabile)
- riga 3: chiamata a metodo print, passando come valore del parametro il valore di a. Notiamo che non abbiamo passato la variabile a, ma il suo valore, e non verrà stampata la lettera “a”. Verrà stampato il suo valore, che è 5.

Operazioni di input

- Una operazione di input acquisisce dati dall'esterno del programma e li mette, anche per poco, in RAM

- Il nostro principale strumento di input sarà la tastiera, acquisita come parte della Console (parte inferiore a destra di Eclipse)

- La useremo per chiedere all'utente un numero intero, salvarlo nella variabile a, e stamparne il doppio salvato nella variabile b. a viene detta *variabile di input*.

Operazioni di output

- Una operazione di output invia informazioni (valori) dalla RAM all'esterno del programma.

- Il nostro principale strumento di output sarà lo schermo, tramite la Console. In effetti la console è uno strumento di input (tastiera) / output (schermo). La variabile b dell'esempio che vedremo a breve è una *variabile di output*.

Video 3 - Esempio di input e output

Espressioni

Analizziamo la seguente riga dell'esempio:

b = a*2;

a*2 viene detta *espressione*. Una espressione è *qualunque cosa produca un valore*, tipicamente un calcolo.

In questo caso siamo partiti da un numero intero (a) e siamo arrivati a un altro numero intero (b) tramite una semplice moltiplicazione. Le espressioni però sono molto più complesse di così: possono produrre testi, numeri con la virgola, rappresentazioni di stringhe di DNA, musica, immagini… ai nostri occhi sono sempre e solo "valori".

Tipi delle espressioni

- Come per le variabili, anche le espressioni hanno un tipo
- Una espressione ha un tipo che viene ricavato dalla procedura di calcolo. Avendo moltiplicato due interi, a e 2, il risultato sarà intero a sua volta. Quindi a*2 è una espressione di tipo intero, o una espressione che restituisce un intero.
- Essendo l'espressione di tipo intero, è stato necessario salvarlo in una variabile di tipo intero. Questa è una manifestazione della regola di concordanza dei tipi su cui ci fermeremo in seguito. Notiamo che b avrebbe anche potuto essere double, vale a dire accettare numeri con la virgola.

Una struttura comune per i nostri primi programmi

- Prima fase: dichiarazione delle variabili necessarie, di input o di output che siano. “Fase D”, o fase di dichiarazione.
- Seconda fase: input dei valori da parte dell’utente, tramite Console. “Fase I”, o fase di input.
- Terza fase: fase di calcolo dei risultati tramite espressioni e loro salvataggio in variabili. “Fase C”, o fase di calcolo.
- Quarta fase: fase di output, in cui stampiamo i risultati tipicamente tramite Console.print. “Fase O”, o fase di output.

“DICO”, volendo riassumere con un acronimo.

Video 4 - Analisi DICO del programma per il calcolo di area e perimetro del quadrato

Valutazione di una espressione

Analizziamo

area = side * side;

Le espressioni vengono valorizzate per essere calcolate. Le variabili vengono sostituite dai rispettivi valori e si procede col calcolo. Ipotizzando che l'utente abbia inserito 5, l'espressione viene valorizzata e l'istruzione diventa

area = 5*5;

Di conseguenza la variabile **area** conterrà il valore 25. Il valore potrebbe cambiare in futuro, ma non cambierà automaticamente se cambiamo side. Una espressione non è retroattiva - un valore prodotto e memorizzato resta quello prodotto al momento della valorizzazione.

Questo esempio è estremamente semplice, ma in seguito sarà necessario saper valutare espressioni molto complesse che non riguarderanno solo i numeri.

Esercizi

- Scrivere un programma che chieda in input un numero intero e ne stampi il doppio a schermo

- Scrivere un programma che chieda in input le misure dei due lati di un rettangolo e ne stampi area e perimetro

Conclusioni

- La maggior parte dei programmi ha bisogno di eseguire operazioni di input e di output

- L'input significa acquisire valori in RAM, anche per poco tempo, mentre l'output significa spostare valore spesso calcolati dalla RAM ad altre parti del sistema (schermo, files, database, rete…)

- Le variabili conservano sia i valori di input che di output, per il tempo necessario, e sono tipizzate. Per ora abbiamo visto pochi tipi: int, String, double, boolean.

- Gli output vengono tipicamente calcolati tramite espressioni. Una espressione è "qualunque cosa produca un valore", ed è tipizzata a sua volta.

3 - Sequenza e selezione

- Studiando l'esercizio precedente, notiamo come sia stato necessario chiedere il lato (side) prima del calcolo dell'area e del perimetro.

- L'ordine delle istruzioni conta. Se non avessimo avuto un valore in side, o non avessimo dichiarato side, non avremmo potuto calcolare area e perimeter.

- L'ordine delle istruzioni definisce il concetto di sequenza del programma. E in effetti un'altra definizione di programma è "traduzione di un algoritmo in linguaggio di programmazione", dove algoritmo è "una sequenza di passi ordinati per svolgere un compito".

- Un metodo viene eseguito, in generale, dalla prima riga all'ultima, dall'alto verso il basso.

Limiti della sequenza

- La sequenza da sola non è sufficiente per scrivere tutti i programmi computabili.

- In alcuni casi è necessario saltare delle istruzioni, quindi eseguire solo un sottoinsieme delle istruzioni del metodo.

- Questo principio di funzionamento viene detto *selezione* - scegliere, in base a delle *condizioni*, cosa eseguire e cosa no.

Un esempio concreto

Ipotizziamo di voler calcolare le tasse di un libero professionista. Ipotizziamo un Paese in cui ci sia una no-tax zone di 20.000 euro annui, e che si paghi il 20% sulla somma eccedente la no-tax zone.

Ipotizzando un libero professionista con un fatturato di 40.000 euro annui, il programma dovrà stampare come risultato il 20% di 20.000 euro, vale a dire 4000 euro. Se invece dovesse fatturarne 19.000, il programma dovrà stampare zero, o un messaggio di esenzione.

Non c'è una espressione matematica in grado di fare questo calcolo. Siamo costretti a dividere il nostro programma in due branche - la branca in cui dobbiamo pagare delle tasse, e quella in cui siamo esentati, e solo una delle due verrà eseguita.

Video 5 - Esempio di selezione: calcolo delle tasse di un libero professionista

Analisi dell’esempio precedente

L’esempio precedente è abbastanza intuitivo da leggere, ma è bene precisare alcuni aspetti tecnici:

- if-else è detto *costrutto di selezione*. Permette di dividere il nostro programma in due rami, un ramo if, eseguito se la condizione è vera, e un ramo else, eseguito se la condizione è falsa. Non è necessario ribadire che la condizione sia falsa dentro l’else - se non viene eseguito il primo ramo (il primo *blocco di codice*), viene automaticamente eseguito il secondo.

- L’espressione fra parentesi tonde nell’if viene detta *condizione*. Una condizione è un particolare tipo di espressione che restituisce un valore di tipo booleano, vale a dire che potrà assumere solo uno fra due valori: vero o falso. Il simbolo > viene detto operatore di confronto, e si usa solo coi numeri. Stringhe, booleani e altri tipi vanno gestiti diversamente.

- Un blocco di codice è un’area delimitata da parentesi graffe, che può avere le proprie variabili oltre a quelle dei blocchi che lo contengono. Approfondiremo in seguito, parlando di *scope* di una variabile.

Costrutto if senza else

- In alcuni casi non ci serve dividere il programma in due rami separati. Ci è sufficiente inserire un ramo opzionale rispetto al resto del flusso del programma. Ad esempio:

  ```
  if(year<5)

  {
          taxes = taxes * 0.8;

  }
  ```

  Questo blocco verrà eseguito solo se la variabile year è inferiore a 5, e offrirà uno sconto sulle tasse calcolate in precedenza. Notiamo anche che stiamo modificando un valore partendo dal suo valore precedentemente calcolato.

  Non è necessario fornire un else - se la condizione è falsa non faremo niente, semplicemente salteremo la terza riga di questo programma.

Costrutto if-else e sintassi alternativa

Vale per il costrutto if, o if-else, una regola generale che varrà in seguito in altre occasioni in cui vorremo usare blocchi di codice - non è obbligatorio dichiarare un blocco di codice quando abbiamo una sola istruzione.

L'esempio precedente può essere scritto anche come

```
if(year<5)
    taxes = taxes * 0.8;
```

Ma è meglio usare questa forma una volta presa dimestichezza con gli if-else, e con gli altri costrutti.

Condizioni composte

- La condizione **income > 20000** è detta condizione semplice. E’ una condizione che è formata da un unico valore di verità (valore booleano).

- Alcune condizioni non si possono scrivere così facilmente e necessitano di una logica più articolata, ottenuta combinando fra di loro diverse condizioni semplici in una *condizione composta*. A tale scopo disponiamo di una serie di operatori specializzati detti *operatori logici*. Introduciamo brevemente i tre principali:

Operatore && (AND)

Ipotizziamo di voler valutare se un soggetto sia adatto a militare nei corazzieri.

Un corazziere deve avere una età compresa fra i 18 e i 60 anni, e una statura minima di 190 cm. Scriviamo l'espressione booleana per calcolare l'ammissibilità o meno di un soggetto:

boolean acceptable= age>=18 && age<=60 && height>=190;

Il simbolo && si legge "and", e serve a congiungere fra di loro diverse condizioni, o diversi valori booleani (e infatti viene detto anche connettore booleano). Il valore di acceptable sarà vero solo se tutte le sottocondizioni (age>=18, age<=60 e height>=190) sono vere.

In tutti gli altri casi, il valore di acceptable sarà falso.

Operatore && (AND) - approfondimento

Riportiamo di seguito la tavola di verità dell'operatore AND. Ipotizzando due sottocondizioni , o due booleani qualunque, A e B, il risultato di A && B è il seguente:

A	B	A && B
V	V	V
V	F	F
F	V	F
F	F	F

E' facile estendere la tabella a un numero arbitrario di condizioni. Nel caso di && (AND), TUTTE le sottocondizioni devono essere vere per avere come risultato vero. Altrimenti sarà automaticamente falso. Inoltre, Java smetterà di valutare la condizioni appena troverà una sottocondizione falsa, per risparmiare tempo. Questo processo viene detto "lazy evaluation".

Operatore || (OR)

Ipotizziamo di voler valutare se applicare o meno uno sconto all'ingresso in una palestra pubblica. Lo sconto si applica sia alle persone molto giovani (sotto i 13 anni) che alle persone con un reddito familiare inferiore ai 20.000 euro annui. Non serve che siano vere entrambe le condizioni, per quanto in questo caso sia possibile. La condizione composta avrebbe questo aspetto:

boolean discount = age < 13 || familyincome<20000;

|| si legge "or". Una condizione composta costruita tramite un or è vera se almeno una delle sottocondizioni (o dei booleani) è vera.

Operatore || (OR) - approfondimento

Riportiamo di seguito la tavola di verità dell'operatore OR. Ipotizzando due sottocondizioni , o due booleani qualunque, A e B, il risultato di A || B è il seguente:

A	B	A || B
V	V	V
V	F	V
F	V	V
F	F	F

E' facile estendere la tabella a un numero arbitrario di condizioni. Nel caso di || (OR), basta che UNA sottocondizione sia vera per avere come risultato vero. Se nessuna sottocondizione è vera, il risultato sarà falso. Inoltre, Java smetterà di valutare la condizioni appena troverà una sottocondizione vera, per risparmiare tempo. Questo processo viene detto "lazy evaluation".

Operatore ! (NOT)

L’operatore ! (not) è molto semplice, e serve a invertire il valore di una condizione (o di un booleano).

Ne approfittiamo per introdurre il concetto di condizione su una stringa. Vogliamo essere sicuri che l’utente non abbia inserito come città di residenza Milano. La condizione in questo caso sarebbe:

boolean notFromMilan = !city.equals("Milano");

Equals è un metodo delle stringhe che restituisce (produce) un valore booleano. Approfondiremo questo concetto in seguito.

Video 6 - Esempio di condizioni composte

Uso delle condizioni

- Le condizioni possono essere memorizzate in variabili booleane o passate come parametro (input) di un metodo, stampate e riutilizzate in calcoli successivi.

- Il loro uso principale tuttavia è gestire il flusso del programma. Hanno un ruolo fondamentale nel processo di selezione (che abbiamo introdotto) e in quello di iterazione.

Esercizi elementari sul costrutto if

- Scrivere un programma che richieda all'utente l'età. Se l'utente è maggiorenne il programma stampa "ingresso consentito", altrimenti stampa "ingresso vietato".

- Scrivere un programma che richieda all'utente età e statura. Se l'utente ha più di 13 anni ed è alto almeno 120 cm il programma stampa "ingresso consentito", altrimenti stampa "ingresso vietato".

- Scrivere un programma che richieda all'utente l'età e il reddito familiare annuo. Nel caso in cui l'utente avesse meno di 13 anni o un reddito familiare inferiore ai 20000 euro stampare "Abbonamento richiesto 15 euro al mese", in caso contrario "Abbonamento richiesto 50 euro al mese".

Operatore ternario

In alcuni casi non è necessario disporre di interi blocchi di codice. In determinati casi è sufficiente scegliere fra due valori da assegnare a una variabile, o da passare a un metodo come input.

Nel caso in cui serva solo la scelta fra due espressioni (o valori) Java dispone di una forma rapida per la scelta che prende il nome di operatore ternario. La sua sintassi è:

v = cond ? v1 : v2;

La variabile v prende il valore di v1 se cond è true, altrimenti prende il valore di v2. I tipi di v1 e v2 devono essere compatibili con quello di v, mentre cond deve sempre e comunque produrre un booleano.

Riprendendo l'esempio del corazziere di prima:

String qualifies = age>=18 && age<=60 && height>=190 ? "YES" : "NO";

Notiamo che qualifies è una stringa. Se la condizione composta è vera, qualifies varrà YES, altrimenti varrà NO.

Video 7 - esempio di operatore ternario

Esercizi di base con l’operatore ternario

Riprendere gli esercizi elementi col costrutto if e riscriverli utilizzando l’operatore ternario per scegliere quali stringhe stampare in base alle condizioni.

Esempio di selezione complessa e sequenza

Studiamo un caso che richiede una logica più complessa rispetto a quella binaria vista fino ad ora.

Vogliamo calcolare il prezzo d'ingresso per un castello in una località fittizia sul lago di Como. Il prezzo dipende da una serie di fattori:

- i residenti del paese, di nome Riedo, non pagano. Il biglietto è gratuito
- i residenti in provincia di Como pagano un fisso di 5 euro.
- Il biglietto pieno invece è di 10 euro, ma si possono applicare due sconti cumulativi: 2 euro se la persona è un insegnante o uno studente, 1 euro se è un donatore del sangue, presentando relativa tessera. Questi sconti non sono cumulabili con i prezzi convenzionati specificati sopra.

Video 8 - Castello di Riedo

Esercizio di riepilogo

Vogliamo calcolare il prezzo d'ingresso per un museo:

- Residenti di Milano o Monza pagano un prezzo ridotto di 5 euro
- Junior (sotto i 12 anni) o senior (sopra i 65) pagano 6 euro.
- Il prezzo pieno è di 10 euro, ma ci sono due sconti cumulativi, 1 euro per gli iscritti alla pro loco, 1 euro per i donatori del sangue.

Affrontare il problema prendendo come riferimento l'esercizio sul Castello di Riedo.

Costrutto switch

Analizziamo questa problematica di selezione, relativa all'applicazione di uno sconto in base a un livello di membership:

- i membri Gold hanno diritto a uno sconto del 20%
- i membri Silver del 10%
- i membri Bronze del 5%
- per tutti gli altri lo sconto è 5% solo per il primo ordine, altrimenti è 0

Si tratta di condizioni che riguardano principalmente il livello di membership, una stringa. Possiamo affrontare la logica di calcolo tramite un classico if-else:

Prima soluzione con if-else

```
if(membership.equals("Gold"))
        discount = 20;
else
        if(membership.equals("Silver"))
                discount=10;
        else
                if(membership.equals("Bronze"))
                        discount = 5;
                else
                        if(firstorder)
                                discount = 5;
                        else
                                discount = 0;
```

Non facilissima da leggere, anche indentandola bene. Ha un suo fascino dadaista, ma non è quello che consiglieremmo nella pratica.

Notiamo che firstorder è già una variabile booleana, e infatti posso usarla dentro l'if come condizione.

Esempio di switch

```
switch(espressione)
{
        case v1:
                // istruzione 1
        break;
        case v2:
                // istruzione 2
        break;
        default:
                // se nessun caso si verifica

}
```

Possiamo riscriverla con uno *switch*.

Lo switch è un costrutto che permette di gestire le casistiche relative al valore di una data espressione, riscrivendo l'esercizio di prima enumerando i valori della membership e gestendoli uno per uno.

Rifattorizzazione dell’esercizio precedente con gli switch

```
if(membership.equals("Gold"))
        discount = 20;
else
        if(membership.equals("Silver"))
                discount=10;
        else
                if(membership.equals("Bronze"))
                        discount = 5;
                else
                        if(firstorder)
                                discount = 5;
                        else
                                discount = 0;
```

```
switch(membership)
{
        case "Gold":
                discount = 20;
        break;
        case "Silver":
                discount = 10;
        break;
        case "Bronze":
                discount = 5;
        break;
        default:
                discount = firstorder ? 5 : 0;
}
```

Analisi dell’esempio rifattorizzato

```
1  switch(membership)
2  {
3       case "Gold":
4               discount = 20;
5       break;
6       case "Silver":
7               discount = 10;
8       break;
9       case "Bronze":
10              discount = 5;
11      break;
12      default:
13              discount = firstorder ? 5 : 0;
14 }
```

- Questa forma di selezione, come tutte le altre, nasconde dei salti nel codice.
- Nel caso in cui membership fosse “Bronze” l’esecuzione salterebbe dalla riga 1 alla riga 9.
- Verrebbe quindi eseguita la riga 10, e il break ci porterebbe poi fuori dallo switch, alla riga 15 del metodo.

Conclusioni

- Sequenza e selezione sono intimamente legate. La selezione, nelle sue tre forme, ci permette di decidere quali istruzioni vengono eseguite e quali saltate, ma le istruzioni eseguite rispettano comunque i principi di sequenza: dobbiamo avere i valori nelle variabili per usarle, e i cambiamenti a una variabile non sono retroattivi, non cambiano il risultato dei calcoli già eseguiti.

- Le tre forme di selezione in Java sono if (con o senza else), l'operatore ternario e lo switch. Lo strumento generale è l'if, mentre operatore ternario e switch sono forme specializzate di comodo.

- La selezione si basa sul concetto di condizione, vale a dire di dato booleano calcolato o di espressione booleana. Una condizione vera in un costrutto di selezione indica che verranno eseguite delle istruzioni che altrimenti non verrebbero eseguite.

Esercizi finali - Seconda parte

- Creare un programma che chieda all'utente un valore intero. Se il valore è >=18, il programma stampa "ingresso consentito", altrimenti stampa "ingresso negato".
- Creare un programma che chieda all'utente età e genere (usare Console.readInt e Console.read). Se l'utente ha più di 17 anni ed è un uomo, stampare "Costo biglietto 18 euro", se ha più di 17 anni ed è una donna stampare "Costo biglietto 10 euro". Se non ha almeno 18 anni stampare "ingresso vietato".
- Creare un programma che chieda all'utente nome, data ed età e stampi il biglietto del museo per quel cliente. Il biglietto deve riportare nome, data e prezzo. Gli over 65 e gli under 12 pagano 5, tutti altri 10.
- Creare un programma che stampi un biglietto d'ingresso al Palazzo Mediceo: chiedere all'utente gli stessi input dell'esercizio precedente, con l'aggiunta dell'orario di visita. Il biglietto deve riportare nome, data, ora e prezzo. Gli over 65 e gli under 12 pagano 5, tutti gli altri 10. La fascia oraria di apertura del Palazzo è dalle 9 alle 18, se l'utente inserisce un orario che non rientra in questa fascia, stampare un messaggio di errore, altrimenti stampare il biglietto.
- Riprendendo il programma precedente modificare la procedura di calcolo del prezzo. Abbiamo permesso aperture straordinarie dalle 21 alle 23. Quindi accetteremo anche ingressi in quella fascia, con una maggiorazione di 2 euro.
- Riprendendo l'esercizio precedente, rendiamo gratuito solo l'ingresso straordinario a insegnanti e studenti d'arte. Nel caso di ingresso gratuito il biglietto deve riportare la dicitura "ingresso libero per ragioni di studio".

Esercizi finali - seconda parte

- Creare un programma che chieda all'utente statura, peso e genere. Il programma deve stampare il BMI, il fabbisogno calorico giornaliero, che dipenderà dal genere e dal peso, e un avviso in caso di BMI troppo alto (>28). In caso di statura o peso inferiori a zero o di BMI inferiore a 5 stampare un messaggio di errore. La formula del BMI è peso / altezza al quadrato, con peso espresso in kg e altezza in metri, quindi con la virgola (il punto, in programmazione).

- Creare un programma che calcoli il costo di una vacanza, chiedendo all'utente di inserire la destinazione, il costo del biglietto di andata, il costo del biglietto di ritorno, il costo per notte dell'albergo e il numero di notti. Se tutti i costi sono positivi, stampare un riepilogo coi singoli costi, col costo totale di pernottamento, col costo totale di viaggio e col costo totale della vacanza. Se un qualunque costo dovesse essere negativo stampare un messaggio di errore.

