Ingegneria del Software

Alessio Massetti

Politecnico di Milano - A.A. 2011/12 - II Semestre

Indice

Informazioni sul Corso:

Docente:

Pierluigi San Pietro

E-Mail:

pierluigi.sanpietro@polimi.it

Libri di Testo:

B.Liskov, J. Guttag, Program Development in Java

H. M. Deitel, P. M. Deitel, Vol 1+2: Java Fondamenti di programmazione+ Java Tecniche avanzate di programmazione

M. Fowler, UML distilled: guida rapida al linguaggio di modellazione standard

Pagina del Corso:

Sito del dipartimento di Elettronica e Informazione

Orario:

lunedì

13.30 - 15.00 (E.G 1)

martedì

12.15 - 16.15 (S 1.7) - Laboratorio (prova Finale)

mercoledì

10.30 - 12.00 (L 26.1.3)

venerdì

8.30 - 10.00 (L.26.0.2)

Parte I
Introduzione

1 Che cos'è l'ingegneria del software?

1.1 Definizione

E' la parte dell'informatica che studia software di grande dimensione, risultato di un lavoro di gruppo, che hanno durata di anni e vengono rilasciati in diverse versioni. La materia nasce storicamente quando i grandi progetti software incominciarono a fallire.

Definisco **prodotto** il software vero e proprio, mentre definisco **processo** il modo in cui avviene lo sviluppo

1.2 Qualità del software

Sono qualità del software: **afidabilità**, **efficienza** e **facilità** di **comprensione** del codice. Le prime due servono al cliente, la terza a chi dovrà effettuare le operazioni di **manutenzione**. Si definisce manutenzione tutta la serie di interventi sul software che **vengono effettuati dopo il suo rilascio**. Per questo motivo occorre del software che sia facilmente manutenibile, cioè modificabile.

1.3 Progettazione di un sistema

Per l'abilità di ingegnere del software non basta quindi saper programmare, ma bisogna saper progettare. Le principali caratteristiche dell'ingegnere del software sono quelle dell'analisi dei problemi e dei requisiti, del progetto di componenti riusabili **e del saper lavorare in un gruppo**. Il sistema grande e complesso richiede infatti di essere **progettato** prima di essere **implementato**.

Per progettare il sistema Si applica la tecnica "**divide et impera**" ovvero si divide il sistema in **moduli** che possono essere risolti indipendentemente. Questo consente sia uno sviluppo indipendente delle varie parti, sia una maggiore indipendenza tra i diversi sviluppatori del sistema.

Come scegliere i moduli? Si devono cercare di **ridurre le interazioni** tra i moduli stessi e di essere rigorosi nel definirli: chi usa un modulo deve infatti sapere **cosa** il modulo fa e **ignorare come** lo fa.

Un esempio di modulo elementare può essere considerato **il sottoprogramma C**: chi lo usa deve sapere solamente **come invocarlo e cosa effettivamente faccia senza sapere** poi **come** la funzione scelta viene effettivamente fatta.

2 Ciclo di vita del Software

2.1 Definizione

Si definisce ciclo di vita del software la descrizione del processo industriale del software, ovvero dalla sua concezione iniziale fino al suo sviluppo completo, al suo rilascio, alla successiva evoluzione e al suo ritiro.

2.2 Ciclo a cascata

Le fasi del ciclo a cascata vengono effettuate in successione e sono le seguenti:

- Studio di fattibilità
- Analisi e specifica dei requisiti
- Progettazione (di alto livello e di dettaglio)
  - di massima
  - di dettaglio
- Implementazione e test (di unità e di integrazione)
  - codifica
  - test di modulo
  - test di integrazione e di sistema
- Messa in esercizio ("Deployment")
- Manutenzione

2.2.1 Obiettivi del modello a cascata

- Identificare le fasi
- Caratterizzare gli output
- Forzare un processo lineare →nessun ritorno all'indietro considerato dannoso

2.3 Le fasi

2.3.1 Studio di fattibilità

Determina se il progetto debba essere avviato o meno, ovvero determina se l'azienda debba comprare un software o adattarsi a un software esistente. Molto spesso si modifica software esistente per le esigenze dell'azienda. Vengono quindi redatti progetti e piani di manutenzione

2.3.2 Analisi e specifica dei requisiti

Occorre capire **quale** sia il servizio richiesto dal cliente e **in che modo** ottenerlo. Richiede interazione con l'utente e comprensione del dominio applicativo. E' infatti necessario capire **immediatamente tutte le specifiche** richieste. Correggerle più avanti porta infatti a maggiori costi nella correzione.

2.3.3 Progettazione

Definisce l'architettura software. Deve supportare lo sviluppo parallelo dell'applicazione. Partendo dall'architettura complessiva vanno identificati i singoli moduli che compongono il sottosistema (classi e packages in java). L'idea del progetto è di avere un progetto diviso in **moduli**, con **ogni modulo che supporta una unità logica autosufficiente.**

La scomposizione in moduli deve quindi facilitare lo sviluppo concorrente del programma e le sue successive modifiche: **deve essere possibile modificare interamente un modulo senza andare ad impattare sugli altri moduli**. Un modulo deve incapsulare decisioni di progetto che si possono cambiare senza che se ne modifichi l'interfaccia.