- Riprendendo l'esercizio precedente, chiedere all'utente se vuole aggiungere una assicurazione sul viaggio. Il costo dell'assicurazione è pari al 5% del costo totale. Modificare il riepilogo per presentare anche il costo dell'assicurazione se l'utente la sceglie.

- Creare un programma che calcoli il costo di un viaggio in treno, chiedendo all'utente partenza, destinazione, distanza in km. Il costo dipende dalla distanza e dalla tipologia del treno. Per i regionali il costo è 30 centesimi per km. Per gli intercity è 45. Per le frecce è 55. Un biglietto di freccia rossa non può mai costare meno di 19 euro.

3 - Iterazione

- Gli strumenti fino ad ora presentati, sequenza e selezione, non sono sufficienti a risolvere molti problemi di uso comune.

- Abbiamo spesso bisogno di ripetere porzioni di codice a seconda del valore di verità di determinate condizioni. In gergo diremo che abbiamo bisogno di forme di iterazione, vale a dire di cicli.

- L'iterazione, come la selezione, si traduce nel concetto di salto da una istruzione all'altra, quindi di una eccezione alla normale sequenza del programma, al suo normale flusso.

Un esempio pratico di iterazione

Vogliamo chiedere all'utente dei costi, dei numeri interi per praticità, fino a quando non ci segnalerà che ne ha abbastanza. Al termine del processo di inserimento, vorremmo stampare la somma di questi costi.

Si tratta di una procedura piuttosto standard, comune nel pianificare, ad esempio, una vacanza o un budget casalingo, ma non siamo in grado di affrontarla efficacemente con gli strumenti che abbiamo a disposizione. In particolare, non siamo in grado di ripetere un numero arbitrario di volte la procedura di inserimento, e di somma, senza usare un ciclo, vale a dire una forma di iterazione.

Video 9 - Il ciclo do-while

Punti chiave del video

- Il blocco di codice che comincia dopo il do e finisce prima del while è detto *corpo del ciclo*
- Il corpo del ciclo viene ripetuto fin tanto che la condizione dentro il while è vera. Infatti viene detta *condizione di ripetizione*.

- Il ciclo do-while verrà eseguito da una a infinite volte. Ogni esecuzione viene detta “iterazione del ciclo”.

- Possiamo capire meglio il ciclo se capiamo che è un modo raffinato di indicare un salto da una riga all’altra del codice. Se la condizione di ripetizione è vera, l’esecuzione ricomincia dalla prima istruzione dopo il do. In caso contrario, l’esecuzione prosegue con la prima istruzione dopo il while. Nel primo caso diciamo che restiamo nel ciclo, nel secondo che usciamo dal ciclo.

Ciclo while

In alcuni casi vogliamo ripetere una operazione da 0 a infinite volte. In questo caso si parla di un ciclo a esecuzione opzionale, e il do while non è adatto, venendo sempre e comunque eseguito almeno una volta.

In questo caso possiamo ricorrere a un *ciclo while*, che si comporta come il do while con una differenza: la condizione di ripetizione viene valutata prima del corpo del ciclo, non dopo, e nel caso in cui fosse falsa da principio l'intero ciclo verrebbe saltato.

Un esempio di ciclo while

```
1 int n = Console.readInt();
2 while(n>0)
3 {
4     n = n - 1;
5     Console.print(n);
6 }
7 Console.print("Finito");
```

- In questo esempio chiediamo un valore per la variabile n da Console, e procediamo con un conto alla rovescia.
- La condizione di ripetizione è n>0, e viene valutata una prima volta alla riga 2 del programma. Se è vera, vengono eseguite le righe 4 e 5, poi viene rivalutata la riga 2.
- Se è ancora vera, rientreremo nel ciclo (riga 4), in caso contrario usciremo (riga 7). Si tratta di modifiche nel flusso del programma, in cui saltiamo da una istruzione all’altra.

Il ciclo for

- Nell’esempio precedente siamo partiti da un valore, lo abbiamo cambiato a ogni iterazione, e abbiamo continuato a ripetere il corpo del ciclo fin tanto che quel valore soddisfava una data condizione.

- Si tratta di una attività comune. Abbiamo spesso bisogno di modificare delle variabili a ogni iterazione di un ciclo, e terminiamo quando il valore di quelle variabili non rispetta più una determinata condizione.

- In Java disponiamo di un ciclo specializzato per questa tipologia di operazione, che permette di specificare in maniera elegante e concisa delle variazioni da ripetere a ogni iterazione, così come i valori di partenza per quelle variabili e la condizione di ripetizione. Si parla di ciclo *for*.

Struttura di un ciclo for

La struttura di un ciclo for è la seguente:

```
for(inizializzazione;condizione di ripetizione;aggiornamento)
{
        // corpo del for
}
```

Laddove :

- inizializzazione è un'istruzione, o una serie di istruzioni, da eseguire una sola volta, prima della prima esecuzione del for
- la condizione di ripetizione è esattamente analoga a quella del while: viene eseguita prima della prima esecuzione, e quindi anche il for è un ciclo ad esecuzione opzionale
- aggiornamento è una istruzione, o una serie di istruzioni, che servono tipicamente a modificare le variabili "gestite" dal for

Esempi di ciclo for

```
for(int i=0;i <10 ;i++)
        Console.print(i);

//inizializza i a 0
//prosegui fino a che i è <10;
//a ogni giro aumenta i di 1
//(i++ è la forma ristretta di i = i+1)
//stamperà i numeri da 0 a 9
```

Notiamo come in questo caso il corpo del for si riduca a una linea, e quindi non siano obbligatorie le graffe.

Esempi di ciclo for - continua

```
 for(int n=1, i=1; i*n <30; i++, n+=2) Console.print(i*n);
//inizializza i a 1, n a 1
//prosegui fin tanto che il loro prodotto è inferiore a 30
// stampa n*i
// a ogni giro aumenta i di uno, n di 2.
// L'aggiornamento viene eseguito DOPO l'esecuzione
// stamperà 1,6,15, 28. Poi i*n sarà> 30 e usciremo dal ciclo
```

Approfondimento sul ciclo for

Le parti della dichiarazione del ciclo for - inizializzazione, condizione e avanzamento - sono tutte opzionali.
Potrebbe mancarne una o più di una. Ad esempio, possiamo riscrivere il programma del conto alla rovescia di prima in questo modo:

```
int n = Console.readInt();
for(;n>0;n - -)
        Console.print(n);
```

In questo caso abbiamo lasciato l'inizializzazione fuori dal for.

Esistono anche *for* in cui omettiamo la condizione di ripetizione o quella di avanzamento, e che facciamo terminare con altri strumenti che vedremo a breve.

Un discorso da rimandare - il for avanzato

Esiste una ulteriore specializzazione del for che dovremo rimandare a tempi successivi, ed è il for avanzato o “for each”. Viene utilizzato per ciclare gli elementi di un insieme di dati, e lo vedremo contestualmente all’introduzione di questo concetto.

Saltare una iterazione - continue

Java mette a disposizione due parole chiave che permettono di alterare l'esecuzione di un ciclo, saltando un "giro" in un caso e terminando il ciclo anticipatamente in un altro. Queste sono rispettivamente *continue* e *break*.

La parola chiave continue indica al programma di passare al prossimo giro del ciclo, quale che sia il ciclo in questione. Come per break, possiamo usare continue in qualunque tipo di ciclo. Supponiamo di voler sommare solo i numeri positivi fra i primi dieci inseriti dall'utente:

```
int sum = 0;
for(int i = 0 ; i <10 ; i++)
{
	int n = Console.readInt();
	if(n<=0)
		continue;
	sum+=n;
}
```

La parola chiave continue ci porterà alla successiva iterazione del for. Il ciclo non viene terminato, ma la terza riga del suo corpo (sum+=n, vale a dire sum = sum +n;) verrà saltata, e la variabile i verrà incrementata di 1, essendo questa la condizione di avanzamento. Al verificarsi della condizione di continue l'esecuzione ricomincia dalla riga del for (seconda riga del programma), portando all'esecuzione dell'avanzamento e alla rivalutazione della condizione di ripetizione.

Interrompere prima un ciclo - break

- Di natura diversa la parola chiave *break*, che interrompe il ciclo a prescindere dalla condizione di ripetizione.

- Possiamo usarlo, e spesso lo faremo, per interrompere una ricerca quando abbiamo trovato il nostro risultato. Supponiamo di avere una lista con 100.000 stature, e di cercare qualcuno sopra il metro e novanta. Una volta trovato non avrebbe molto senso continuare nel ciclo, e a quel punto la soluzione più naturale sarebbe interromperlo prima:

```
int i;
for(i = 0 ; i <100000 ; i++)
{
        //chiedo una statura
        int n = Console.readInt();
        if(n> 190)
                break; //break termina prima il ciclo
}
Console.print("Trovato alla posizione "+i);
```

Questo ciclo terminerà appena troviamo qualcuno sopra il metro e novanta, o dopo diecimila inserimenti. Notiamo che non siamo mai obbligati a usare break e continue, ma in qualche caso semplificano il lavoro.

Video 10 - Esempio di uso classico di un for - statistiche su n stature.

Conclusioni

- Sequenza e selezione da sole non bastano a scrivere tutti i programmi possibili. Serve poter ripetere ad libitum porzioni di codice. In sostanza, serve poter saltare da una riga dall'altra del codice. Questo processo di ripetizione viene detto iterazione.

- Java offre quattro forme di iterazione: while, do-while, for standard e for avanzato (che vedremo in seguito).

- Ciascuna è adatta a uno scopo particolare, ma notiamo che il do-while verrà sempre eseguito almeno una volta, mentre il for e il while potrebbero essere saltati interamente.

- Come visto anche in precedenza, le variabili definite in un blocco di codice iterato esistono solo in quel blocco.

Esercizi

- Scrivere un programma che inserisca da uno a infiniti numeri interi. Quando l'utente ha finito di inserire, stampare il più grande inserito e la media di detti numeri.

- Scrivere un programma che chieda all'utente quante bollette deve pagare. Proceda poi, tramite un ciclo for, a chiedere per ogni bolletta la causale e l'importo. A fine programma stampare l'elenco delle bollette con i relativi importi e la somma totale da pagare.

- Scrivere un programma che stampi i numeri pari inferiori a 1000. Usare un ciclo for o un while.

- Scrivere un programma che stampi sulla stessa riga ogni numero dispari sotto mille col numero pari che lo segue. Ad esempio 1-2, 3-4, 5-6 ecc…

4 - Metodi

Concetti di base

- Un metodo è un sottoprogramma atto a svolgere un compito.
- Fino ad ora abbiamo scritto solo nel metodo main, o *sottoprogramma di avvio*. Potremo dire che abbiamo scritto il *corpo* del main.
- Nonostante questo abbiamo usato, *richiamato*, diversi altri metodi. Console.print, Console.readInt(), per fare due esempi rapidi.
- In un progetto Java reale, scriveremo diversi metodi, in classi diverse. In un progetto Java, metodi scritti nella stessa classe, e in classi diverse, si richiamano a vicenda, si usano a vicenda.

Definizione di metodo

- Un metodo è un sottoprogramma . Può appartenere a un tipo o a un oggetto di un dato tipo (vedremo a breve la differenza formale), ma non sarà mai "libero": avrà sempre un contenitore. Apparterrà sempre a qualcosa, e il suo contenitore determinerà a quali informazioni, e a quali altri metodi, avrà accesso.

- Creiamo, *definiamo*, un metodo quando identifichiamo una operazione ripetitiva che vogliamo compiere più volte senza doverla riscrivere ogni volta. In questo caso creiamo il metodo e lo *richiamiamo* a seconda delle necessità. Abbiamo avuto un esempio di fronte agli occhi tutto il tempo - Console.print(), Console.readInt().

- Creiamo un metodo anche quando dobbiamo scorporare un metodo più grande, dividendolo in pezzi più piccoli e più facili da gestire.

Metodi void e non void

Una prima suddivisione fra i metodi è:

- i metodi *void*, fra cui il già noto main(), che eseguono un compito senza produrre un valore di risposta (senza un valore di ritorno). Ad esempio Console.print(). Console.print() svolge un compito, ma non produce un valore per il *chiamante* (vedremo a breve).

- i metodi *non void*, che devono restituire, *produrre*, un valore. Un esempio è Console.readInt(), che ha sempre prodotto numeri interi per i nostri metodi main().

Chiamante e chiamato

- La classe Square contiene un metodo main() che richiede un intero da Console. Nel metodo main() scriviamo int side = Console.readInt();

- Diremo che Square.main() è il *chiamante,* il metodo che usa un altro metodo, e Console.readInt() è il *chiamato*. Il chiamante richiede un servizio, il chiamato lo fornisce.

- Il metodo main() è nella classe Square. Il metodo readInt() è nella classe Console. In questo caso main() può usare readInt() - diremo che readInt() è *visibile* a main, ma non è sempre vero.

Un esempio di scrittura di metodo

- Vogliamo scrivere un programma che calcoli il fattoriale di un numero n, con n>1. Il fattoriale viene calcolato come n * n-1 * n-2 …. * 1. Chiameremo questo metodo fact.

- Il calcolo del fattoriale è usato spesso in matematica, e ci conviene isolarlo in una classe a parte per poterlo richiamare in vari progetti, come stiamo facendo con Console.readInt(). Lo metteremo in una classe di nome OurMath, che includeremo nella nostra libreria.

- Dobbiamo definire lo scambio di dati fra chiamante e chiamato perché questo funzioni. Il chiamante, che non conosciamo, dovrà fornire un valore n di cui calcolare il fattoriale. Il chiamato, OurMath.fact, dovrà fornire la risposta.

Il metodo fact(int n)

```
public static int fact(int n)
{
	int res = 1;
	for(int i=n;i>1;i- -)
		res*=i;
	return res;
}
```

Non spaventiamoci. Andremo ad analizzare questo esempio a breve, riga per riga. Per ora notiamo il “public” a inizio metodo.

public viene detto modificatore di visibilità, ed indica che questo metodo sarà utilizzabile da qualunque altra classe (o oggetto, come vedremo in seguito) del nostro programma. Non è il caso di default, ma è un caso piuttosto comune.

Approfondiremo i modificatori di visibilità nella sezione *sull’incapsulamento*.

La parola chiave static

- static, applicato a variabili o metodi, indica che quella variabile o metodo appartiene alla classe, e non agli oggetti che creeremo da quella classe (vedremo a breve)

- mi permette di usare direttamente la dicitura OurMath.fact(5) per calcolare il fattoriale di 5, da main() o da qualunque altro metodo. Quando avremo metodi di oggetto dovremo lavorare in maniera diversa.

- notate che main() è public static void: utilizzabile da qualunque altro metodo, utilizzabile direttamente tramite la classe che lo contiene, e privo di ritorno.

Parametri e ritorno del metodo int fact(int n)

- Ma cosa vuol dire int fact(int n)?

- dopo public e static troviamo la parola chiave int, numero intero. E' posta appena prima del nome del metodo (fact), e indica il *tipo di ritorno*. Questo metodo si impegna a restituire al chiamante un numero intero.

- dopo il nome del metodo troviamo le parentesi tonde. Fra le parentesi tonde troviamo int n. n è detto *parametro* del metodo fact: un parametro è *un valore che arriva dal chiamante al chiamato*, ed è necessario al chiamato per funzionare. Qualunque valore il chiamante (nel nostro caso sarà il main) ponga fra le parentesi all'atto del richiamo del metodo verrà copiato nella variabile n.

- Come si legge, quindi, int fact(int n)? Si legge: chi richiama fact dovrà fornirgli un valore che verrà salvato nel parametro n, e ne otterrà in cambio un numero intero, contenente il fattoriale del valore n passato a fact.

Il corpo del metodo e la parola chiave return

- ad ora abbiamo esaminato solo la prima riga del metodo, anche detta sua dichiarazione o *firma*.
- esaminiamo il corpo:

```
int res = 1;
for(int i=n;i>1;i- -)
        res*=i;
return res;
```

Variabili locali al metodo: res

- la variabile di nome res appartiene al metodo fact(). Non esiste al di fuori. In generale, il campo di esistenza delle variabili, il loro *scope*, è il blocco di codice in cui sono dichiarate. Un metodo è un blocco di codice, res viene dichiarata dentro il metodo, res esiste solo dentro il metodo.

- il valore di res però potrà uscire dal metodo, e continuare a vivere molto dopo che il metodo sarà terminato, come vedremo a breve.

- in questo caso la variabile res ci serve per calcolare il fattoriale. Viene inizializzata ad 1, e la moltiplicheremo per tutti i numeri della sequenza, fino a 2.

Un for insolito

- for(int i=n;i>1;i- -)
  res*=i;

- In questo for la variabile di scorrimento, i, parte dal valore n.
  Notiamo che non sappiamo da che valore partiremo. n è un valore che arriva dal chiamante, e potrebbe essere qualunque numero intero.

- Il for viene ripetuto fin tanto che i è >1, e a ogni giro i viene decrementato. E' un conto alla rovescia. Per n=5, i varrà 5, 4, 3, 2, poi falsificheremo la condizione e usciremo dal for.

- res*=i, è un esercizio di stile, e si legge res = res * i; Moltiplico il vecchio valore di res per il nuovo i. Per i = 5, res sarà prima 5*1, poi 5*4, poi 5*4*3, poi 5*4*3*2, e poi usciremo dal for. Si tratta di una forma di *accumulazione*.

La parola chiave return

- Abbiamo parlato di chiamante e chiamato, e di ritorno. return significa letteralmente “rimanda l’esecuzione al chiamante”.

- Invocare un metodo significa lasciare in sospeso l’esecuzione del metodo che stava venendo eseguito, e saltare al metodo invocato. Quando il metodo invocato, o richiamato, termina, l’esecuzione torna alla riga del chiamante che aveva eseguito l’invocazione.

- Nel caso dei metodi void, si prosegue senza altre conseguenze, ma i metodi non void *restituiscono un valore al chiamante*.

- Un metodo void non è obbligato ad avere un return, ma può averlo, mentre uno non void è obbligato ad averlo. In ambo i casi, l’esecuzione dell’istruzione return termina il metodo richiamato e torna al chiamante. Nel caso dei metodi non void, torna fornendo un valore, che il chiamante può salvare in una variabile, stampare, usare per un calcolo o ignorare.

Il primo esempio di return

- Come si legge return res; ?
- Si legge: ritorna al chiamante, qualunque metodo egli sia, e forniscigli come valore il valore calcolato per la variabile res.
- Notiamo che non stiamo rimandando la variabile res, ma il suo valore.
- Un esempio concreto:
  int a = Console.readInt(); // il metodo readInt() fornisce al chiamante un int. Il chiamante lo salva in a.

  int f = OurMath.fact(a); // il metodo fact()

  Se ipotizziamo che l'utente inserisca 5 da Console, a varrà 5, OurMath.fact.res varrà 120 alla fine, e il valore 120 verrà copiato nella variabile f del chiamante, che ipotizziamo essere main. Quello che scriviamo in mezzo alle parentesi quando invochiamo il metodo è il **valore** del parametro, mentre "int n" che potevamo leggere nella dichiarazione del metodo è la dichiarazione del parametro. Definendo il metodo decidiamo quali parametri ci servono, di che tipo, e il loro nome - prepariamo le scatole per i valori in ingresso. Richiamando il metodo, riempiamo quelle scatole.

Passaggio di parametri

- Abbiamo appena visto un esempio di passaggio di parametro a un metodo
- main() ha passato, inviato, il valore della variabile a al metodo fact(int n). fact ha salvato il valore nella propria variabile n, e ha poi rimandato a main il valore 120, che main() ha salvato in f.
- Il chiamante può inviare parametri, dati di input, e può ricevere un ritorno, dati di output dal chiamato al chiamante. Entrambe le azioni sono opzionali. I parametri possono essere diversi, il ritorno è uno solo.
- **Console.print("Ciao");** Invochiamo il metodo print della classe Console e passiamo come valore del parametro x la stringa Ciao. Nessun ritorno, un parametro.
- **int a = Console.readInt();** Invochiamo il metodo readInt() della classe Console. Niente parametri, ma il chiamante riceve un valore che memorizza in a.

Video 11 - Un esempio pratico: scrivere e richiamare il metodo OurMath.fact()

Anatomia formale di un metodo

- *visibilità static ritorno* nome (*parametri*) { *corpo del metodo* }
- Le parti in corsivo sono opzionali. Il corpo del metodo deve contenere l'istruzione return se il metodo non è void.
- Il ritorno è "il tipo di ciò che viene restituito al chiamante"
- I parametri sono "ciò che il chiamante invia al metodo"

Approfondimenti su return

- Si dice convenzionalmente che i metodi void “non hanno ritorno”, ma non significa che non possano avere la parola chiave return

- return si usa, nei metodi void, per restituire il controllo al metodo chiamante. In questo caso non viene restituito nessun valore.

- Possiamo usare return per terminare prima un metodo void.

- Un metodo può presentare diverse volte la parola chiave return, ma il primo return che viene eseguito termina il metodo e restituisce il controllo al chiamante, sia nel caso di metodi void che non void.

Analisi di metodi di esempio

int sum(int a, int b) { return a+b;}

- visibilità: non specificata, “default”, che come vedremo dopo si legge “package”
- **static**: no. Questo metodo non potrà essere usato direttamente sulla classe in cui lo scriviamo, ma necessiterà di un *oggetto*.
- **tipo di ritorno**: int
- **parametri**: int a, int b.

Esempio di uso: sum(4,5). a=4, b=5, ritorno = 9

Ripetere l’analisi precedente per i seguenti metodi

- public static int sum(int a, int b) { return a+b; }
- public static double sum(int a, int b, int c) { return a+b+c+;}
- private static boolean isAdult(int age){return age>=18;}
- protected void printCard(String name, String surname)
  {
  Console.print("Mr "+name+" "+surname);
  }

  Notate come non abbiate bisogno di sapere cosa “private” e “protected” significhino per ora. Conoscendo la struttura del linguaggio, sapete di cosa si tratta anche senza sapere cosa significano in questo momento.

Conclusioni

- un metodo è un sottoprogramma. Come tutti i programmi, può accettare degli input e produrre degli output
- i parametri sono i dati di input, il ritorno è il dato di output
- un metodo deve dichiarare ciò di cui ha bisogno (i parametri) e ciò che produce (il tipo di ritorno)
- un metodo può dichiarare altre variabili al suo interno, che però sono locali al metodo, e non visibili all'esterno.
- un metodo void non è tenuto a eseguire return e comunque non produce nessun valore. Può eseguire return per terminare prima della sua fine naturale, ma comunque non restituirà valori al chiamante.
- chi invoca il metodo, passa gli eventuali parametri al metodo e riceve l'eventuale return viene detto chiamante. Il metodo viene detto chiamato.
- un metodo, per essere richiamato, deve essere visibile al chiamante. Lo vedremo nel capitolo sull'incapsulamento.

5 - Classi e Oggetti

Introduzione ai tipi primitivi

- Fino ad ora abbiamo lavorato con variabili di vari tipi di variabili: int, double, boolean, String
- Erano strumenti che avevamo già nella cassetta degli attrezzi. Con l'eccezione di String, per cui è corretto fare un discorso separato, si parlava di *tipi primitivi*.
- I tipi primitivi sono i mattoni da cui tutto il resto è composto, e non sono solo quelli che abbiamo visto fino ad ora.
- sono riconoscibili dal fatto che il nome del tipo è in *minuscolo* (int, double, boolean, **non** String)

Tipi primitivi

- servono a indicare valori semplici
- sono parte integrante del linguaggio, non serve definirli tramite librerie o componenti esterni
- non possono avere valore “null”, come vedremo in seguito

Tipi primitivi: numeri interi

byte, short, int, long

- Interi di dimensioni sempre maggiori. Abbiamo visto principalmente int fino ad ora, il più usato e il più comune nella pratica.

Tipi primitivi: numeri con la virgola

float, double

- Numeri con la virgola con precisione singola e doppia. Il più usato è double.

- Nessuno dei due tipi è adatto a gestire dati finanziari con una buona precisione - i calcoli eseguiti su queste tipologie di variabili vengono approssimati e potrebbero non fornire risultati precisi al 100%.

Tipo primitivo boolean

boolean;

- Può contenere solo vero o falso. E’ equivalente a una condizione, e può essere usato come una condizione:

  boolean a = true;

  if(a) ….

Tipo primitivo: char

char;

- un char è un singolo carattere, e può anche essere utilizzato come valore numerico, convertendo il carattere nella sua posizione in un alfabeto convenzionale.

- A differenza delle virgole, si utilizzano gli apici singoli. char a = 'A';

- Usato raramente, ma può tornare comodo nella manipolazione dei testi.

Ma ci sono più cose in cielo e in terra...

- Notiamo che in Java non esiste un tipo primitivo per memorizzare i dati di una persona (i tipi primitivi, lo ricordiamo, sono byte, short, int, long, float, double, boolean, char).

- C'è una buona ragione per questo: programmi diversi richiedono di definire lo stesso oggetto fisico (una persona) in maniera diversa. Un programma potrebbe avere bisogno di registrare gli orientamenti politici o religiosi della persona trattata (col suo consenso) o averne il divieto assoluto. Per il programma di una palestra potremmo avere bisogno di registrare peso e statura della persona (per il calcolo della fitness), mentre non ci interesserà assolutamente registrando i clienti di un negozio online.

- In effetti, *non esiste un solo modo di definire un oggetto* . Nel momento in cui parliamo di un oggetto, fisico o meno (anche un conto in banca è un oggetto), siamo costretti a decidere cosa ci interessa sapere e a scartare il resto. Questo è detto anche problema di *modellizzazione*, cioè di creazione di un modello su cui lavorare, che si adatti ai nostri scopi. Java, quindi, non offre un tipo Person, ma ci permette di crearlo seguendo le nostre necessità.

- Creeremo un nuovo tipo di variabile, usando uno strumento che abbiamo usato per altri scopi fino ad ora - *una classe*.

Definire una classe modello - Person.java

```
// definito in Person.java.

Il file deve chiamarsi come la classe class Person
{
        public String name, surname, dateofbirth;

}
```

Abbiamo appena insegnato a Java come è fatta una Person. In termini tecnici, abbiamo appena creato un *tipo*, e per essere ancora più precisi una *classe tipo*.

Stiamo dicendo che **ogni** persona avrà **i propri** valori per le variabili name, surname e dateofbirth. Usando tre componenti che avevamo da principio (tre String), abbiamo ricavato un nuovo modello, più complesso e articolato, per un nuovo componente - abbiamo insegnato a Java come è fatta una persona, e per Java sarà il tipo *Person*, che decisamente **non** è un primitivo.

Le variabili name, surname e dateofbirth sono *proprietà* o *attributi* di ogni oggetto Person, **non** della classe Person. L'oggetto george ha la propria variabile name, e il proprio valore per name. Non ha senso dire "il nome delle Person", come se tutti avessero lo stesso nome.

Video 12 - Creazione di una classe ed esempio di creazione oggetti

Definizioni di oggetto, classe tipo e variabile

- **Classe tipo**: la classe tipo è lo stampo da cui abbiamo creato un oggetto - una istanza di una classe. *Creare un oggetto significa istanziare una classe*. La classe indica la categoria generale, l'istanza è il caso singolo. La classe regola la struttura dell'oggetto, essendo l'oggetto creato dallo stampo della classe. Socrate (oggetto) è un uomo (classe), tutti gli uomini sono mortali, quindi Socrate è mortale.

- **Oggetto**: un'area di memoria che si è convinta di essere più che un ammasso di bytes, e che contiene i dati atti a descrivere un individuo della categoria indicata dalla classe. Deve sempre essere creato da new, e con l'uso di un metodo particolare che vedremo in seguito e che viene detto "costruttore".

- **Variabile**: la variabile george, nel metodo main(), è un collegamento all'oggetto appena creato con l'*operatore new*. Per convenzione potremmo dire che george è un oggetto di classe Person, o "george is_a Person", ma bisogna capire come leggerlo. E' più corretto dire che la variabile george punta all'oggetto che abbiamo creato e riempito. E potrebbe non essere l'unica variabile che punta a quell'oggetto, che è collegata all'area di memoria che contiene i dati dell'oggetto. Tipicamente non lo sarà. Una variabile di tipo non primitivo (ad esempio Person george) è un collegamento all'area di memoria che contiene i dati dell'oggetto.

Variabile e oggetto non sono la stessa cosa

Notiamo che:

Person george;

Non crea nessun oggetto. Crea una variabile che un giorno potrebbe o meno “contenere”, in realtà *puntare a*, un oggetto.

La parola chiave *null*

- Un giorno la variabile george punterà a un oggetto Person, probabilmente, ma per ora è *null* .

- null è un termine particolare, ed equivale all'assenza di valore. Non è l'oggetto george vuoto, non è zero, non è la stringa vuota "".

- Questi sono tutti valori non nulli. null significa "assenza", una variabile che non punta a nessuna area di memoria. Il valore predefinito di tutte le variabili che possono contenere oggetti è null.

  String a;
  Person p;

  a un giorno conterrà una stringa, p un giorno conterrà una persona, ma per ora sono entrambe null. Tecnicamente, "non puntano a niente". Non equivalgono a nessuna area di memoria. Notiamo che a, String, è un oggetto di classe String (a is_a String) esattamente p è una persona (p is_a Person).

Costruttori, impliciti ed esplicit

- perchè new Person()? Cosa sono le parentesi?

- In realtà, creando un oggetto stiamo *sempre* usando un metodo particolare, da cui le parentesi, detto *costruttore*, che si trova dentro la relativa *classe tipo* . Il costruttore ha come caratteristiche quelle di avere lo stesso nome della classe e di non avere un tipo di ritorno esplicito (come invece accade agli altri metodi). Nel caso del costruttore, il tipo di ritorno e il nome coincidono, ed c'è come ritorno implicito *l'intero oggetto appena creato*.

- Ogni classe ha un costruttore implicito, privo di parametri e privo di corpo, che è presente anche se non lo scriviamo. In effetti, la classe precedente ha grosso modo questo aspetto, anche se non la vediamo:

```
public class Person
{
        String name, surname, dateofbirth;
        public Person(){ }              // costruttore, che restituisce l'oggetto che stiamo creando al chiamante
}
```

Il costruttore implicito

- public Person(){} è un costruttore implicito, ed esiste se non scriviamo nessun costruttore esplicito. Esiste, con questa forma (niente argomenti, niente corpo), fin tanto che non scriviamo un nostro costruttore.

- Nel momento in cui ne scriviamo uno nostro, un nostro metodo con lo stesso nome della classe, questo costruttore viene disabilitato, e siamo costretti a usarne uno scritto da noi, uno *esplicito*.

Un costruttore esplicito

```
public class Person
{
	String name,surname,dateofbirth;
	public Person(String n, String s, String d)
	{
		name = n;
		surname = s;
		dateofbirth = d;
	 }
 }
```

Analisi del costruttore esplicito

- Il costruttore è un metodo e valgono le regole dei metodi. Viene richiamato, ha dei parametri, ha un ritorno. In questo caso, i tre parametri n, s e d. Il ritorno è il nuovo oggetto Person che sta venendo creato in questo momento.

- In questo caso, usiamo il costruttore per impostare i valori delle proprietà dell'oggetto, ed è un caso comune. Il valore passato nel parametro n finisce nella proprietà name. Il valore passato nel parametro s finisce nella proprietà surname. Il valore passato nel parametro d finisce nella proprietà dateofbirth.

- Mentre n, s e d sono parametri, e le variabili smettono di esistere alla fine dell'esecuzione del metodo, name, surname e dateofbirth vivranno fin tanto che vive l'oggetto appena creato. Sono object-scoped - esistono dentro l'oggetto, e fin tanto che esiste l'oggetto.

Il concetto di this e l’uso di this nei costruttori

- I costruttori hanno spesso questa forma:

```
public Person(String name, String surname, String dateofbirth)
{
        this.name = name;
        this.surname = surname;
        this.dateofbirth = dateofbirth;
}
```

  this serve a indicare l’oggetto stesso in cui ci troviamo, e viene utilizzate per evitare omonimie, o ambiguità, fra parametri e proprietà.

- Nel costruttore sopra riportato i parametri (vengono da fuori, locali al metodo, verranno usati e buttati via) hanno lo stesso nome delle proprietà (restano dentro l'oggetto e ne costituiscono lo *stato*). Il costruttore Person deve poter distinguere fra le due cose. La riga this.name = name; si legge come segue: "prendi la variabile temporanea name e copia il suo valore all'interno della mia prorietà name, che invece esisterà anche fuori dal costruttore". this.name è la proprietà, name è il parametro.

- Questo è necessario solo in caso di omonimia. Negli altri metodi, o in generale quando non abbiamo parametri con lo stesso nome delle proprietà, non c'è differenza fra scrivere this.name e name. Il this in questo caso è implicito. Resta aperta una questione: cosa è this? this è come dire "me stesso", "io". Quando scrivo this dentro Person, io mi sto riferendo all'oggetto stesso.

Polimorfismo dei costruttori

- Una classe può fornire diversi costruttori, distinti uno dall'altro dal tipo e dal numero dei parametri. In questo caso si parla di polimorfismo (poli-morphos, molte forme) dei costruttori, o anche di overloading dei costruttori.

- Al polimorfismo dedicheremo un capitolo a parte, ma notiamo che la classe Person potrebbe avere due costruttori:

  public Person(){}
  public Person(String name, String surname, String dateofbirth){...}

  E la classe sceglierà quale usare in base ai parametri che riceverà. Nel caso in cui non riceva nessun parametro, userà il primo costruttore, nel caso in cui riceva tre stringhe, il secondo.

Stato di un oggetto

- Lo stato di un oggetto è *l'insieme dei **valori** delle sue **proprietà** in un dato momento*.

- L'oggetto puntato dalla variabile george aveva il valore "George" nel nome, ma questo può cambiare:

  george.name = "Giorgio";

- Lo stato di un oggetto quindi può essere mutevole.

L’oggetto come componente autonomo - insieme di proprietà e metodi

- Un altro concetto teorico importante è quello dell'oggetto come mondo a sè. L'oggetto creato (“george”) conosce solo ciò che lo riguarda (in questo caso name, surname e dateofbirth). Non conosce il mondo circostante, se non tramite i parametri che riceve nel costruttore o eventuali altri parametri dei suoi metodi.

- In effetti, per ora la classe Person è stata solo un modo elegante per raggruppare tre variabili, raggruppare variabili e rendere facile il cambiamento futuro (potremo sempre aggiungere nuove variabili a Person). Ma possiamo e dobbiamo fare molto di più.

- Vogliamo riunire in un posto solo (l'oggetto) sia i suoi valori (*lo stato dell’oggetto*) che i sottoprogrammi atti a manipolarli (*il comportamento dell’oggetto*), vale a dire dei *metodi di oggetto*. L'idea è che un oggetto sia in grado sia di contenere delle informazioni e sia di eseguire dei calcoli su se stesso . L'oggetto è una struttura in grado di riflettere su se stessa, e di produrre dei risultati o di svolgere dei compiti, in maniera tendenzialmente indipendente dal mondo esterno.

Un esempio concreto - la trasformazione in String

```
public class Person
{
	String name,surname,dateofbirth;
	public Person(String n, String s, String d)
	{
		name = n;
		surname = s;
		dateofbirth = d;
	 }

	public String toString()
	{
		return name+" "+surname+" "+dateofbirth;
	}
 }
```

- toString() è un metodo dell'oggetto Person (ad esempio, del nostro George).
- produce una stringa, ma non riceve parametri, perché non ne ha bisogno: i suoi input arrivano da dentro, dallo stato dell'oggetto.
- l'oggetto Person è necessariamente stato creato tramite l'unico costruttore presente, quindi ha ricevuto tre valori (n,s,d) e li ha salvati nelle proprietà di oggetto name, surname e dateofbirth. toString() lavora su dati che esistevano già ed erano in memoria - valori object scoped.

Video 13 - richiamare metodi di oggetto

Differenza fra metodi di classe e di oggetto

- Un metodo di classe può essere richiamato direttamente sulla classe. Non serve creare un oggetto, vale a dire un individuo, una istanza. Il programma è pensato per lavorare sulla classe.

- Un metodo di oggetto è subordinato alla creazione di un oggetto. Non posso chiedere a una persona di stampare il proprio biglietto da visita (toString) se quella persona non esiste. Il metodo toString() prevede di avere “sotto” un oggetto da cui pescare i dati relativi a nome, cognome e data di nascita visto che non gli vengono “passati” da fuori.

Esercizio di verifica

- aggiungere la proprietà String gender alla classe Person
- modificare il costruttore per ricevere anche un parametro String g che andrà a finire nella proprietà gender
- modificare il metodo toString() per incorporare nel return anche il valore della variabile g.

Elementi visibili a un metodo di oggetto

- Nella scrittura del toString() abbia usato, disinvoltamente, name, surname e dateofbirth.
- Notiamo che non potevamo usare n, s o d. Quelle variabili esistevano solo nel costruttore, erano locali al metodo, mentre name, surname e dateofbirth erano comuni a tutto l'oggetto e quindi ai suoi metodi.
- toString() ha potuto usare queste variabili, e quindi questi valori, perché erano nello stesso *scope*, nello stesso contesto.
- toString() poteva anche richiamare altri metodi dell'oggetto (un metodo può richiamarne un altro, come abbiamo visto), scrivendo semplicemente il nome del metodo, utilizzare proprietà di classe (static, ad esempio Person.MAXAGE, come vedremo fra poco) e metodi di classe.

Elementi visibili a un metodo di classe

- Al contrario, un metodo di classe non può utilizzare direttamente metodi di oggetto senza prima creare oggetti, e non può utilizzare proprietà degli oggetti della propria classe.
- Può sembrare contro intuitivo, ma vediamo un esempio. Scrivere :

  Console.print(george.name);

  ha un senso. Vogliamo stampare il nome della persona puntata dalla variabile george.
- Scrivere invece:

  Console.print(Person.name);

  E' assurdo. E' vero *che tutte le Person hanno un name*, ma *ogni Person ha il proprio*. Non esiste una sola data di nascita per tutta la razza umana - ognuno ha la propria. La data di nascita è una proprietà di oggetto, mentre l'età massima raggiungibile da un essere umano (120 anni) è una proprietà del genere umano, vale a dire della classe.
- Tecnicamente: l'individuo (l'oggetto) ha accesso a proprietà e metodi di classe (la categoria, metodi static). La classe NON ha accesso a metodi e proprietà dell'oggetto. L'oggetto conosce la propria categoria di appartenenza, la categoria non conosce l'oggetto.

Nota sulla visibilità

Notiamo che per ora stiamo parlando sempre e solo di metodi e proprietà scritti nello stesso file (la classe Person.java). Siamo sempre “interni alla classe” (o all’oggetto).

A queste problematiche si aggiungono le regole di visibilità relative a metodi presenti in classi, o oggetti, differenti dal proprio.

Parametri per i metodi di oggetto o di classe

In aggiunta a quanto scritto sopra, possiamo sempre fare arrivare ai nostri sottoprogrammi dei parametri, nella maniera a cui siamo abituati. Potremmo dire che un metodo dispone di due input principali: le proprietà dell'oggetto e della classe in cui si trova (o della classe, nel caso dei metodi static, cioè di classe) e dei parametri, quando questi esistono.

Si tratta solo di maniere diverse di fare arrivare dati a un sottoprogramma. Useremo proprietà per mantenere in scope valori che useremo diverse volte, o che potremmo dover mantenere nel tempo, mentre i parametri per i valori usa-e-getta.

Esercizio di modellizazione

- Creare una classe di nome House
- Fornirle le proprietà address (String), area (int) e spm (square meter price, int).
- Fornirle il metodo di oggetto int getPrice(), per il calcolo del prezzo, ottenuto come spm * area.
- Fornire il metodo di oggetto public String toString(), che produca una stringa con indirizzo, area e prezzo della casa.
- Scrivere una classe con un main (classe di avvio, non classe modello) che crei un oggetto casa coi seguenti dati:

  Via Verdi 100, Cassano
  Area: 100 MQ
  Prezzo al MQ: 1000

  E ne stampi il toString().

Conclusioni

- Possiamo insegnare a Java “come sono fatte le cose”, creare nuovi tipi di variabili, tramite classi modello
- Le classi modello sono stampi da cui ricaviamo oggetti. La classe è lo stampo e allo stesso tempo la categoria. L’oggetto è l’individuo.
- Gli oggetti creati dallo stesso stampo hanno sempre struttura uguale (stesse variabili, stessi metodi) ma spesso stato diverso (valori diversi nelle variabili).
- Gli oggetti possono contenere proprietà (variabili) o metodi.
- Un metodo può appartenere a una classe o a un oggetto, e la sua appartenenza determina a quali dati avrà accesso.

6 - Incapsulamento

Concetto teorico e necessità pratica

- Gli oggetti creati fino ad ora hanno permesso a determinati altri oggetti e classi di accedere alle loro proprietà e metodi. Questo può portare a problemi sgraditi: Person p = new Person("F","P", "05/02/1980"); p.dateofbirth = null;

- Abbiamo appena manomesso, da fuori, il funzionamento dell'oggetto p. La data di nascita, prima valida, ora è null. Se l'oggetto proverà a usarla per calcolare, ad esempio, l'età della persona che rappresenta, verrà generato un errore - una *eccezione*.

L’oggetto come padrone del proprio stato e l’information hiding

- L’oggetto dovrebbe essere responsabile del proprio funzionamento. A tale scopo, è buona prassi evitare che le altre parti del programma, magari scritte da altri programmatori, interagiscano direttamente con le sue proprietà.

- Inoltre, non vogliamo che il resto del sistema faccia delle supposizioni sul funzionamento interno dell’oggetto. L’oggetto deve mascherare il proprio comportamento interno e comportarsi come una scatola nera che interagisce col mondo esterno tramite i suoi metodi.

Definizione formale e pratica di incapsulamento

- L’incapsulamento è uno dei principi (quattro considerando l’astrazione) della programmazione a oggetti, ed è l’applicazione del principio di information hiding.

- Non vogliamo che il sistema esterno all’oggetto (o al package, nel caso in cui vogliamo intende il package come unità invece dell’oggetto) posso agire sul suo funzionamento interno o *fare congetture su di esso*.

- E’ utile immaginare l’oggetto, o il package, come scatole nere che rispondono a domande. Come questa risposta viene elaborata non dovrebbe, in generale, essere di interesse del chiamante. Il chiamante ha bisogno di un servizio, l’oggetto lo fornisce. Il rapporto è cliente-fornitore, un rapporto di uso.

Implementazione pratica dell’incapsulamento

- L’incapsulamento viene implementato tramite *i livelli di visibilità*.

- Questi sono quattro, in ordine di visibilità, *apertura*, crescenti, si possono applicare indifferentemente a proprietà o metodi di una classe o di un oggetto.

private

- E’ il livello più basso di visibilità, e quindi il massimo di incapsulamento. E’ il più comune per le proprietà, mentre è più raro per i metodi.

- Una proprietà o un metodo private possono essere utilizzati solo all’interno dell’oggetto o della classe in cui esistono. Ad esempio:

  - private int a; nella classe Person potrà essere utilizzato all’interno dei metodi di un oggetto di classe Person.

  - private static int b; nella classe Person potrà essere utilizzato solo nei metodi di un oggetto di classe Person o nei metodi statici della classe Person. Ricordiamoci che un metodo static “vede” solo i metodi e le proprietà di classe.

default (package)

- Omettendo la dichiarazione, un metodo o una proprietà vengono dichiarate con visibilità package. Una proprietà o metodo con visibilità package è visibile all'interno della stessa classe e nelle classi dello stesso package in cui la classe è dichiarata.

- Altre classi dello stesso package hanno accesso alla proprietà, in lettura e scrittura, e possono richiamare il metodo. Classi scritte al di fuori di quel package non vedono la proprietà o il metodo.

- E' un caso piuttosto comune, visto che un dato package viene spesso scritto dagli stessi sviluppatori, e le sue classi sono pensate per lavorare assieme, e per conoscersi a vicenda.

protected

- Una proprietà o metodo dichiarati come protected (protected int a) sono visibili nella stessa classe, nello stesso package e in tutte le *sottoclassi* di quella classe, ovunque esse si trovino, anche in package diversi.

- Il concetto di sottoclasse non è ancora stato affrontato ma lo faremo a breve. Approfondiremo maggiormente protected in quella sede.

public

- Il livello più alto di visibilità, e il più basso di incapsulamento, è public.

- Una proprietà o metodo public è visibile in tutto il progetto Java, senza distinzioni. E’ il caso di molti metodi, molto raramente delle proprietà.

Gerarchia della visibilità

- I livello vanno, dal meno visibile al più visibile, come segue:
  private -> package -> protected -> public

- *sovrascrivendo* i metodi (vedere il paragrafo *sull'override*), potremo aumentare la visibilità, mai diminuirla.

private, public e default per le classi

- I livelli private e default possono anche essere utilizzati per intere classi, in aggiunta al comune e quasi scontato public, ma questo esula dallo scopo di questo documento.

- Ci limitiamo a dire che in alcuni casi può essere utile nascondere una classe dentro il suo package, e non renderla accessibile da fuori, e anche creare una classe dentro un'altra classe ("private class"), ma non vedremo questa tecnica a questo livello.

getter e setter

- Un concetto chiave dell'incapsulamento è quello dell'accesso controllato.
- Un oggetto può concedere accesso a una proprietà proprietà privata, ma non direttamente. L'accesso è concesso tramite metodi specializzati, pensati per permettere di leggere, o di scrivere, la proprietà.
- Dovendo dare accesso, questi metodi vengono detti metodi accessori (non *superflui*, ma fornitori di accesso a parti non visibili).
- I metodi accessori pensati per la lettura di una proprietà vengono detti getter. Quelli pensati per la scrittura, setter.

Metodi getter

- Un metodo getter fornisce una lettura controllata a una proprietà.

- L'oggetto non ha l'obbligo di restituire il valore reale di quella proprietà - in effetti, può anche non esserci una vera proprietà sotto. Potrebbe trattarsi di un finto getter, e come abbiamo visto non è qualcosa che debba importare al chiamante.

- Anche nel caso di un vero getter, nessuno vieta all'oggetto di rispondere *mentendo*, o di manipolare le informazioni per rendere il tutto più semplice per il chiamante.

Struttura di un getter

- un metodo getter restituisce tipicamente un valore dello stesso tipo della proprietà a cui da' accesso. Se abbiamo una proprietà private String name, avremo un getter con questa forma:

```
public String getName()
{
    return name;
}
```

Per convenzione i getter cominciano col verbo "get", prendere, come in realtà quasi tutti i metodi di calcolo. Il getter che vedete in cima è un getter "stupido", che si limita a restituire il valore, rendendolo visibile, e potrebbe farvi dubitare dell'utilità dell'incapsulamento, ma conviene non essere frettolosi.

Utilità di un getter stupido

- Il metodo riportato in precedenza potrebbe sembrare inutile. Non fa altro che restituire il valore di una variabile che potremmo prendere normalmente se fosse public. Ipotizzando che name sia public, sarà indifferente scrivere *charles.name* o *charles.getName()*. Produrrà lo stesso valore.

- In realtà risulta estremamente utile perché potremmo avere i getter ma non i setter, vale a dire, potremmo fornire al resto del sistema una proprietà in *sola lettura*. Ipotizzando che name sia private, tutti potranno leggerlo, tramite getName(), ma nessuno potrà cambiarlo. E questa è una necessità abbastanza comune.

- C'è anche un'altra ragione, più difficile da seguire, ed è la preparazione al futuro. Un giorno potremmo non avere più la proprietà name così com'è nell'oggetto, ma voler comunque fornire il getName(). Un giorno potremmo voler memorizzare i dati internamente in un altro modo, o fornire nome e cognome in maiuscolo, e non nella loro forma memorizzata.

- Quel giorno cambieremo il getName(), che da stupido diventerà un getter trasformativo, un getter che farà effettivamente un lavoro. Vediamo di seguito un esempio di getter non stupido.

Un esempio di getter non elementare

```
public String getName()
{
	return name==null ? "" : name;
}
```

Questo getter non potrà mai restituire null, neanche quando il nostro name manca. Non potendo restituire null, avremo sempre un oggetto come risposta (la stringa vuota, "", non è null, è una stringa), e ci risparmieremo delle fastidiose *eccezioni* di tipo Null Pointer Exception.

Approfondiremo le eccezioni a breve, dopo avere introdotto un altro principio che è l'ereditarietà. Per ora definiamole "errori durante l'esecuzione del codice".

Metodi setter

- Se i getter sono lettura controllata, i setter potrebbero essere considerati scrittura controllata.

- Si tratta di metodi pensati per modificare il valore di una proprietà non visibile, spesso private, e possono svolgere anche dei compiti automatici corollari.

- Sono tipicamente void, e hanno come parametro un valore dello stesso tipo della proprietà da modificare, o di un tipo convertibile.

Struttura di un setter

Un setter per la proprietà name potrebbe essere il seguente:

```
public void setName(String name)
{
	this.name = name;
}
```

Ricordiamo che this indica l'oggetto in cui siamo, e per lo più è implicito. In questo caso abbiamo due variabili con lo stesso nome, un *parametro* di nome *name* e una *proprietà* di nome *this.name*. Usiamo this per distinguere la proprietà, quello che voglio cambiare, dal parametro, che contiene il suo nuovo valore. E in effetti useremo this *quasi* solo per evitare ambiguità.

Questo è un setter "stupido", analogo a quanto abbiamo visto prima per i getter, ma valgono per il setter stupido le stesse precisazioni fatte per il getter. Notiamo anche che il setter potrebbe mancare, pur avendo il getter.

Un esempio di setter non elementare

```
public void setName(String name)
{
	this.name = name==null ? "UNKNOWN" : name;
}
```

In questo caso scegliamo di non accettare un nuovo valore null per la proprietà name, e impostiamo il valore a “UNKNOWN”. E’ una pratica aggressiva, e non consigliabile, ma è solo un esempio. Può avere senso in altri ambiti, come nel caso degli enum che vedremo in seguito.

La pratica corretta sarebbe, volendo, generare una *eccezione*, per notificare al chiamante che il valore inviato non è corretto.

Esercizio - incapsulare la classe Person

- Riprendere in mano la classe Person
- Impostare la visibilità di tutte le proprietà a private
- Scrivere getter e setter stupidi per tutte le proprietà
- Modificare i getter e i setter per fare in modo che non accettino, né restituiscano, valori null

Video 14 - Soluzione e dimostrazione

Conclusioni

- L’incapsulamento è uno dei principi della programmazione a oggetti. L’idea è che ogni oggetto, o classe, o package, o insieme di classi imparentate, come vedremo nel prossimo capitolo, nasconda il proprio funzionamento interno al resto del sistema.

- L’accesso allo stato dell’oggetto è governato tramite metodi accessori (getter e setter), che si possono anche occupare di eseguire operazioni di controllo, sanitizzazione del dato, collegamento di variabili non collegate, e così via.

- La best practice è di avere metodi public e proprietà private, package o protected (come vedremo nel prossimo capitolo).

7 - Ereditarietà

Derivare un tipo da un altro

- Creando la classe Person abbiamo creato un nuovo tipo di variabile, che andrà a “contenere” un oggetto di tipo Person.

- Per ottenere Person abbiamo messo assieme tre stringhe e del codice. Abbiamo ottenuto Person per *composizione* di tre oggetti precedentemente disponibili - tre stringhe.

- La composizione è un metodo, il più usato e probabilmente il migliore, per creare nuovi tipi, ma Java ci mette a disposizione un secondo modo - *l’ereditarietà*.

Basi di ereditarietà

- In Java possiamo ottenere un nuovo tipo, detto *sotto-tipo*, o *sotto-classe*, o *classe-figlia*, partendo da un'altra classe, detta *super-tipo, o super-classe, o classe-madre*.

- Il processo tramite cui si ricava un nuovo tipo viene detto *estensione*, ed è una implementazione del meccanismo di *ereditarietà*.

- Se la classe B estende A (public class B extends A), la classe B eredita proprietà e metodi di A. Non necessariamente tutti, ma può ereditarli. Vuol dire che dispone già di quel materiale "genetico" per via della sua discendenza.

Un esempio concreto - Employee e Person

- La classe Person definisce solo name, surname e dateofbirth.
- Una classe Employee potrebbe avere bisogno di name, surname, dateofbirth e poi job e salary.
- Riscrivere Employee da zero sarebbe sciocco. Possiamo partire da Person e ottenere Employee *tramite ereditarietà*.

Revisione della classe Person

```
public class Person
{
        protected String name, surname,dateofbirth;

        // getter e setter....

}
```

Prima di andare avanti, operiamo una piccola revisione della classe Person.

Manteniamo i getter e i setter per il mondo esterno (e saranno public), ma apriamo leggermente la visibilità verso le sottoclassi.

Employee, che sarà sottoclasse di Person, non avrebbe potuto vedere le proprietà name, surname e dateofbirth se le avessimo lasciate private. Con protected le proprietà restano comunque nascoste al mondo esterno (al di fuori del package o delle sottoclassi di Person), ma disponibili alle sottoclassi di Person ovunque queste si trovino. Saranno quindi disponibili a Employee.

public class Employee

```
public class Employee
extends Person
{
        private int salary;
        private String job;

        // getter e setter….

}
```

Di cosa dispone un oggetto di classe Employee?

Di name, surname e dateofbirth, ereditati da Person, in quanto protected.

Di getName(), getSurname(),getDateOfBirth(), setName(),setSurname(),setDateOfBirth(), ereditati da Person, in quanto public.

Di salary e job, che invece vengono definiti localmente. Sono solo suoi.

Non serve ridichiarare name, surname e dateofbirth. Sono già presenti, sono ereditati.

Future-proofing

- Abbiamo ricavato Employee da Person. Employee non partiva da zero, ma da una base predefinita che era Person. Questo meccanismo, oltre a rendere comoda la scrittura, ci rende anche molto semplice modificare Person, e tutti i suoi sotto-tipi, in futuro.

- Ipotizziamo di avere bisogno anche del genere della persona (String gender). Se avessimo scritto Employee e Person come due classi non collegate, avremmo dovuto inserirlo due volte. Invece ad ora sarebbe sufficiente aggiungere protected String gender in Person (con eventuali getter e setter), e questo verrebbe automaticamente ereditato da Employee - e da tutte le classi presenti e *future* derivate da Person tramite ereditarietà.

- Questo ci complica la gestione. Notiamo che un cambiamento in una super classe potrebbe *rompere* tutte le sottoclassi. Queste non compilerebbero più.

Logica dell’ereditarietà

- Se B extends A, e b è un oggetto di classe B, diremo che b is_a A, o anche, b instanceof A, oltre che b instanceof b. b è istanza di B, direttamente, e di A, indirettamente.

- “Tutti gli oggetti di classe B sono anche A”, ma non tutti gli oggetti A sono B.

- Se A ha una proprietà, e questa è ereditabile (public o protected, o package se B è nello stesso package di A), allora B ha questa proprietà.
  In logica, siccome A ha p, e B è A, allora anche B ha p.

Esempio concreto di logica

- Tutti gli impiegati sono persone (se b is_a B, b is_a A)
- Tutte le persone hanno la proprietà name (se a is_a A, a has_a String name)
- Ne consegue che tutti gli impiegati hanno una proprietà name (se b is_a B, b has_a String name), a patto che name sia ereditabile.
- Se george è un impiegato, ed Employee estende Person, allora george è una Person. Se tutte le Person hanno un nome, allora george ha un nome.

Video 15 - Scrittura e dimostrazione di uso della classe Employee

Esercizio - scrittura della classe Student

- Derivare la classe Student da Person per ereditarietà.
- Aggiungere int year e String section come proprietà con visiblità protected
- Scrivere getter e setter
- Scrivere un main per creare un oggetto Student

Regole di ereditarietà

- Una classe può estenderne direttamente solo un'altra. Java viene detto "a ereditarietà singola" per questa ragione.

- Allo stesso tempo, una classe può estenderne indirettamente infinite. Se C extends B, e B extends A, allora C is_a A.

- Se c è un oggetto di classe C, allora c è anche un oggetto di classe B ed A, ed eredita da B, e tramite B da A. Diremo che c is_a C (direttamente), c is_a B, c is_a A (indirettamente). L'oggetto c appartiene a tre tipi - anche se in realtà sono quattro come vedremo a breve.

Genealogia delle classi e la classe Object

Per dimostrare questo principio creiamo una classe ForeignEmployee, con proprietà aggiuntiva protected String nativeLanguage.

Notiamo che se michael è un oggetto di tipo ForeignEmployee, michael è allo stesso tempo un Employee, e quindi anche una Person.

michael è inoltre anche un *Object*. La classe *Object* viene implicitamente estesa da tutte le altre classi di Java, ed è la base del processo di ereditarietà. Fornisce alcuni metodi di base per il funzionamento delle altre classi, ma si aspetta che questi vengano *ridefiniti*, come vedremo a breve.

Genealogia dell’oggetto michael di classe ForeignEmployee

Object	equals()			
Person	equals()	name		
Employee	equals()	name	salary	
ForeignEmployee	equals()	name	salary	nativeLanguage

Sopra vediamo la struttura genealogica che ha portato all’oggetto Michael di classe ForeignEmployee. Ogni livello contribuisce in qualche modo. Object porta il metodo, per ora usato poco, equals(). Person offre name. Employee offre salary, Foreign Employee la lingua straniera.

I colori non sono casuali. Notate che equals è rimasto scuro a tutti i livelli - viene definito, offerto, da Object, e poi ereditato da tutti i livelli successivi. Il campo name è rosso, come il colore che ho scelto per Person. Viene definito in Person e resta identico nelle sottoclassi di Person. Salary è blu perché il blu è il colore che ho scelto per Employee. nativeLanguage è verde, e non viene ereditato ma definito in ForeignEmployee.

michael avrà dei valori per tutte queste proprietà, probabilmente, e potrà richiamare tutti i metodi presenti in queste classi, a patto che siano ereditabili.

Logica della genealogia

- Person extends, implicitamente, Object: tutte le Person sono Object, non tutti gli Object sono Person

- Employee estende, esplicitamente, Person: tutti gli Employee sono Person, non tutte le Person sono Employee

- Foreign Employee estende, esplicitamente, Employee: tutti i ForeignEmployee sono Employee, non tutti gli Employee sono ForeignEmployee. Tutti i Foreign Employee sono, indirettamente, Person.

Esercizio - genealogia di un oggetto

- Scrivere la classe ForeignEmployee
- Scrivere un main che crei un oggetto di tipo ForeignEmployee
- Impostare tutte le sue proprietà.
- Rispondere alle domande: perché dispone di un nome? Perché dispone di un salario? A quanti tipi appartiene l’oggetto creato?

Override

- Una sottoclasse può ereditare sia proprietà che metodi.

- I metodi della superclasse vengono ereditati dalla sottoclasse, ma, in generale, non è obbligata a tenerli così come sono.

- Una sottoclasse può sovrascrivere (ridefinire, fare *override*) i metodi della superclasse. “Quello che andava bene per mio padre non va più bene per me”.

Un esempio emblematico di Override - il toString()

- Il metodo toString() viene definito in Object, ed è quindi presente in tutti gli oggetti di Java, anche senza scriverlo. Lo scopo del toString è quello di dare una rappresentazione stampabile, una stringa appunto, dello stato dell'oggetto a cui appartiene.

- L'implementazione di default del toString(), fornita da Object, non è particolarmente espressiva - riporta il nome della classe dell'oggetto in questione e il suo hashCode (vedremo in seguito di cosa si tratta).

- E' buona prassi, scrivendo un nuovo oggetto in Java, soprattutto nel caso di una entity, fornire una versione del toString() differente da quella standard. Si dice che è buona prassi *fare override* di toString(), quindi fornire una versione specializzata di questo metodo.

toString() in Person

```
public class Person
{
        // resto del codice
        @Override
        public String toString()
        {
                return name+" "
                        +surname+
                        " "+
                        dateofbirth;
        }
}
```

@Override è detta annotation, e serve per dire a Java che sto sovrascrivendo un metodo di un supertipo (la classe madre, o la classe “nonno”, o bisnonno, o un’interfaccia, come vedremo in seguito).

Il metodo toString() di Object è stato coperto, sovrascritto, da quello di Person. Il comportamento di mio padre non è adatto a me, e lo ridefinisco secondo i miei scopi. E’ un caso comune.

toString() in Employee

```
public class Employee extends
Person

{

	// resto del codice
	@Override
	public String toString()
	{
		return super.toString() +

			" "+job+" "+salary;

	}

}
```

In Employee mi trovo di nuovo a voler sovrascrivere toString(), ma non parto da zero.

La parola chiave super mi ricollega al mio supertipo diretto. super.toString() richiama il toString() di Person, e quindi mi fornisce una stringa con nome, cognome e data di nascita. Nel toString() di Employee aggiungo la parte puramente mia - lavoro e salario.

E’ un caso comune - partire dalla “vecchia” versione del metodo per ricavarne una nuova. Notiamo che non stiamo sovrascrivendo il metodo toString() di Object, ora, ma di Person.

Esercizio - scrivere il toString() di ForeignEmployee

- Ricavare il toString() di ForeignEmployee utilizzando la parola chiave super.
- Scrivere un main che crei un oggetto di tipo ForeignEmployee, lo riempia con dei dati e stampi il suo toString()
- Verificare che vengano stampate tutte le proprietà, inclusa la lingua nativa.

Genealogia dell'oggetto michael di classe ForeignEmployee, dopo l'override dei toString()

Object	toString()			
Person	toString()	name		
Employee	toString()	name	salary	
ForeignEmployee	toString()	name	salary	nativeLanguage

Notiamo che toString() è stato ridefinito in tutte le sottoclassi. Ogni sottoclasse ha la propria versione ora. E' il caso tipico.

Riepilogo su metodi ed ereditarietà

Per quanto riguarda i metodi rispetto all’ereditarietà possiamo trovarci in una di quattro situazioni:

- proprio, definito nella nostra stessa classe e non presente in genealogia.
- ereditato e non modificato perché ci va benissimo come lo faceva "nostro padre".
- sovrascritto (override), usando o meno il metodo originale. Il toString() è l'esempio principale
- non ereditato (metodo private o metodo package e padre ed figlio sono in packages diversi)

Polimorfismo dei metodi

Un metodo è polimorfico quando è presente in varie forme in parti diverse del sistema. Parliamo di tre tipologie di polimorfismo dei metodi

- polimorfismo dei costruttori, visto in precedenza, che è in realtà un tipo particolare di overloading

- overloading: presenza di più metodi con lo stesso nome ma parametri diversi nella stessa classe.

- overriding: presenza dello stesso metodo, stesso nome e stessi parametri, in un supertipo e un sottotipo o, se preferite, “sovrascrittura del metodo di un supertipo”.

Vincoli per la sottoclasse

- una sottoclasse non può fare override dei metodi della super classe quando questi sono marcati come “final”, che indica, per i metodi, la non sovrascrivibilità

- una sottoclasse deve sempre richiamare un costruttore della superclasse. Ricordiamoci che possiamo avere diversi costruttori per una classe, fra cui quello implicito fino a che non viene disabilitato. Una sottoclasse deve sempre richiamare un costruttore, e lo fa esplicitamente, o può farlo implicitamente, nel caso in cui la superclasse disponga di un costruttore vuoto.

- un override non può mai diminuire la visibilità di un metodo, solo aumentarla. Possiamo andare da protected nella superclasse a public nella sottoclasse, ma mai viceversa.

Richiamare il costruttore della superclasse

```
public class Person
{
	public Person(String n, String s, String d)
	{
		// codice
	}
}

public class Employee extends Person
{
	public Employee
	(
		String name,
		String surname,
		String dateofbirth,
		String job,
		int salary;
	)
	{
		super(name,surname,dateofbirth);
		this.job = job;
		this.salary = salary;
	}

}
```

In questo caso la classe Person **non** ha un costruttore vuoto da richiamare. Employee è costretto a richiamare l'unico costruttore di Person, che prevede tre parametri, tramite la parola chiave super(), questa volta usata per sinonimo per il costruttore del padre.

In termini di logica potremmo dire: una Person ha bisogno di tre stringhe per essere costruita, un Employee è una Person, di conseguenza è obbligata a fornire tre stringhe per costruire la Person, prima di costruire la parte di sè che corrisponde a un impiegato.

Se ci fosse un costruttore vuoto in person (Person(){}), il super non sarebbe obbligatorio nel costruttore di Employee.

Concordanza dei tipi ed ereditarietà

Analizziamo il seguente codice:

Person p = new Employee(...);

Potrebbe sembrare sbagliato. Person e Employee sono tipi diversi, ma affini.

In pratica, Person è un tipo più generico di Employee , e i due tipi sono legati da un rapporto di ereditarietà. Tutti gli Employee sono Person, ma non tutte le Person sono Employee. Noi diremo che un Employeeè un Person (sempre), ma un Employee non è necessariamente un Employee.

I tipi Person e Employee sono compatibili asimmetricamente : Employee è compatibile con Person (posso assegnare un oggetto Employee a una variabile di tipo Person), ma Person non è necessariamente compatibile con Employee(non potrò, salvo conversioni, assegnare un oggetto di tipo Person a una variabile di tipo Employee.

Tipo della variabile e tipo dell’oggetto

Person p = new Teacher(...); // corretto

Employee e = new Person(...); // ERRATO

Ribadiamo che variabile e oggetto sono due concetti differenti e separati.
Person p; dichiara una variabile, vale a dire una "scatola a forma di Person".

La scatola p potrà ospitare un qualunque tipo di Person (Person, Employee, Student, ForeignEmployee...), ma ad ora è vuota , vale a dire null .

Scrivendo p = new Employee(), io sto creando un oggetto di tipo Employee, e lo sto "mettendo" dentro p. Ma l'oggetto non è p: è il contenuto di p, o meglio, è quello a cui p punta.

Diremo che il tipo di p è Person, e questo è fisso, non cambierà mai. Il tipo dell’oggetto dentro p deve essere compatibile col tipo della variabile (deve essere una Person), ma potrebbe essere una Person o una sua sottoclasse. In questo caso il tipo della variabile p è Person, mentre il tipo dell’oggetto è Employee.

Noi dobbiamo lavorare tenendo conto del tipo della variabile, cercando di ignorare per quanto possibile quello dell’oggetto specifico.

Approfondimenti - polimorfismo di oggetto

Immaginiamo di avere una classe Teacher, sottotipo di Person. Partiamo da

Person p = new Teacher();

E poniamoci alcune domande:

- l'oggetto in p è un insegnante?
- l'oggetto in p è una persona ?
- l'oggetto in p è uno studente?

Cosa è Person p = new Teacher(), e come possiamo usarlo

- La risposta alla prima domanda è sì.
  Abbiamo creato un oggetto di tipo Teacher (new Teacher()), quindi è sicuramente un Teacher.

- La risposta alla seconda domanda è "sì", perchè Teacher è una sottoclasse di Person. Il rapporto di ereditarietà si legge anche "è" o “is_a”: se F è sottoclasse di P, F è anche di tipo P.

La risposta alla terza domanda è no. L'oggetto è una Person, è un Teacher, ma non è uno Student. In effetti, nessuno potrà essere allo stesso tempo Teacher e Student. Quindi, cosa rispondiamo alla domanda "a che tipo appartiene l'oggetto in p?".

Diremo che è un Teacher, e di conseguenza anche un Person. Nella pratica, l'oggetto in p appartiene a due tipi: Person e Teacher (oltre a Object, che è implicito). E potrà essere usato sia come Person che come Teacher

Polimorfismo di oggetto

Questo meccanismo viene detto polimorfismo di oggetto.

Un oggetto viene detto polimorfico quando appartiene a più tipi, e lo possiamo vedere in molti modi. In Java tutti gli oggetti sono polimorfici: tutti gli oggetti hanno un antenato in comune, la classe Object .

E' lì che viene definito il metodo toString(), ed è per questo che toString() deve restare public in tutti i sottotipi (in tutti gli oggetti): non possiamo ridurre la visibilità di un metodo di Object, essendo tutti gli oggetti discendenti di Object.

Quindi in realtà l'oggetto nella variabile p appartiene a tre tipi: Object, Person e Teacher.

instanceof

Questo ci porta a un problema: se mi arriva una variabile Person p, come posso essere sicuro del contenuto?

p potrebbe contenere una Person, così come un Teacher, così come uno Student o qualunque sottoclasse di Person.

Java mette a disposizione un operatore per verificarlo, **instanceof** :

```
Person p = new Teacher(...);
Console.print(p instanceof Person); //true
Console.print(p instanceof Teacher); //true
Console.print(p instanceof Student); //false
```

L'operatore instanceof prevede di avere una variabile a sinistra e un tipo a destra, e si legge "dimmi se il contenuto di questa variabile appartiene a questo tipo". Essendo una risposta del tipo sì-o-no, produce un booleano. p instanceof Teacher si legge anche "p è un Teacher", per semplicità, per quanto andrebbe letto come "p punta a un oggetto di tipo Teacher".

Restrizioni dovute al tipo della variabile

```
Person p = new Teacher(...);
if(p instanceof Teacher)
 {
      // QUESTO FUNZIONA
      Console.print(p.getName());
      // QUESTO NO
      Console.print(p.getSubject());
}
```

Perché non funziona?

p contiene un Teacher, lo abbiamo anche controllato, ma Java vede p come una Person.

Java lavora sui tipi formali, sui tipi delle variabili. p è stato dichiarato come Person. Java sa che sono garantiti solo i metodi e le proprietà di Person, e quindi non ci permetterà di accedere a elementi della sottoclasse (in questo caso, getSubject(), il getter per la materia di insegnamento probabilmente definito in Teacher).

Quindi, come accedere a quei dati, quando Java vede l'oggetto come se fosse del suo supertipo(una Person)? Devo forzare il tipo formale, effettuando una operazione che viene detta di **casting**, per eliminare le restrizioni dovute ai controlli di Java sul tipo degli oggetti (“type safety di Java”).

Primo esempio di casting

```
Person p = new Teacher(...);
if(p instanceof Teacher)
{
        // CASTING
        Teacher t = (Teacher) p;
        Console.print(t.getName());
        Console.print(t.getSubject());

}
```

La prima riga all'interno del blocco if è una operazione di *casting*.

Letteralmente diremo che la variabile t punta allo stesso oggetto puntato da p ma lo vede come un Teacher.

NON ne ho creato una copia : c'era un solo oggetto prima, quello puntato da p, e c'è ancora un solo oggetto.

Le variabili t e p puntano allo stesso oggetto . t e p si riferiscono alla stessa area di memoria, che contiene un oggetto senza nome, ma lo "vedono" in modo diverso. p vede l'oggetto come una Person, e il compilatore vi permetterà di usarlo solo coi metodi di Person. t invece vede l'oggetto, l'area di memoria, come un Teacher (come in effetti è) e vi permetterà di usarne i metodi.

Lavorare con variabili polimorfiche

Ci troveremo spesso a lavorare senza conoscere il tipo concreto (il tipo dell'oggetto) ma conoscendo il tipo della variabile.

In alcuni casi dovremo forzare i tipi: in tutti questi casi prima controlleremo con instanceof (è davvero di quel tipo?), poi faremo casting. Lo faremo quando avremo bisogno di accedere a metodi o proprietà delle sottoclassi. Dovremo richiedere a Java sicurezze maggiori di quelle che ci offre il tipo della variabile che stiamo gestendo.

A livello teorico, diremo che il tipo della variabile definisce **cosa** una variabile è in grado di fare, quali metodi o proprietà possiamo usare da fuori, mentre il tipo concreto, il tipo dell'oggetto, definisce **come** l'oggetto eseguirà quel compito.

Avendo dichiarato p come Person, non potrò invocare metodi della classe Teacher a meno di castare l'oggetto p. Potrò invocare toString(), perchè tutti gli oggetti lo hanno, ma ogni oggetto lo eseguirà a modo suo: ogni oggetto avrà il suo toString(). Anche se invoco toString() su Person il come verrà eseguito dipenderà dal tipo dell'oggetto, vale a dire dal tipo concreto

Sintassi formale di casting

Terminiamo fornendo la sintassi formale del casting: posso castare un oggetto o a una classe C scrivendo (C)o.

Tipicamente salverò il risultato di questa operazione (che è solo un punto di vista diverso sulla stessa area di memoria) in un'altra variabile, quindi C c1 = (C) o;.

c1 è la variabile che punta all'oggetto o visto come appartenente alla classe C. Il casting può dare eccezioni. o potrebbe non appartenere al tipo C, nel qual caso il programma andrà in eccezione, ma è facile da prevedere, essendo sufficiente verificare prima con instance of se o appartiene o meno a C.

Torniamo alla classe Object

- Tutte le classi (String, Person, Console) hanno un antenato comune nella classe Object . La classe Object definisce i metodi fondamentali condivisi da tutti gli oggetti, fra cui il toString (che infatti è e deve restare public), e altri metodi di interesse che vedremo di seguito, assieme ad alcune differenze di base nella gestione di oggetti e primitivi.

- La divisione fondamentale fra le variabili in Java è fra tipi primitivi (out of the box) e tipi di riferimento (discendenti da Object, quindi oggetti).

Operatore = per variabili di tipo primitivo

La differenza è evidente quando utilizziamo l'operatore = fra variabili di tipo primitivo o fra variaibli di riferimento.

Nel caso dei primitivi:

int a = 5;

int b = a;

b = b + 1;

Alla fine di questo programma, a = 5, b = 6. Abbiamo creato una copia del valore in a in uno spazio di memoria diverso, b, e abbiamo modificato la copia.

Operatore = per variabili di riferimento

Se invece scriviamo:

```
Person p = new Person();
p.name = "George";
Person q = p;
q.name = "Paul";
Console.print(p.name);
```

Il programma stamperà “Paul”. q non è una copia dell’oggetto p. q è *una copia del riferimento in memoria all’oggetto prima puntato da p*. Una variabile di un tipo di riferimento (una variabile che punta a un oggetto) è un indirizzo in memoria - avremmo detto un puntatore nel linguaggio C.

Scrivendo q.name = "Paul" abbiamo modificato lo stesso oggetto puntato da p. p e q sono alias per lo stesso oggetto, riferimenti alla stessa posizione di memoria.
Notate che non abbiamo fatto new due volte, ma una sola. E’ solo con new che occupiamo nuova memoria per gli oggetti. Di conseguenza in questo programma non sono stati creati due oggetti di tipo Person, ma uno solo. Sono semmai stati creati due riferimenti (p e q) verso lo stesso oggetto.

L’operatore == fra variabili primitive e fra variabili di riferimento

- Nel caso di valori primitivi, ad esempio due interi, == è l’uguaglianza di valore, o di *stato* . Ad esempio, posti int a = 5; int b = 5; boolean c = a==b; fornirà true per il valore di c

- Nel caso delle variabili di riferimento, e dei valori di riferimento, è comunque uguaglianza di valore, *ma è l’uguaglianza dei valori di riferimento*. Vale a dire, l’uguaglianza dell’indirizzo in memoria.

- Poste due variabili Person a e Person b, a==b non significa che gli oggetti puntati da a e b siano uguali, ma qualcosa di più preciso: a e b *sono la stessa cosa, puntano alla stessa area di memoria*.

== ed equals() per le variabili di riferimento

- Come abbiamo appena visto, se a e b sono due oggetti, a==b si legge “a e b puntano alla stessa area di memoria”. I due riferimenti sono uguali. a e b sono alias. Questa si chiama uguaglianza di indirizzo, o uguaglianza fra i valori dei riferimenti.

- Java prevede anche un’altra forma di confronto fra oggetti, che è il metodo equals(). equals è definito in Object, e se non viene sovrascritto è perfettamente identico a ==. Uno sviluppatore che voglia confrontare il contenuto, lo stato, di due oggetti, dovrebbe farlo facendo override di equals nella classe che produrrà gli oggetti che andranno confrontati.

- Il metodo equals prende in ingresso un altro oggetto, che viene detto other, oppure o, e restituisce true se l’altro oggetto, pur essendo in un’altra area di memoria, ha il mio stesso contento, il mio stesso stato.

- Fra gli oggetti, a==b si legge “a e b puntano allo stesso oggetto, sono la stessa cosa, quindi sono anche uguali”. a.equals(b) si dovrebbe leggere, ipotizzando di avere fatto override di equals nella classe di a, “a e b hanno lo stesso contenuto”, almeno per quello che conta. a==b implica a.equals(b), a.equals(b) **non** implica a==b.

Video 16 - scrittura di equals() per Person e sua dimostrazione

Esercizio - scrittura del metodo equals() per Employee

- Creare due oggetti di tipo Employee in un main, con lo stesso contenuto, in due variabili di nome a e b.
- Stampare a.equals(b) nel main. Verificare che dia false.
- Sovrascrivere il metodo equals() in Employee, confrontando tutte le caratteristiche di Employee.
- Eseguire di nuovo il main e verificare che sia true.

Il ruolo di equals()

- equals() è definito in Object, ed è pensato per essere usato in molte librerie standard di Java. Per la precisione, quando uno sviluppatore vuole confrontare lo stato di due oggetti si presuppone che usi equals().

- equals() viene usato implicitamente da componenti standard di Java, come le List, che vedremo a breve, per capire se un elemento è già presente o meno fra quelli conosciuti. E’ buona prassi definire equals() in un tipo ogni volta in cui vogliamo creare collezioni di elementi di quel tipo.

hashCode()

- hashCode() è un metodo definito in Object, e bisogna sovrascriverlo quando sovrascriviamo equals(). Viene utilizzato anch'esso, implicitamente, in diverse librerie Java, in particolare quelle che utilizzano tecnologie di hashing.
- La discussione delle tecnologie di hashing esula dallo scopo di questo documento, ma l'idea è quella di generare un numero intero che rappresenti lo stato dell'oggetto e che ne faciliti la ricerca in memoria - una forma di *indicizzazione*.
- L'idea è che se due oggetti sono uguali, devono restituire lo stesso hashCode. Se due oggetti hanno hashCode diversi, devono essere considerati diversi (equals deve restituire false). Questo meccanismo permette una ricerca più rapida in memoria.
- Ricordiamoci di implementare hashCode(), o anche di farlo implementare automaticamente ad Eclipse. Eclipse implementa entrambi i metodi automaticamente tramite un wizard. Sarà fondamentale farlo quando useremo una struttura dati nota come Set (da non confondersi coi setter).
- L'idea è di tradurre l'oggetto in un intero, e questo intero mi darà un'idea dell'indirizzo a cui cercarlo in memoria in strutture dati particolari. Renderà rapidissimo l'accesso.

Un esempio di hashCode()

Ipotizziamo un hashCode per una classe Date con proprietà year, month e day. Questo metodo fornisce un hashCode per collocare una data in memoria e capire più rapidamente se la abbiamo già. Una data come il 5 febbraio 1980 produrrà come hashCode 198025.
Lo vedremo in azione studiando i Set.

```
@Override public int hashCode()
{
	return Integer.parseInt(year+""+month+""+day);
}
```

Conclusioni

- Possiamo ricavare nuovi tipi tramite ereditarietà o composizione. L'ereditarietà è uno dei principi basilari della programmazione a oggetti, mentre la composizione è solo una tecnica operativa. L'ereditarietà si ottiene tramite la parola chiave extends e permette l'override.

- Preferiamo nascondere il funzionamento degli oggetti e dei package a coloro che li usano. Questo processo si chiama incapsulamento, ed è l'applicazione dell'information hiding. L'incapsulamento è uno dei principi fondamentali della programmazione a oggetti, e si esplicita con i livelli di visibilità, i getter e i setter.

- Il polimorfismo può riguardare metodi o oggetti, e si traduce con più forme dello stesso concetto. Il polimorfismo è uno dei principi fondamentali della programmazione a oggetti.

- Esiste un quarto principio, quello di astrazione, che vedremo a breve.

8 - Astrazione - parte prima

Concetto di astrazione

- L’ Astrazione è il quarto principio della programmazione a oggetti, dopo ereditarietà, incapsulamento e polimorfismo.

- La si può intendere sensu strictu, tramite l’utilizzo di strumenti che separano il comportamento atteso del codice dalla sua implementazione, o sensu lato, come attività di eliminazione dei dettagli operativi e ininfluenti per concentrarsi sulle linee generali del problema da risolvere o degli oggetti da descrivere.

Astrazione sensu lato

- Il processo generale di astrazione è stato già utilizzato nel momento in cui abbiamo isolato le caratteristiche di nostro interesse di un concetto (ad esempio, una persona o un insegnante) e scartato quelle inessenziali per i nostri scopi (ad esempio, la statura o il peso di un insegnante - che non sono rilevanti per il nostro processo decisionale o di calcolo, mentre potrebbero esserlo scrivendo il software di gestione di una palestra).

- La scrittura di una qualunque classe modello corrisponde alla scrittura di una astrazione, vale a dire di una rappresentazione astratta, modellizzata, di un oggetto, reale o meno.

Astrazione sensu strictu

- Nella programmazione a oggetti l’astrazione è un meccanismo che permette di separare la definizione di ciò che vogliamo ottenere (“cosa”) dal modo in cui lo otterremo (“come”). E’ correlata al concetto di incapsulamento e di polimorfismo, come vedremo a breve.

- I vantaggi sono molteplici, soprattutto nell’ambito della manutenibilità, del future proofing e della prevedibilità del comportamento del software.

- I due strumenti fondamentali per dell’astrazione sensu strictu in Java sono le *classi astratte* e le *interfacce*.

Classi astratte

- Una classe astratta è una classe che non può essere istanziata direttamente e può, non deve, ma può, disporre di *metodi astratti*.

- Un metodo astratto è un metodo senza corpo, ridotto alla propria firma e contrassegnato dalla parola chiave abstract. I sottotipi concreti della classe astratta hanno l'obbligo di implementarlo, tramite override, se non ci sono antenati nella loro genealogia che lo hanno già implementato.

- Un metodo astratto definisce un servizio che è richiesto agli oggetti di un determinato tipo. Il modo in cui quel servizio viene erogato dipende dal sottotipo. Ci troviamo nel caso di un override obbligato per i metodi astratti, perché in effetti un metodo astratto non ha un corpo, e quindi non ha un funzionamento di base.

Esempio di classe astratta

- Modifichiamo Person rendendola abstract:
  public abstract class Person { ….

- Non potremo più scrivere new Person nel resto del codice (una classe astratta non è istanziabile), ma potremo creare oggetti Person indirettamente, tramite le sue sottoclassi.

- Person p = new Teacher(); sarà una istruzione valida. L'oggetto puntato da p sarà allo stesso tempo una Person e un Teacher, ma non potrà mai essere solo una Person.

- Marcando Person come abstract abbiamo deciso che Person è un tipo troppo astratto, troppo generico, per essere istanziato direttamente. Nel nostro programma, accettiamo oggetti Teacher o Student, ma non "Person lisci". Ha senso se scriviamo il gestionale di una scuola - una Person dovrà ricadere in una di poche sotto tipologie, magari Employee, Teacher, Student…

La classe astratta come base per altri tipi

- Mentre una classe non astratta (concreta) funziona da base per la creazione di oggetti, una classe astratta funziona solo come base per altre classi, che verranno poi istanziate, o come *contenitore di metodi statici*, una sorta di libreria.

- Una classe può essere astratta anche senza la presenza di metodi astratti, ma la presenza di un metodo astratto rende necessariamente astratta la classe. Il metodo astratto **implica** che la classe sia astratta, il fatto che la classe si astratta **non implica** la presenza di metodi astratti.

Metodi astratti

Ipotizziamo la seguente genealogia, che modellizza le persone coinvolte nella gestione di una scuola:

Person (abstract)

Employee (abstract) Student

Teacher Support

Person è la base per Student ed Employee, ed è astratta senza definire metodi astratti. Employee è astratta, ed è estesa da Teacher (insegnante) e da Support (chiunque svolga altre funzioni - amministrative, ad esempio).

Il metodo astratto getYearlyRetribution()

- Definiamo un metodo astratto in Employee, public int getYearlyRetribution().
- La retribuzione annua cambia a seconda che l'impiegato sia un insegnate (tredici mensilità) o un impiegato di supporto (quattordici mensilità, a fronte, magari, di una retribuzione annua e di scatti di stipendio più significativi.
- Potrebbe venirci la tentazione di scrivere getYearlyRetribution() in Employee, ma sarebbe un errore. Questo ci costringerebbe a conoscere tutte le logiche di calcolo delle sue sottoclassi, e ci costringerebbe a cambiare il metodo ogni volta in cui vorremo aggiungere una qualche nuova categoria.
- La soluzione corretta, in termini di astrazione, è la seguente - la classe Employee definisce il servizio da offrire (getYearlyRetribution) **ma ne delega l'implementazione alle sue sottoclassi**. E' il meccanismo dei metodi astratti.
- La regola operativa è che la superclasse non debba mai conoscere le sue sottoclassi. Le sottoclassi conoscono la superclasse, ma il contrario non è mai una buona idea, e ci renderà le cose difficili in fase di manutenzione e di ampliamento del sistema.

Soluzione operativa

- public abstract class Employee {
  int salary;
  public abstract int getYearlyRetribution();

- public class Teacher extends Employee{
  @Override
  public int getYearlyRetribution() { return salary * 13;}

- public class Support extends Employee{
  @Override
  public int getYearlyRetribution(){ return salary * 14;}

Punti chiave:

- la classe Employee è astratta, le classi Teacher e Support non lo sono.
- il metodo getYearlyRetribution è astratto in Employee, concreto nelle sottoclassi.
- quando emergerà un terzo tipo di impiegato, con una logica di calcolo diversa, ci basterà scrivere la relativa classe (ad esempio, Freelance) e fare override di getYearlyRetribution() con la relativa logica di calcolo.

Richiamare un metodo astratto

- una classe astratta può avere anche dei metodi concreti, e tipicamente li ha, e questi metodi possono richiamare un metodo astratto, anche se la sua implementazione non è presente.

- Employee potrebbe avere il seguente toString(): public String toString(){ return this.name+" "+this.surname+" "+this.getYearlyRetribution()+ " euro";}
  Non è un problema, perché sarà impossibile scrivere “new Employee”. Saremo costretti a scrivere new Employee o new Teacher o new Freelance o comunque una qualunque sottoclasse di Employee, e la sottoclasse sarà necessariamente dotata del metodo, questa volta in forma concreta.

- Employee e = new Teacher();
  e.setName("George");
  e.setSurname("Romano");
  e.setSalary(1000);
  Console.print(e.toString());

  Questo codice stamperà “George Romano 13000 euro”. Il getYearlyRetribution() invocato dal toString() di Employee è quello di Teacher, e questo avviene senza che la classe Employee conosca Teacher.

Video 17 - scrittura e test di Person, Employee, Teacher e Support

Esercizio sulle classi astratte

- Aggiungere il metodo astratto public int getCost() in Person. Notare che aggiungere getCost() in Person porterà a “rompere” tutti i suoi sottotipi.

- Il costo di un Employee per la scuola è la retribuzione annua moltiplicata per due. Possiamo fare Override in Employee, e questo metodo verrà ereditato da Teacher e Support. Non servirà implementarlo in quelle classi.

- Il costo di uno Student per la scuola dipende dalla sua media. La scuola offre borse di studio pari a 1000 euro annui per gli studenti con media superiore a 8 e senza insufficienze. A questa si aggiunge un costo fisso di 2000 euro per la mensa.

Interfacce Java 7.

- Una *interfaccia* è una estremizzazione del concetto di astrazione. Definiamo un tipo solo tramite i servizi che dovrà offrire (i suoi metodi) ma non forniamo niente in termini di implementazione di quei metodi, almeno fino a Java 7.

- A dispetto del nome non c'entrano niente con la grafica, e una delle domande tipiche ai colloqui sarà "che differenza c'è fra interfacce e classi astratte". Lo vedremo in dettaglio.

L’interfaccia Validable

public interface Validable { boolean isValid(); }

- Questo codice, definito in una classe di nome Validable, definisce una interfaccia Validable con un solo metodo - isValid(). Il metodo isValid() deve restituire true se lo stato dell’oggetto in cui si trova è valido, false altrimenti.

- Si tratta di un *comportamento* estremamente generale che possiamo immaginare di applicare a qualunque tipo di oggetto - persone, verificando di avere nome, cognome e data di nascita, case, verificando di avere indirizzo, città e costo, animali, prodotti, movimenti finanziari…

- Non avrebbe senso definirlo classe per classe, quindi lo definiamo in un tipo a sé e ci prepariamo a farlo *implementare* da diverse classi che saranno sottotipi dell’interfaccia, anche se in maniera diversa da quella a cui siamo abituati.

- Il metodo isValid è implicitamente astratto e pubblico, come tutti i metodi di una interfaccia fino a Java 7.

- Le interfacce vengono anche dette “contratti”, perché impongono un obbligo sulla classe che sceglie di implementarle.

implements

- Modifichiamo Person obbligandola ad avere il metodo isValid()
  public class Person implements Validable {

- Mentre una classe può estendere direttamente solo un'altra classe (single inheritance), una classe può implementare tutte le interfacce che vuole (multiple implementations).

- Con la parola chiave implements Person si dichiara un sottotipo di Validable (Person instanceof Validable) e deve garantire di avere il metodo isValid(). Se la classe C implementa l'interfaccia I, la classe C deve fare override di tutti i metodi di C.

- Tuttavia, essendo Person astratta, può delegare l'implementazione ai suoi sottotipi, come se avesse definito lei stessa un metodo astratto. *L'obbligo di implementazione ricade solo sui tipi concreti.*

extends vs implements

- la classe C2, concreta, estende C1, concreta. public class C2 extends C1 - non ci sono obblighi di implementazione.
- la classe C2, concreta, estende A, astratta. public class C2 extends A. Se A conteneva metodi astratti m1(), m2(), C2 deve farne override. C2 non può estendere direttamente altre classi.
- la classe C2, concreta, implementa l'interfaccia I. public class C2 implements I. C2 deve fare override di tutti i metodi di I (da 0 a n metodi). Può implementare anche altre interfacce, e quindi è tenuta a fare override anche di quei metodi.
- la classe A, astratta, implementa I. public abstract class A implements I. La classe A non ha obblighi di implementazione, ma i suoi sottotipi concreti dovranno implementare, o ricevere da altri antenati, tutti gli eventuali metodi astratti di A e tutti quelli di I. La classe A tuttavia può implementare i metodi di I se il programmatore lo decide.

Un esempio formale

- public class C extends A implements I1, I2

  C c = new C();

- C è concreta, A, I1 ed I2 sono astratti.
- La classe C deve fare override di tutti i metodi astratti in C, I1 e I2
- L'oggetto puntato dalla variabile c appartiene ai tipi Object, C ed A per ereditarietà e ai tipi I1, I2 per implementazione.
- Significa che l'oggetto deve fornire i servizi definiti in Object, in A, in C, in I1 e in I2. Come questi servizi vengono forniti è definito nella classe C.

- Gli obblighi di implementazione non riguardano i tipi astratti ma solo quelli concreti, e vengono passati lungo la genealogia da supertipo a sottotipo.

Il senso di onorare un contratto

- L'obiezione tipica è "che senso ha ereditare un obbligo"?
- La risposta rapida, che vedremo meglio lavorando su liste, vettori, mappe e set fra poco, è che definire un obbligo permette di avere la sicurezza che un determinato oggetto avrà un determinato metodo.
- Il compilatore non ci permetterà di richiamare un metodo senza la sicurezza che ci sia sull'oggetto di quel determinato tipo ("type safety"). Questo ci permette di essere sicuri che avremo quello che ci serve in fase di scrittura, anche se non ci mette al riparo da eventuali errori nell'implementazione. Siamo sicuri di "avere i pezzi", non siamo sicuri che i pezzi funzioneranno.
- Le interfacce sono inoltre molto flessibili. Posso scegliere di imporre una interfaccia a classi non imparentate fra loro dal rapporto di ereditarietà, e in effetti è questo il caso comune. Possiamo anche combinare fra di loro più interfacce secondo necessità.
- Allo stesso tempo, una interfaccia può avere molte implementazioni - molti modi diversi di fornire lo stesso servizio.

Video 18 - Aggiunta di Validable all’entità Person

Proprietà nelle interfacce

- Una interfaccia non può definire proprietà di oggetto, ed è una differenza essenziale e permanente nella struttura di Java.

- Le variabili definite nelle interfacce sono implicitamente public final static, e sono quindi costanti di tipo. Non riguardano l'oggetto della classe che implementa l'interfaccia, ma l'interfaccia stessa.

Interfacce in Java 8

- Java 8 ha marcato una grossa evoluzione nella struttura delle interfacce permettendo di scrivere dei metodi concreti, statici o di oggetto (“default”) direttamente nelle interfacce.

- In questo modo una interfaccia può conferire a una classe, e indirettamente a un oggetto, dei metodi concreti, già implementati, oltre a quelli astratti. I metodi concreti dell’interfaccia possono inoltre richiamare i metodi astratti, con lo stesso meccanismo visto per le classi astratte.

Metodi static nelle interfacce

- Sono analoghi ai metodi static delle classi, e possono essere richiamati direttamente sull’interfaccia.

- Sono pensati per lavorare sui parametri che vengono loro passati, non essendoci uno stato di oggetto su cui lavorare.

Metodi di default

- Un metodo di default è un normale metodo di oggetto che però viene scritto nell'interfaccia, e viene ereditato dalla classe che implementa l'interfaccia.

- Può essere sovrascritto. In effetti "default" significa precisamente "per difetto". E' una prima implementazione, ma non necessariamente l'ultima. Non è final.

- L'interfaccia non dispone di uno stato di oggetto, e quindi il metodo di default, pur appartenendo all'oggetto, è limitato a richiamare altri metodi di default o static dell'interfaccia, o i metodi di Object, che sono comuni a tutti i tipi.

Le interfacce come tipo

- Una interfaccia definisce un tipo di variabile, e può essere usato come tale:
  Validable v = new Teacher();

- Si tratta di un tipo solo astratto, è il tipo della variabile, del “contenitore”, non dell’oggetto, del “contenuto”.

- Potrò invocare v.isValid();. Non potrò invocare v.getYearlyRetribution(); Se vogliamo conoscere il nome dell’oggetto in v, dovremo castarlo, almeno a Person:
  Person p = (Person) v;
  Console.print(p.getName());

- Come per gli esempi precedenti, il tipo di variabile definisce cosa possiamo fare, mentre l’oggetto definisce come verrà fatto. isValid() di Teacher sarà diverso da quello di Student.

Conclusioni - what and where

Nelle classi concrete: metodi concreti, proprietà di oggetto, metodi di classe, proprietà di classe.

Nelle classi astratte: metodi concreti, proprietà di oggetto, metodi di classe, proprietà di classe, metodi astratti (di oggetto).

Nelle interfacce fino a Java 7: metodi astratti, proprietà final static

Nelle interfacce da Java 8 in avanti: metodi astratti, proprietà final static, metodi static, metodi di default (di oggetto)

Il discorso non è completo

- Ci sono forme più raffinate di astrazione che andranno affrontate in seguito. Prima però ci concederemo un breve detour nel lavoro su collezioni di valori (insiemi) e sulle *strutture dati* per gestirle.

- Parlando di strutture dati, vedremo che le interfacce sono ovunque, e definiscono le regole di uso delle stesse.

Conclusioni

- Java ci permette di separare nettamente quello che desideriamo (interfacce, classi astratte) dal modo in cui lo otteniamo (classi concrete, e successivamente oggetti).
- Tanto le classi astratte quanto le interfacce definiscono dei tipi, ma non possono essere usate per creare direttamente oggetti. Servono come base per l'ereditarietà o l'implementazione.
- Java 8 innova pesantamente nelle interfacce rendendole più simili alle classi astratte e permettendo di fornire metodi già implementati, ma manca il concetto di stato di un oggetto definito tramite la sua classe, astratta o concreta che sia.
- Molto spesso il tipo formale (tipo della variabile) è astratto, mentre il tipo dell'oggetto creato (col new) è sempre concreto.

9 - Lavorare su insiemi

La necessità di lavorare su insiemi

- Fino ad ora abbiamo lavorato principalmente su singoli oggetti. Nella pratica, è una evenienza molto rara.

- E’ necessario saper lavorare su collezioni, o più genericamente, su insiemi di n oggetti dello stesso tipo o di tipi diversi. E’ necessario saperli *filtrare*, *mappare*, *ridurre* a singoli valori e in generale saperci ragionare sopra.

- Dovremo saper lavorare sulle *strutture dati*.Una struttura dati è, informalmente, un insieme di variabili legate fra di loro, sul quale abbiamo definito delle operazioni (dei metodi) e una logica di funzionamento che governa l’insieme delle variabili, e non la singola variabile.

La nostra prima struttura dati - il vettore

- Definiamo vettore **una lista ordinata di dimensione finita di elementi dello stesso tipo**.

- Riconosciamo tre momenti potenzialmente separati nell'uso di un vettore: la dichiarazione, la creazione e il riempimento.

- Un vettore è un tipo particolare di oggetto, e in effetti ha proprietà e metodi propri, e può essere null.

Dichiarazione di un vettore

// ho dichiarato una variabile di nome v, di tipo "vettore di interi"

// le quadre servono a indicare a Java che stiamo creando un vettore

// e non un singolo intero

int v[ ];

v è conterrà un vettore di interi. int v [ ], o int [ ], è equivalente, non si legge “intero”, si legge “vettore di interi”, impropriamente “lista”. v conterrà interi, ordinati.

Creazione di un vettore

v = new int[3];

ORA esiste il vettore. Prima esisteva solo una "scatola vuota" , vale a dire, null, e il compilatore mi avrebbe perfino impedito di usarla .
Ora esiste una "impalcatura", una struttura che contiene tre scatole, che a loro volta conterranno degli interi.

Le scatole sono numerate: 0,1,2. Si comincia a contare da 0. Il numero della scatola viene detto “indice”

Graficamente:
vettore v

indice della scatola	0	1	2
contenuto	0	0	0

Riempimento del vettore

v[1] = 5;

Terza fase: riempimento.
Ho messo il valore 5 alla posizione 1, vale a dire la seconda posizione del vettore. Ho riempito la scatola centrale. Le altre due restano col loro valore di default, che nel caso degli interi è 0.

Graficamente:
vettore v

indice della scatola	0	1	2
contenuto	0	5	0

Algebricamente, v = [0, 5, 0], o v =[0=>0, 1=>5, 2=>0], con la sintassi indice=>valore.

Accedere ai singoli elementi del vettore

- Possiamo accedere ai singoli elementi del vettore tramite la loro posizione all'interno della lista. Questo numero, lo ripetiamo, viene detto **indice dell'elemento nel vettore** .

- String[ ] names = new String[3];
  names[1] = "George";
  String greeting = "Hello "+names[1]; // names[0] e names[2] sono oggetti e sono al loro valore di default, vale a dire null.

- names è [null, "George", null]

  Creare il vettore non significa creare il suo contenuto. Il vettore è uno "scaffale", il contenuto è costituito dagli oggetti che ci metteremo dentro. In totale abbiamo due oggetti - il vettore names e la stringa George, posta alla posizione 1 del vettore.

Tipo vettore, tipo del contenuto

- Vale per i vettori, come per tutto il resto, la regola di concordanza dei tipi. names = "George" produce un errore - names NON è una String, è un vettore di String.

- names[1] = 5; produce un errore. L'elemento names[1] è una String, mentre 5 è un intero. A questo si aggiunge un limite dovuto alla natura finita e fissa dei vettori. names[3]="Pino" produrrà un errore. Gli elementi del vettore hanno posizione (indice) 0, 1 e 2, perchè cominciamo a contare da 0. names[3] produrrà ArrayIndexOutOfBoundsException, letteralmente, siamo "fuori" dal vettore. Non esiste quel ripiano nello scaffale.

Creazione e riempimento del vettore

- In alcuni casi possiamo creare un vettore e riempirlo allo stesso tempo. E' il caso del metodo *split*. split è un metodo dell'oggetto di classe String che divide la String in base a un pattern e restituisce un vettore di String.

- String[] parts = "10/02/2020".split("/"); //parts è ["10", "02", "2020"], con posizioni 0, 1 e 2 rispettivamente;
  Il metodo split riceve in ingresso una stringa separatore, e splitta la stringa di riferimento (10/20/2020) dove trova il separatore.

  Ma dove è il new? Dove è il dimensionamento del vettore? Vengono eseguiti da split, in maniera trasparente. Ci sono, ma non si vedono. split è usato molto spesso, e lo useremo in seguito, lavorando su file di testo.

- Un altro modo di riempire un vettore è la seguente forma contratta, che dichiara, crea e riempie un vettore: int v[] = new int[]{6, 8, 10}; Il vettore è stato dichiarato, creato e riempito. Il contenuto è quello che abbiamo messo fra le graffe, quindi sarà lungo 3, e i valori sono 6,8 e 10 alle posizioni 0,1 e 2 rispettivamente

Scorrimento di un vettore

- Una categoria piuttosto ampia di problemi (detti lineari) possono essere ricondotti a un'operazione di scorrimento di un vettore, vale a dire al ciclare tutti gli elementi, partendo tipicamente dal primo e arrivando all'ultimo, di solito in sequenza.

- E' estremamente intuitivo. Si parte con l'elemento 0, poi l'1, il 2... fino ad arrivare all'elemento vettore.length-1 (l'ultimo), e su ogni elemento viene eseguita una qualche operazione.

  Di solito per farlo si utilizza un ciclo for, che dispone degli strumenti per mandare automaticamente avanti l'indice, facendoci così passare da una "cella" all'altra del vettore, da un ripiano all'altro dello scaffale.
  Nella sua forma più basilare, lo scorrimento può essere scritto così:

```
for(int i=0;i <vettore.length ; i++)
        // fai qualcosa con vettore[i]
```

i è una contrazione di “indice”, e stiamo accendo a tutti gli elementi tramite il loro indice, uno per volta.

Esempi di scorrimento - stampa del vettore

Prendiamo int v[] = new int[] {40, 30, 20, 25, 17, 18, 21, 34}; come esempio, e vediamo una serie di applicazioni dello scorrimento. Si consiglia di studiarle tutte perché trovano spessissimo applicazioni nella pratica:

```
//Stampa del vettore
for( int i=0;i <v.length;i++)
        Console.print(v[i]);
```

Esempi di scorrimento - somma di un vettore

Questo codice somma tutti i numeri del vettore v:

```
int sum = 0;
for( int i=0;i <v.length;i++)
        sum+=v[i];
```

Questo codice somma tutti gli elementi del vettore, partendo da v[0], il primo, fino all'ultimo, v[v.length-1]. E' un esempio di *riduzione*, vale a dire di trasformazione di un insieme in un singolo elemento.

Esempi di scorrimento - media di un vettore

```
int sum = 0;
for( int i=0;i <v.length;i++)
      sum+=v[i];

// media
int avg = sum / v.length;
```

Notiamo che la media si calcola fuori dal ciclo di scorrimento. Prima la somma, poi il calcolo della media.

Esempi di scorrimento - conteggio dei numeri superiori a 10

```
//Conteggio dei maggiorenni
int res = 0;
for(int i=0; i< v.length;i++)
        if(v[i]>10)
                res++;
```

Ogni volta in cui troviamo un elemento che rispetta la nostra condizione (v[i] > 10) aumentiamo di uno il valore di res. Anche questa è una forma di riduzione, ed è molto comune. In seguito potremmo contare, ad esempio, tutti gli insegnanti con più di 10 anni di esperienza.

Esempi di scorrimento - ricerca del massimo

```
 //ricerca del massimo
int max = v[0];
for(int i=1;i max)
        max=v[i];
```

Partiamo prendendo il primo valore come massimo, e poi confrontiamo i massimi trovati via via col valore precedente. E' l'ennesima forma di riduzione. Un esercizio utile è calcolare il minimo.

Per concludere, notiamo che i è *locale al for*. Non esiste fuori.

I vettori come insiemi e le operazioni su insiemi: Map, Filter e Reduce

- I vettori sono un modo primitivo e scomodo ma utile didatticamente di trattare il concetto di "insieme". Non trattiamo più un singolo intero, una singola String e neanche un singolo oggetto ma un insieme arbitrariamente lungo di elementi dello stesso tipo.

- Come abbiamo visto, i cicli e in particolare lo scorrimento sono gli strumenti principe per trattare questa tipologia di dati. Un vettore NON corrisponde al concetto matematico di insieme (quello semmai è Set, che vedremo in seguito), ma ci permette di cominciare a ragionare sulle problematiche relative.

- In particolare nella pratica ci capiterà spesso di dover trasformare insiemi, di dover filtrare insiemi e di dover ridurre insiemi a singoli numeri o elementi. Queste operazioni possono essere chiamate, con una certa approssimazione: map, filter e reduce.

L’operazione di mappatura, Map o Mapping

- **Mapping**: partiamo da un vettore o una collezione O (**originale**), di dimensione n, e otteniamo una seconda collezione T (**trasformata**), sempre di dimensione n.
- Attenzione a non confondere questa operazione con la struttura dati Map, che vedremo in seguito.
- L’idea è quella di trasformare un insieme in un altro insieme di uguali dimensioni, applicando la stessa **trasformazione** a tutti gli elementi.
- Un esempio concreto: trasformare un vettore di Teacher in un vettore di String, contenenti solo i loro nomi e cognomi. Non vogliamo fornire tutti i dati ai richiedenti, ma solo quelli strettamente necessari.

Esempio di mappatura: da vettore di Teacher a vettore di String

- // Insieme di origine
  Teacher [ ] teachers = new Teacher[] { new Teacher(....), new Teacher(...), new Teacher(…) };

- // Insieme trasformato. Ha la stessa dimensione dell'originale.
  String[ ] names = new String[teachers.length];

- for(int i=0;i<teachers.length;i++)
  names[i] = teachers[i].getName()+" "+teachers[i].getSurname();

- La trasformazione è quella che trasforma un Teacher (teachers[i]) in una stringa (teachers[i].getName()+" "+teachers[i].getSurname()).

- Matematicamente, la trasformazione è una funzione con dominio Teacher e codominio String: f(Teacher)->String.

il ciclo for avanzato - *foreach*

- Lo scorrimento è una operazione estremamente comune, al punto che disponiamo di un ciclo specializzato, non visto fino ad ora, per eseguirlo. Si chiama for avanzato, o for each, e si scrive come:

  for(Tipo *ago*:*pagliaio*) {}

  dove pagliaio è un vettore, o una collezione, i cui elementi sono dello stesso tipo di ago. Vediamo un esempio pratico:

```
int v[] = {5, 6, 1}
for(int i=0;i<v.length;i++)
{
        int e = v[i];
        Console.print(e);
}
```

  Equivale a:

```
int v[] = {5, 6, 1}
for(int e:v)
        Console.print(e);
```

L’operazione di riduzione - Reduce

- Abbiamo già visto esempi di riduzione, “comprimendo” un vettore in un singolo valore. La riduzione si può applicare a qualunque tipo di insieme, e può produrre più o meno qualunque cosa.

- Svilupperemo un esempio di riduzione, calcolando il costo totale sostenuto da una scuola per le sue Persone, seguendo l’esempio precedente. Ne approfitteremo anche per illustrare un caso da manuale di polimorfismo.

Esempio di riduzione - il costo totale sostenuto da una scuola per le sue Person

- Cominciamo definendo un vettore di persone, e riempiendolo con vari sottotipi di Person. Un vettore Person[] è un vettore che potrà contenere oggetti Person di qualunque sottotipo di Person.
  Person[ ] people = new Person [new Teacher(...), new Student(...), new Support(..), new Teacher(...),....];

- Notiamo che Person è astratto, quindi il contenuto di people sarà necessariamente costituito da sottotipi **concreti** di Person (non Employee, ad esempio).

La sicurezza di avere getCost()

- people è di tipo Person[], vettore di Person, e Person è astratta. I suoi elementi sono sottotipi di Person.
- getCost() è definito, astratto, nella classe Person. Di conseguenza, tutti gli oggetti generati dalle sue sottoclassi concrete avranno il metodo getCost().
- Possiamo usarlo allo stesso modo su tutti gli oggetti contenuti in people. Ogni oggetto potrebbe calcolarlo a modo suo, ma tutti forniranno il costo della persona per la scuola.
- Questo viene detto late binding: non conosciamo il tipo concreto degli elementi in people fino al momento dell'esecuzione, ma possiamo essere comunque sicuri che funzioneranno secondo le regole che abbiamo stabilito nel tipo Person.

Implementazione della riduzione

L’implementazione è molto semplice una volta capita la teoria dietro:

```
int res = 0;
for(Person p:people)
        res+=p.getCost();
```

p sarà di volta in volta un Teacher, un Support, uno Student o anche una *qualunque sottoclasse che ancora non abbiamo scritto*.

L’operazione di filtro - filter

- L’operazione di filter, filtraggio, o estrazione di sottoinsieme, produce un sottoinsieme S a partire dall’insieme originale O. Gli elementi del sottoinsieme S sono dello stesso tipo di quelli di O, e non vengono trasformati, a differenza di quanto accade con la mappatura.

- Matematicamente, un filtro è una funzione che va da un insieme O di elementi e, con | O | = n, a un insieme S di elementi e con | S | = m, con m<=n.

- Un esempio concreto: estrarre tutte le donne dall’elenco delle Person visto in precedenza.

Una nuova struttura dati - la List

- I vettori sono strutture a lunghezza fissa. Prima di estrarre il nostro target, dobbiamo contarli per sapere di quanto spazio avremo bisogno nel vettore. Il vettore è in effetti una struttura molto poco duttile per le operazioni di filtro.

- Possiamo introdurre una nuova struttura dati, più duttile, per facilitarci nel lavoro - la List.

La struttura dati List

- Una lista può essere vista come una evoluzione di un vettore. In termini pratici, una lista è una sequenza ordinata di elementi dello stesso tipo, ma a differenza del vettore non è necessario dimensionarla: la lista nasce vuota e può espandersi o contrarsi a seconda delle necessità.

- La sintassi per creare una lista, in questo caso di stringhe, è la seguente: List names = new ArrayList(); I simboli < e > vengono detti "parentesi angolari", in questo caso, e servono a indicare il tipo dell'elemento della lista. In questo caso, la lista contiene oggetti di tipo String. Ci sono alcune precisazioni da fare prima di proseguire.

Approfondimenti sulle List

- Non esistono liste di tipi primitivi. Non posso scrivere List < int >. Le liste contengono solo oggetti . Vale ogni volta in cui useremo le parentesi angolari.

- List < > è il tipo formale della lista, e useremo sempre questo. Per la precisione, List è una interfaccia che definisce i metodi che tutte le liste dovranno fornire al mondo.

- ArrayList è il tipo concreto, un caso particolare di lista. I due tipi non sono identici ma sono compatibili . Tecnicamente diremo che un ArrayList è un List.
-
- Come per i vettori, le liste sono oggetti, con metodi e proprietà. names è un oggetto di tipo List < String >, names.get(0) è il primo elemento della lista, ed è di tipo Person

Tipi boxati

- E se volessimo liste di interi? List <int> non funziona, come abbiamo visto, ma possiamo usare il *boxato* di int, Integer.
- Un tipo boxato, o wrapper, è un oggetto Java che contiene un unico valore corrispondente a un tipo primitivo. Ad esempio, Integer a = 5; contiene un valore intero, ma è un oggetto della classe Integer. In effetti è equivalente a Integer a = new Integer(5);
- Per creare List, e successivamente Set o Map, contenenti valori di tipo intero, scriveremo List<Integer>.
- I tipi boxati sono Byte, Short, Integer, Long, Float Double, Character e Boolean, uno per ogni tipo primitivo. Non esiste un boxato di String, perché String non è un primitivo, anche se lo sembra.
- I boxati possono essere convertiti automaticamente nel rispettivo tipo primitivo, e viceversa. Si parla di boxing e unboxing:
  int b;
  Integer a = 5;
  b = a;
  b alla fine conterrà 5, come valore primitivo (unboxing), pur avendolo preso dall'oggetto a, mentre con Integer a = 5; abbiamo avvolto (boxing) un valore primitivo in un oggetto.

Uso dei tipi boxati

I tipi boxati servono per:

- essere usati nelle strutture dati che ammettono solo oggetti (List, Set, Map, altre)

- contenere dati di utility, tipicamente come metodi di classe. Ad esempio, il metodo Console.readInt() usa il metodo di classe Integer.parseInt(x) per convertire una stringa lettera in input in un numero.

-

Confronto fra liste e vettori

Creazione
Person v = new Person[2];
List<Person> l = new ArrayList ();

Inserimento di un elemento
v[0] = george;
l.add(george);

Inserimento di un secondo elemento
v[1] = clare;
l.add(clare);
Notiamo che non serve specificare la posizione. Il metodo add crea automaticamente lo spazio e vi inserisce, in questo caso, l'oggetto Clare.

Inserimento di un terzo. Notiamo che il vettore darà errore, la lista no:
v[2] = luke;
l.add(luke);

Accedere al primo elemento:
v[0];
l.get(0); // il metodo get permette di prendere un elemento data la posizione. La lista è ordinata come il vettore, partendo da 0.

Trovare le rispettive lunghezze. length non cambia, size() sì
v.length;
l.size();

Torniamo all’esempio del filtro

Vogliamo partire da un insieme di Person, e ottenerne un altro. Il primo conterrà tutte le Person, il secondo solo le donne. Non stiamo mutando gli elementi, ma estraendo una collezione che avrà solo alcuni di quelli iniziali.

Possiamo partire dal vettore originale, people, e ottenere una List<Person> women, in maniera molto semplice

Implementazione del filtro con vettore e List

```
// Insieme originale, O, un vettore
Person[ ] people = new Person [ new Teacher(...), new Student(...), new Support(.. ), new
Teacher(...),.... ];

// Sottoinsieme filtrato, S, una List
List<Person> women = new ArrayList<Person>();
// Scorrimento e test, per decidere quali elementi prendere o lasciare
for(Person p:people)
        if(p.getGender().equals("F"))
                women.add(p);
                // la lista women era vuota. Ogni elemento di cui facciamo add
                // crea lo spazio necessario e aggiunge la persona creata
```

Video 19 - Filtro, mappatura e riduzione

Esercizio sulle operazioni sugli insiemi

Partendo dai dati dell’esercizio precedente:

- creare un sottoinsieme di Person contenente solo gli insegnanti
- creare una lista di String contenenti nomi e cognomi degli insegnanti trovati
- calcolare il totale delle retribuzioni annue degli insegnanti trovati
- stampare i risultati a schermo

Problemi di liste e vettori

- Liste e vettori sono entrambi scorribili tramite foreach e condividono il concetto di insieme ordinato. Hanno un'altra caratteristica comune: ammettono duplicati.

- Posso inserire più volte lo stesso elemento nello stesso vettore o nella stessa lista. Non sempre questo è desiderabile. Supponiamo ad esempio di voler memorizzare una lista di interessi, rappresentati come stringhe. Non avrebbe senso aggiungere due volte lo stesso elemento nello stesso elenco, dovremmo impedirlo. Ci è anche indifferente conoscere l'ordine degli elementi.

- Per ragioni simili, potrei voler ricavare i singoli nomi di battesimo da una lista, eliminando quindi i duplicati.

La struttura dati Set

- La soluzione a questa tipologia di problema è la struttura dati **Set** , che corrisponde al concetto matematico di insieme in maniera rigida.

- Un insieme matematico non ha duplicati e non deve necessariamente avere un ordine (non tutti gli insiemi hanno una relazione d'ordine di interesse). La sintassi per la sua creazione somiglia a quella della lista.

Creazione e primo esempio di un Set

```
Set<String> interests = new HashSet<String> ();
interests.add("G1");
interests.add("G1");
interests.add("A");
// stamperà 2
Console.print(interests.size());
for(String i:interests)
        Console.print(i); // output: A, G1. Non ha rispettato l'ordine di inserimento
```

Il set ha ignorato il secondo inserimento di G1. Per il resto somiglia abbastanza a una lista - ammette solo oggetti, non primitivi, ed è a dimensione variabile. Possiamo aggiungere o rimuovere elementi, ma dobbiamo ricordarci che manca il concetto di ordine. Non ha senso prendere “il primo elemento del set”, e infatti usiamo il for each per scorrere i suoi elementi.

Approfondimento dei Set

- Come visto in precedenza Set è il tipo formale, HashSet quello concreto.
- HashSet è una classe che implementa Set, e quindi è compatibile con Set (si rimanda al capitolo sull'ereditarietà per i dettagli), mentre non è vero il contrario: tutti gli HashSet sono Set, ma non tutti i Set sono HashSet.
- Una operazione estremamente comoda, presente sia sulle List che sui set, è il metodo contains(). Contains restituisce true o false a seconda che un elemento sia presente o meno in un insieme:

```
interests.contains("A"); // true
interests.contains("B"); // false
```

Questi metodi sono subordinati alla presenza di un override per equals e, nel caso degli HashSet, di hashCode, nel tipo dell'elemento del Set. In questo caso, in String. In mancanza di quegli override, o nel caso di una scrittura non conforme, il metodo contains rischia di non funzionare.

Set possiede inoltre metodi adatti a simulare le operazioni insiemistiche di base (intersezione, unione, differenza).

Video 20 - Estrazione dei valori univoci da un vettore di String

Video 21 - Eliminazione dei duplicati con e senza equals e hashCode

Struttura dati Map

- Da non confondersi con l'operazione di mappatura (mapping, o map), per quanto ci sia un ragionamento comune sotto.
- Le mappe (struttura Map) definiscono una struttura dati potente che ricorda in qualcosa un incrocio fra classi e liste.
- Una Map è definita come un *insieme di coppie chiave-valore , con chiave di tipo K e valore di tipo V, potenzialmente diversi*. Si prega di mandare a memoria questa definizione, che non è intuitiva e che viene spesso chiesta ai colloqui.
- Ogni mappa esprime il concetto matematico di funzione , mettendo in relazione una chiave k con un valore v. Le chiavi k non si possono ripetere, sono un Set, mentre i valori sì. La sintassi è: Map<K,V> m = new HashMap<K,V> (). Sì, nel caso di una Map dovete tenere conto di due tipi, quello della chiave e quello del valore.
- Le Map **non** sono una collezione sensu strictu, non in Java almeno, ma sono estremamente diffuse e necessarie per la pratica.

Introduzione ed esempi di uso

Map m<K,V> = new HashMap<K,V> ();

- Map < K, V > è il tipo formale. HashMap è il tipo concreto. HashMap è una classe che implementa l'interfaccia Map, ed è compatibile con Map, ma non viceversa. Tutte le HashMap sono Map ma non tutte le Map sono HashMap.

- Cerchiamo di chiarire con un esempio:

```
Map<String,String> p = new HashMap<String,String> ();
// Mappa, insieme di coppie. In questo caso, K = String, V = String.
// non posso inserire un valore per volta, ma sempre e solo una coppia. put è come dire "add", ma aggiungo una coppia.
// la mappa, come la lista, ha dimensioni variabili, ma contiene sempre e solo coppie, in questo caso, coppie di String
p.put("name", "George");
p.put("surname","Romano");
p.put("age", "40");

// e per leggere?
// "stampa il valore della chiave age nella mappa (contenitore) p"
// anche se è un numero, lo vediamo lo stesso come una stringa
Console(p.get("age")); // stamperà 40
```

Notiamo la differenza. Con una lista, avrei scritto qualcosa del tipo lista.get(0), per prendere il primo elemento. Con un vettore, vettore[0]. Con un Set non avrebbe avuto senso parlare di "primo elemento". Nel caso della Map, per risalire all'elemento devo usare la chiave. In questo caso, la chiave è age, e p.get("age"); si legge "prendi il valore v associato alla chiave k di valore age".

Mappe e oggetti

- La mappa si può rappresentare come segue: {"name":"George", "surname":"Romano", "age":"40"}, con la rappresentazione chiave:valore.

- Potrebbe ricordarvi un oggetto, ma non è esatto. In un oggetto, age sarebbe un int, mentre qui deve necessariamente essere una stringa, avendo dichiarato V = String. Le coppie sono sempre e solo {String, String} in questo caso.

- C'è anche un'altra differenza fondamentale rispetto a un oggetto: le classi dichiarano tutte le proprietà dell'oggetto in fase di creazione, e quindi sono sempre presenti, e al massimo vuote, e non possono dichiararne altre . L'oggetto Person ha solo quelle proprietà (name,surname,dateofbirth,gender, ad esempio), e non posso aggiungerne, mentre posso scrivere tutte le chiavi che voglio in una mappa.

- Un'altra grossa differenza rispetto agli oggetti è che non posso definire metodi extra all'interno delle mappe. Le mappe hanno i propri metodi - noi possiamo aggiungere coppie, non metodi.

Map<String,String> e primo esempio operativo di Map con valori complessi

- La mappa che abbiamo visto sopra, Map < String, String >, è piuttosto comune e molto usata nella pratica, perchè è adatta a rappresentare lo stato di una ampia categoria di oggetti.

- Molto spesso gli oggetti "viaggiano" sotto forma di Map, per essere poi ricostruiti. Noi abbiamo usato le stringhe, ma il principio è lo stesso. Le applicazioni però sono svariate. Prendiamo il seguente esempio:

  Map<String,Person> squadra = new HashMap (); // K = String, V = Person.

- Questa mappa esprime una relazione fra una stringa e una o più persone. Immaginiamo di avere le persone george, stella e pino (oggetti di tipo Person, non stringhe). George è portiere, pino è terzino sinistro e stella è terzino destro. George fa anche da allenatore. In Java, si tradurrebbe come:

- squadra.put("portiere", george);
  squadra.put("terzino sinistro", pino);
  squadra.put("terzino destro", stella);
  squadra.put("allenatore", george); // la mappa squadra esprime il rapporto fra una stringa (un ruolo) e un Person

- notiamo che george è collegato sia alla chiave “portiere” che a quella “allenatore”.

Esempio operativo - parte seconda

E se ora volessi sapere quanti anni ha il portiere? La sintassi sarebbe:

squadra.get("portiere").getAge();

squadra è la mappa (l'insieme delle coppie), get("portiere") prende il valore corrispondente alla chiave "portiere".

Il valore corrispondente è l'oggetto george. Quindi squadra.get("portiere") restituisce un oggetto di tipo Person, o, in gergo, squadra.get("portiere") instanceof Person.

Su quell'oggetto eseguo il metodo getAge(), che mi aspetto di avere in Person, producendo un intero.

In definitiva, il nostro scopo è organizzare i valori secondo le chiavi. Come dicevamo prima, le chiavi non si possono ripetere (in effetti le chiavi sono memorizzate in un Set di K). Non potremmo avere due persone con lo stesso ruolo. Ipotizziamo che arrivi un nuovo allenatore, a fianco del precedente:

squadra.put("allenatore", vlad);

L'oggetto george non è più collegato alla chiave allenatore, ma sarà ancora presente come portiere nella mappa squadra.

Video 22 - Analisi delle frequenze delle lettere di una parola usando le Mappe

Esercizio sulle Map - calcolo della frequenza dei nomi inseriti dall’utente

- Scrivere un main che preveda un ciclo principale in cui l’utente inserisce un nome.
- Se il nome è già presente in una Map<String,Integer>, aumentare il conteggio di 1, altrimenti impostare la coppia <nome,1> all’interno della mappa.
- Ripetere finché l’utente non inserisce una stringa vuota.
- Stampare la mappa a ciclo concluso.

Conclusioni

- Tipicamente lavoriamo su insiemi di oggetti, non solo su oggetti singoli.
- Un vettore è una lista ordinata di valori dello stesso tipo, potenzialmente anche primitivi, di lunghezza fissa. Tutti gli altri insiemi presentati sono di lunghezza variabile e possono contenere solo oggetti. I duplicati sono ammessi.
- Una List può essere vista come una versione dinamica del vettore: ordinata, duplicati ammessi, lunghezza variabile, valori dello stesso tipo, ma ammette solo oggetti, non primitivi.
- Un Set è un insieme di oggetti dello stesso tipo, non ammette ripetizioni e non garantisce il concetto di ordine.
- Una Map è un insieme di coppie chiave-valore. Il tipo della chiave e del valore non devono per forza coincidere.

10 - Approfondimenti di programmazione a oggetti avanzata

Interfacce funzionali e lambda

- Esiste una categoria specifica di interfacce, nota come "funzionale". Una interfaccia funzionale è una interfaccia che dispone di un solo metodo astratto . Può avere metodi statici, metodi di default, costanti, ma al massimo un metodo astratto.

- Ne diamo un esempio scrivendo una interfaccia che definisca il servizio di rendering (graficazione, stampa) di una Person.

Video 23 - Interfaccia TeacherView e implementazioni in italiano e in inglese

Punti chiave della soluzione presentata

- Il metodo render(Teacher t) è astratto, e viene definito in maniera diversa in ogni classe implementante.

- Il metodo render(List<Teacher> teachers) è di default e richiama il metodo render, ma è concreto. L'interfaccia TeacherView è funzionale.

Lambda

- In determinati casi ci serve una implementazione usa-e-getta di una interfaccia funzionale. Vogliamo, al tempo stesso, fornire una implementazione dell’unico metodo astratto, crearne una classe anonima e istanziare un oggetto.

- Siamo in grado di fare tutto quanto allo stesso tempo usando le lambda, che sono uno strumento rapido ed elegante, per quanto difficile da usare, per esprimere una interfaccia funzionale.

Riadattare le implementazioni di TeacherView

- Ci serve un oggetto TeacherView usa e getta, che ha come metodo astratto render(Teacher). Ci basta implementare quello.

- Posso creare una implementazione usa e getta, in questo modo:

  TeacherView v = new TeacherView() { public String render(Teacher t) { return "Mr "+t.getName()+" "+t.getSurname()+" "+t.getSubject();}

  Questa sintassi crea una classe anonima che implementa TeacherView, e le fornisce l'unico metodo astratto, render. Il resto è già presente. Contestualmente, viene creato un oggetto di questa classe anonima, e viene collegato alla variabile v.

Soluzione con Lambda

- La soluzione vista sopra è universale, e può essere utilizzata per creare implementazioni usa e getta di interfacce e classi astratte, fornendo di volta in volta i metodi mancanti.
- Disponiamo però di una soluzione più elegante: forniamo solo il metodo astratto, per altro con una sintassi ridottissima, input->output, come segue:
- TeacherView v = t -> "Mr "+t.getName()+" "+t.getSurname()+" "+t.getSubject();
- Questa sintassi produce un oggetto di una classe anonima, e la riga presentata sopra è esattamente l'implementazione del metodo render. Essendo l'unico metodo astratto, Java sa che ci riferiamo a lui. Avendo in ingresso il parametro Teacher, la t viene compresa immediatamente come parametro di tipo Teacher. Quanto scritto dopo la freccia viene interpretato immediatamente come valore del ritorno, e infatti è di tipo String.
- **t -> "Mr "+t.getName()+" "+t.getSurname()+" "+t.getSubject();**
  equivale strettamente a
  public String render(Teacher t) { return "Mr "+t.getName()+" "+t.getSurname()+" "+t.getSubject();}
- Una Lambda è una maniera elegante e rapida di creare contestualmente una classe che implementa una interfaccia funzionale e contestualmente di istanziarne un oggetto, e dispone di diverse sintassi.

Un esempio integrato in Java

- Java dispone di una interfaccia funzionale generica (vedremo a brevissimo cosa vuol dire) di nome Predicate<X>.
- L'interfaccia funzionale Predicate definisce un metodo public boolean test(X x) che analizza un input di tipo arbitrario (qualunque cosa, in realtà) e produce un booleano - sì o no. Serve principalmente a filtrare.
- Le liste in Java (interfaccia List) dispongono di un metodo removeIf, che passa tutti gli elementi attraverso il "setaccio" di un predicate, eseguendo un filtro. Elimina dalla lista tutti gli elementi per cui il metodo test del predicate restituisce true.
- Useremo questi due componenti standard (il metodo removeIf e l'interfaccia funzionale predicate) per rimuovere i nomi più corti di 9 lettere da una lista.

Rimozione dei nomi corti da una lista tramite lambda - parte 1

- Cominciamo definendo una implementazione dell'interfaccia Predicate<String>, per capire chi scartare. Possiamo farlo con la sintassi estesa o con la sintassi di lambda, e il risultato sarà identico:

- ```
  Predicate<String> isShort = new Predicate<String>(){public boolean test(String s) {return s.length()<9;}};
  ```

- ```
  Predicate<String> isShort = s->s.length()<9;
  ```

Rimozione dei nomi corti da una lista tramite lambda - parte 2

- Proseguiamo creando la lista, e poi passando il predicato al metodo removeIf.

```
List<String> names = new ArrayList<String>();
names.add("George");
names.add("Stephanie");
names.add("Hannibal");
names.add("Josephine");
names.add("Christopher");
names.removeIf(isShort);
```

Risultati finali

- Il metodo removeIf ha invocato il metodo test() dell'oggetto isShort, di tipo Predicate<String>, su tutti gli elementi della lista a cui appartiene, e ha rimosso quelli per cui il test ha restituito true.

- La stampa di names a fine programma restituirà: [Stephanie, Josephine, Christopher]

- Molti metodi standard di Java accettano lambda, cioè oggetti che implementano interfacce funzionali, come parametri, e questo permette di semplificare molti task comuni.

Esercizio sulle lambda

- Creare una List di Person
- Inserirvi dieci Person con dati diversi.
- Filtrare le persone al suo interno usando una lambda, eliminando le Person di genere maschile.
- Stampare il risultato.

Spiegare, finalmente, le parentesi angolari.

- Ragioniamo brevemente su List, Set, Map e Predicate.

- Notiamo che abbiamo sempre scritto List < Person >, o List < Teacher > o List < String >. Abbiamo usato le parentesi angolari per decidere su cosa volevamo lavorare, e di volta in volta abbiamo definito un tipo diverso per il contenuto delle liste, dei Set, o per l'oggetto da testare, ma il funzionamento generale non cambiava .

- Richiamando List.add() inserivamo un elemento del tipo che avevamo scelto. Richiamando size() ottenevamo sempre un intero, il numero degli elementi presenti. Non ci interessava, per il suo funzionamento, di che tipo si trattasse.

Interfacce e classi generiche

- List è in effetti una interfaccia generica , o meglio, che usa il sistema dei generics in Java .

- Le classi e le interfacce generiche sono pensate per lavorare non su tipi specifici ma su tipi più o meno arbitrari . Una List infatti funziona su String, Teacher, Person... su qualunque oggetto in realtà.

- Scriviamo una classe o una interfaccia generica quando abbiamo individuato un comportamento che resta valido a prescindere dal tipo , o almeno nell'ambito di alcuni tipi ristretti che possiamo definire in fase di scrittura.

- Le operazioni di aggiunta, rimozione, conteggio, scorrimento, sono valide a prescindere dal tipo, e possono essere riunite in una interfaccia di nome "List". List ci permette di lavorare, allo stesso modo, su qualunque tipologia di oggetto.

Un esempio matematico: Couple<X,Y>

- Vogliamo una classe che rappresenti una coppia di valori, <X,Y>.
- X ed Y non devono necessariamente essere dello stesso tipo.
- Vogliamo essere in grado di invertire la coppia, quindi di andare da <X,Y> a <Y,X>, ma anche di vedere se due coppie sono fra di loro inverse.

L’implementazione di Couple<X,Y>

```
public class Couple<X, Y>
{
	X x;
	Y y;
	public Couple(X x, Y y)
	{
		this.x = x;
		this.y = y;
	}

	public Couple<Y,X> invert()
	{
		return new Couple<Y,X>(y,x);
	}

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

- In questa fase non conosciamo i tipi di X e di Y.
- Questa classe è parametrica, o “generica” - i tipi di X e di Y verranno acquisiti in fase di dichiarazione della variabile.
- I parametri in questo caso non sono oggetti, ma tipi.

Creare una coppia, invertirla, stamparla

```
// tipo di X String, tipo di Y integer
Couple<String,Integer> c = new Couple("George", 44);

Console.print(c.invert()); // 44  George
```

Il meccanismo di inversione, e qualunque altro meccanismo potremo definire sulle coppie, varranno quali che siano i tipi di X e Y.

Ma perchè non usare direttamente Object?

- Potremmo ottenere lo stesso effetto scrivendo semplicemente Object x, Object y, nell'esempio precedente, ma il sistema dei Generics ci permette di fare di meglio.

- Per prima cosa, possiamo verificare in fase di scrittura del codice di stare passando l'oggetto del tipo corretto alla classe. Con Object, andrebbe bene qualunque cosa - type checking, type safety.

- Secondariamente, possiamo imporre delle restrizioni ai tipi su cui vogliamo lavorare. Potremmo decidere che il tipo su cui lavoriamo deve appartenere a una determinata genealogia, avere un determinato antenato, e quindi disporre di un determinato metodo o proprietà, che potremo usare.

Una coppia di Person

- Posso imporre che i tipi di una classe (o di una interfaccia) parametrica siano oggetti di tipo Person, e abbiano quindi un metodo definito in Person di nome getAge().

- Sapendo che i miei oggetti, quale che sia il tipo concreto, sono Person, posso usare all'interno della classe dei metodi che non avrei a disposizione senza questa restrizione, ad esempio getAge(). E' una estensione della type safety - se dovessi .

- Useremo questa restrizione per avere getAge(), e getAge() per calcolare la differenza di età, impossibile senza avere questa certezza.

PersonCouple

```
public class PersonCouple<X extends Person, Y extends Person>
{
	X x;
	Y y;

	public PersonCouple(X x, Y y)
	{
		this.x = x;
		this.y = y;
	}

	public PersonCouple<Y,X> invert()
	{
		return new PersonCouple<Y,X>(y,x);
	}

	public int getAgeDifference()
	{
		return Math.abs(x.getAge()-y.getAge());
	}
}
```

- X extends Person, Y extends Person. Non potrò scrivere PersonCouple<String,Person>, perché String non estende Person.
- Potrò scrivere PersonCouple<Teacher,Support>, o anche PersonCouple<Support,Support> invece.
- Questo permette di calcolare la differenza di età fra le due persone indicate.

Conclusioni

- Possiamo scrivere classi, classi astratte e interfacce generiche. Generica vuol dire che lavorerà su tipi di valori non conosciuti al momento della scrittura del tipo, ma specificati da chi la userà.

- Una classe o interfaccia generica può specificare vincoli sul tipo che gestisce. Ad esempio, posso specificare che l'oggetto che gestisco sia un sottotipo Validable o Person, e quindi avrà isValid o getName() rispettivamente.

- List, Map e Set sono esempi di interfacce generiche. ArrayList, HashSet e HashMap sono esempi di classi generiche che implementano quelle interfacce.

- In determinati casi è comodo passare del codice come parametro di un altro metodo. A tale scopo disponiamo delle lambda, che passano in realtà un intero oggetto anonimo di una classe anonima scritta a partire da una interfaccia funzionale. Le lambda sono integrate nella struttura di molte classi comuni in Java, fra cui le List.

11 - Eccezioni

Definizione di eccezione

- Possiamo definire **eccezione** un "errore durante il flusso del programma", vale a dire una interruzione del normale flusso di esecuzione del programma, a seguito di una istruzione che non è riuscita a completare correttamente la sua esecuzione.

- Le eccezioni nascono dalle istruzioni, tipicamente dal richiamo di un metodo. Un metodo che può generare una eccezione può dirsi “caldo”, e andrà trattato con attenzione, come vedremo a breve. Vediamo un esempio:

- int a = Console.readInt();

- Il metodo statico readInt della classe Console potrebbe non riuscire a convertire il valore inserito dall’utente in un numero, e in quel caso segnalerà correttamente una NumberFormatException. E' un errore molto comune, sia parlando con l'utente che leggendo da file.

Gestire o propagare

- Di fronte a una eccezione, un metodo si trova di fronte a due possibilità: propagare l'eccezione, lasciando che sia il chiamante (o qualcun altro) a gestirla gestirla lui stesso. Console.readInt() sceglie di propagare l'eccezione, giustamente, cioè di notificare il problema al chiamante.

- readInt() deve informare il chiamante che la chiamata non è andata a buon fine, di modo che il chiamante possa decidere cosa fare. Supponendo di dover chiedere un numero all'utente, main() deve sapere che l'input non è andato a buon fine, di modo da poter ripetere la domanda.

- In questo caso readInt() **propaga** l'eccezione, mentre main() la dovrà **gestire**.

- **Il verificarsi di una eccezione non gestita da nessun sottoprogramma porterà a un crash del programma.**

Gestione delle eccezioni - try e catch

La gestione viene effettuata utilizzando una struttura **try-catch**, che somiglia in qualcosa a un if-else:

```
try
{
        int n = Console.readInt();
}
catch(NumberFormatException e)
{
        Console.print("Invalid number");
}
```

Il blocco try è il blocco di "esecuzione normale". Se va tutto bene, verrà eseguito l'intero blocco try e il catch verrà saltato, un po' come accade con if-else. Se invece si verifica una eccezione (e quella eccezione in particolare, come vedremo in seguito), il blocco try terminerà alla riga a cui si è verificata l'eccezione e verrà eseguito il blocco catch, che in questo caso si limita a stampare il messaggio "Invalid number".

Ma cosa è davvero una eccezione?

- Ma cosa significa quel catch(NumberFormatException e)? Il metodo readInt non si è limitato a segnalare un errore. Ha impacchettato i dettagli dell'errore creando un oggetto (NumberFormatException è un tipo, quindi e è un oggetto di classe NumberFormatException) che ha poi passato al chiamante (in questo caso main()).

- Notiamo che, passando col mouse sopra Console.readInt(), possiamo leggere “throws NumberFormatException”. Il metodo ha dichiarato la propria “pericolosità”, o meglio, la necessità di gestire questa casistica.

- In questo modo main() è "informato dei fatti" e può gestire al meglio l'eccezione. Due metodi sempre presenti nell'oggetto Exception (che non ha bisogno di chiamarsi e) e molto comodi sono e.printStackTrace(), che ricostruisce tutto l'iter che ha portato all'eccezione, ed e.getMessage(), che restituisce un messaggio di errore riassuntivo su quanto accaduto.

- In fase di sviluppo tendiamo a mettere e.printStackTrace() in tutti i catch, per capire davvero cosa è successo. La stack trace (in sostanza, l'elenco delle chiamate a metodo che ha portato all'eccezione) merita un discorso a parte, che faremo in seguito

Propagazione delle eccezioni e problema del gestore - un esempio concreto

- Vogliamo scrivere un programma che carichi delle parole da file, per la precisione un elenco di nomi, elimini i duplicati e risalvi l'elenco senza duplicati su un secondo file.

- A tale scopo creeremo un oggetto di nome "NameLoader", che aprirà il file e produrrà una lista di Stringhe coi nomi letti dal file, e un oggetto di nome "NameWriter", che scriverà il set di nomi che ricaveremo dalla lista.

- Ci sono dei problemi: - e se il file da cui vogliamo caricare non dovesse esistere? E se avessimo problemi in fase di scrittura?

Video 24 - Realizzazione e uso di NameLoader e NameWriter

Punti chiave del video

- FileReader propagava FileNotFoundException. NameLoader poteva scegliere se gestire o propagare - ha scelto, giustamente, di propagare.

- FileWriter propagava IOException. NameWriter ha dovuto scegliere se gestire o propagare. Ha scelto nuovamente di propagare.

- In ambo i casi, il gestore è stato il main(), che ha scelto come intervenire - chiedendo all'utente il nome di un file diverso da cui caricare, o in cui salvare.

throws

- throws è la parola chiave che usiamo dopo la firma del metodo e che serve ad avvisare il chiamante: invocare questo metodo potrebbe generare questa eccezione. Il metodo propaga quel tipo di eccezione.

- un metodo può fare throws di quante eccezioni desidera, e anche gestirle internamente in un punto e farne throw in un altro.

- le eccezioni possono essere a gestione o propagazione esplicita obbligatoria (eccezioni checked) o a gestione opzionale e propagazione implicita (unchecked). FileNotFoundException è checked - se richiamiamo un metodo che potrà produrre questa eccezione dovremo fare try-catch oppure propagarla a nostra volta (fare a nostra volta throws). Non siamo obbligati a farlo per NumberFormatException, che è unchecked.

Eccezioni multiple

Un blocco try potrebbe generare più di una tipologia di errore, e avere di conseguenza diversi blocchi catch, ciascuno con una propria gestione separata.
Prendiamo questo codice:

```
String s[] = new String[] {"123", "100", "60A", null};

for(int i=0;i <s.length;i++)
        Console.print("Digits:"+s[i].length()+", value:"+Integer.parseInt(s[i]));
```

Stampo la lunghezza del numero e poi il suo valore, parsato.

Questo codice può generare due tipologie di errore diverse: NumberFormatException quando arriverà a "60A" (non potrà parsare) e NullPointerException quando arriverà a null.

Presenza di più blocchi catch per un try

```
String s[] = new String[] {"123", "100", "60A", null};
for(int i=0;i <s.length;i++)
{
	try
	{
		Console.print("Digits:"+s[i].length()+", value:"+Integer.parseInt(s[i]));
	 }
	catch(NumberFormatException e)
	{
		Console.print("Invalid number "+e.getMessage());
	}
	catch(NullPointerException e)
	{
		Console.print("Warning: null value found, skipping term");
	}
}
```

- Questo programma produrrà il seguente output:
- Digits:3, value:123
- Digits:3, value:100
- Invalid number For input string: "60A"
- Warning: null value found, skipping term

- Verranno eseguiti il try o uno solo dei catch, in base all'eccezione verificatasi.

- Il programma non va in crash, ma continua a essere eseguito fino alla fine.

Genealogia delle eccezioni

- Le eccezioni sono oggetti, e di conseguenza vengono creati tramite classi. Le classi che creano eccezioni sono classi non dissimili da tutte le altre, e sono strutturate tramite ereditarietà.

- Questo viene usato implicitamente nella scrittura dei catch. I catch devono essere scritti dalla eccezione più specifica alla più generica, altrimenti rischiamo di incappare in dead code, vale a dire codice non raggiungibile.

- Questo accade perché verrà eseguito il primo blocco catch che corrisponde all’eccezione generata.

- Un esempio chiarirà tutto.

Esempio di ordine di scrittura dei catch

- Poniamo che il metodo m() possa generare le eccezioni E1 ed E2, con E2 extends E1. Ipotizzando di voler gestire entrambi i casi, con una gestione separata, vediamo due soluzioni apparentemente equivalenti ma in realtà molto diverse. In effetti, solo la prima è una soluzione:

```
// corretta
try
{
        m();
}
catch(E2 e)
{
        gestioneE2();
}
catch(E1 e)
{
        gestioneE1();
}
```

```
// errata
try
{
        m();
}
catch(E1 e)
{
        gestioneE1();
}
catch(E2 e)
{
        gestioneE2();
}
```

- Ricordiamo che, se E2 extends E1, E2 is_a E1.
- **Versione corretta**: ipotizzando che m generi E2 nel caso a sinistra, l'eccezione E2 sarà consumata dal primo blocco catch, come previsto. Nel caso in cui invece generi un E1 che non è un E2, verrà consumata dal secondo.
- **Versione errata**: ipotizzando che m generi un E2, nel caso a destra, questo E2 verrà consumato dal primo catch (tutti gli E2 sono E1, e il primo catch a rilevare il tipo dell'eccezione la consuma). Di conseguenza, il secondo catch, quello che verifica E2 sensu strictu, non verrà mai eseguito, perchè tutte le E2 verranno gestite nel primo catch. Il secondo catch è codice morto (dead code) e il compilatore si rifiuterà di proseguire.

Un esempio pratico

```
try
{
        Console.print(fibonacci(Console.readInt());
}
catch(NumberFormatException e)
{
        Console.print("Problema col numero inserito");
}
catch(Exception e)
{
        Console.print("Errore generico");
        e.printStackTrace();
}
```

- Stiamo invocando il metodo per il calcolo della sequenza di Fibonacci su un numero letto da tastiera.
- Nel caso di NumberFormatException, generato da Console.readInt(), finiremo nel primo catch.
- In qualunque altro caso, ad esempio un overflow dovuto a un numero troppo grande, finiremo nel secondo catch. Exception è l'antenato di tutte le eccezioni, e dire "catch Exception" è come dire "intercettare qualunque eccezione si sia verificata").

Generare eccezioni

- Fino ad ora abbiamo gestito eccezioni generate da altri metodi, ma vogliamo essere in grado di generarne di nostre.

- Prendiamo il caso del metodo fibonacci letto sopra. Il suo codice potrebbe essere:

```
public static int fibonacci(int n)
{
        return n<2 ? 1 : fibonacci(n-1)+fibonacci(n-2);
}
```

- Si tratta di un metodo ricorsivo, vale a dire, un metodo che richiama se stesso.

Ma che succede se invoco fibonacci(-2)?

- Non ha senso calcolare il valore della sequenza di Fibonacci per n = -2. E’ un input errato, e la maniera più logica di procedere è produrre un errore, un avvertimento al chiamante, una eccezione.

- Mentre throws avverte che questo metodo potrebbe dare eccezione, throw propaga una eccezione che verrà gestita dal chiamante o, in generale, dal primo blocco try che troviamo nella gerarchia delle chiamate.

Primo esempio di throw

- ```
  public static int fibonacci(int n) throws RuntimeException
  {
  	if(n<0)
  		throw new RuntimeException("Invalid n "+n+" for fibonacci");
  	return n<2 ? 1 : fibonacci(n-1)+fibonacci(n-2);
  }
  ```

- Nel caso in cui la condizione si verifichi, il metodo non termina, perché la generazione di una eccezione non gestita termina il metodo. In questo caso il metodo non ha fornito un valore di ritorno, ma una eccezione, che può essere quasi vista come un ritorno alternativo del metodo.

- Quando n è inferiore a 0, produciamo un oggetto di tipo RuntimeException e lo inviamo al chiamante, che potrà decidere come gestirlo. Riprendendo l'esempio di prima, verrà gestito all'interno del catch(Exception e).

- E' stata buona educazione dichiarare "throws RuntimeException" - POTREI dare questa eccezione, nella firma del metodo, ma non era obbligatorio in questo caso. RuntimeException è una eccezione unchecked, che non deve essere gestita, o propagata, obbligatoriamente. Se invece avessi generato una eccezione di tipo FileNotFoundException, saremmo stati costretti ad aggiungere throws FileNotFoundException dopo la firma del metodo, come abbiamo visto nel video precedente.
  ```

Multi catch

- In determinati casi vogliamo gestire più tipologie di eccezioni con lo stesso catch. Java prevede una sintassi abbreviata:

```
try
{
        m();
}
catch(E2 | E3 e)
{
        gestioneE2edE3();
}
```

  Il blocco catch gestirà allo stesso modo le eccezioni di tipo E2 ed E3, anche laddove non fossero “imparentate”.

finally

- Il blocco finally è un elemento opzionale della struttura try-catch. Quello che scriviamo nel blocco finally, che viene posto sempre dopo i catch, viene eseguito sempre, sia che sia stato eseguito il blocco try, sia che sia stato eseguito il blocco catch.

- Viene eseguito perfino nel caso in cui uno dei blocchi try o catch, abbia eseguito un return. Viene tipicamente utilizzato per chiudere file aperti e in generale per "fare pulizia" dopo il lavoro dei try e dei catch.

Esempio di finally

```
String s[] = new String[] {"123", "100", "60A", null};
for(int i=0;i<=s.length;i++)
{
	try
	{
		Console.print("Digits:"+s[i].length()+", value:"+Integer.parseInt(s[i]));
	}
	catch(NullPointerException | NumberFormatException e)
	{
		Console.print("Warning: null value found or invalid number, terminating");
		return;
	 }
	catch(Exception e)
	{
		Console.print("Unspecified exception on the element, skipping term");
	}
	finally
	{
		Console.print("Moving to the next element");
	}
 }
```

Output:

- Digits:3, value:123
- Moving to the next element
- Digits:3, value:100
- Moving to the next element
- Warning: null value found or invalid number, terminating
- Moving to the next element

Ho optato di terminare il programma quando trovo un valore non valido. I primi due valori sono validi, il terzo attiva il primo blocco catch (un blocco catch per eccezioni multiple), che stampa un messaggio di errore e invoca return. Nonostante il return termini il metodo, il finally viene comunque eseguito un'ultima volta. In generale, mettiamo nel finally ciò che deve essere sempre fatto, quale che sia il flusso di esecuzione del programma.

Conclusioni

- Una eccezione è un evento che interrompe la normale esecuzione del programma, un "errore". Java lo vede come un oggetto contenente tutte le informazioni sull'evento, per permettere al programmatore di gestirlo correttamente.
- Può essere gestita (try-catch) o propagata (throws) dal metodo in cui potrebbe verificarsi.
- Le eccezioni si dividono in checked e unchecked, a gestione o propagazione obbligatoria ed esplicita (FileNotFoundException) o implicita e opzionale (NullPointerException). Non siamo costretti a dire try-catch per NullPointerException, e neanche throws, mentre siamo costretti a fare una delle due cose per FileNotFoundException.
- Il metodo incaricato di gestire l'eccezione va deciso in fase di progettazione. Si rispetta il "principio di competenza": un metodo propaga l'eccezione se non sa come gestirla, lasciandone la gestione ai livelli superiori, e la gestisce se ritiene di avere sufficienti informazioni per farlo.
- Tutte le eccezioni dovrebbero essere gestite o evitate prima di arrivare a main o da main stesso
- Un blocco try può avere tanti blocchi catch quante sono le tipologie di eccezione producibili nel blocco try stesso. I blocchi catch vengono valutati in ordine. Il primo blocco catch che riconosce l'eccezione avvenuta come propria verrà eseguito, e verranno saltati gli altri.
- In gergo, diremo che il blocco catch "consuma" l'eccezione. Per questa ragione, si mettono i catch specifici in cima, prima degli altri, e quelli più generici sul fondo. un blocco catch può gestire più tipi di eccezione diverse (multi catch)
- In fondo ai catch è possibile avere un blocco finally, che verrà eseguito sia in caso di try che in caso di catch