Information Hiding: un modulo deve essere caratterizzato **da tutto ciò che può essere modificato senza che gli altri moduli ne risentano**. Deve quindi essere chiaro **cosa un modulo offre agli altri in termini di funzionalità** (interfaccia)

2.3.4 Codifica e Test

Il software viene quindi **compilato e implementato usando il linguaggio di programmazione scelto**. Sul software vengono **eseguite le prime modifiche stabili**.

I test sono a loro volta suddivisi in test di **unità, di integrazione e di sistema**. I test di unità vengono eseguiti all'interno dei moduli. Non è infatti conveniente assemblare subito il sistema: non è possibile cogliere facilmente la locazione di eventuali errori: l'assemblaggio viene infatti effettuato solamente quando tutti i moduli sono stati sufficientemente testati e vengono **intersecati incrementalmente** tra loro

2.3.5 Messa in esercizio

L'obiettivo di questa fase è **la distribuzione dell'applicazione sui diversi computer dei clienti e una eventuale configurazione degli stessi**. Il sistema viene installato e il personale addetto al suo uso viene addestrato - è poi possibile un test di accettazione da parte del cliente o da una organizzazione indipendente.

2.3.6 Manutenzione

Sono dette di manutenzione tutte le modifiche che avvengono dopo la consegna del software. **Il software non è soggetto ad usura: tutti i malfunzionamenti sono quindi dovuti ad errori.** Spesso la manutenzione compie più del 50% dei costi. Le tipologie di manutenzione sono due: correzione delle specifiche ed evoluzione del software. La manutenzione viene classificata quindi in **correttiva** (quando non sono rispettate le specifiche), **adattiva** e **perfettiva**. Perché il software evolve? Ci possono essere dei mutamenti nel contesto (ad esempio l'Euro), mutamenti nei requisiti o requisiti non noti in precedenza.

Ci si può aspettare quindi che il software consegnato contenga un gran numero di errori. Errori introdotti presto sono scoperti tardi e **più tardi vengono scoperti più è costosa la loro rimorzione**. Eliminare errori da software maturo costa infatti più di quando il software è piccolo e nuovo. La rimozione di errori causa l'introduzione di nuovi errori.

2.4 Problemi del ciclo a cascata

La fase finale di manutenzione non è disciplinata in alcun processo. La cascata prevede inoltre che il processo sia completamente noto, ovvero che siano comprese già tutte le specifiche, che **i requisiti vengano colti e siano stabili**. Questo non è nè realistico nè pratico: **nell'attività di sviluppo diventa infatti indispensabile il potere effettuare dei "ricicli"** anche coinvolgendo il committente nelle varie fasi del ciclo di vita del software.

Il software può essere rilasciato a "pezzi" per ottenere su questi dei feedback dai committenti (**rilascio incrementale**), in modo da coglierne le specifiche. Vengono quindi rilasciati **dei prototipi che sono "pezzi"** di applicazioni o componenti realizzati in maniera preliminare che **possono quindi evolvere nel tempo**.

2.5 Ciclo di vita a spirale

Consente di effettuare dei "ricicli", iterando è quindi possibile trovare e correggere errori in alto nel ciclo di vita. Attività di verifica vengono quindi svolte in ogni livello del software. Il processo viene quindi visto come un ciclo all'interno del quale vengono realizzate più volte le attività. Ad ogni ciclo **il passo della spirale, che rappresenta il costo**, **aumenta**.

Parte II
Java

3 Introduzione alla progettazione Object-Oriented

3.1 Premessa

Supponiamo che a due programmatori diversi venga affidata una applicazione in cui viene richiesto l'uso di una data. A deve programmare l'interfaccia utente, mentre B deve programmare il pagamento degli stipendi.

Codice di A

```
typedef struct {
    int giorno;
    int mese;
    int anno;
    } Data;

void stampaData(Data d) {
    printf("%d", giorno);
    if (d.mese == 1)
       printf("Gen");
    else
       if (d.mese == 2)
          printf("Feb");
    ...
    printf("%d", anno);
    }
```

Codice di B

```
Data d;
...
d.giorno=14;
d.mese=12;
d.anno=2000;
...
if (d.giorno == 27)
    paga_stipendi();
if (d.mese == 12)
    paga_tredicesime();
```

```
...
```

Il codice di B inizializza un oggetto di tipo data, definito da A. Inoltre usa l'oggetto di tipo data accedendo direttamente ai campi. **Quando definisco un tipo, infatti, definisco anche tutte le operazioni associate a quel tipo**. Per esempio, nel tipo int predefinito sono associate le note operazioni sui numeri interi. Per quanto riguarda invece i tipi definiti dall'utente posso definire nuove tipologie di dato partendo da tipi già noti tramite i costruttori di tipo. Il vantaggio del concetto di tipo è essenzialmente quello di astrarre i dati, ovvero **il programmatore non deve più occuparsi di collocare in memoria i valori** e della protezione da operazioni illegali. Ad esempio:

```
Data d;
int g;
d = g;
```

Viene automaticamente respinta dal compilatore.

3.2 Modifiche al codice in C

Supponiamo quindi, che nel programma al 3.1 la definizione di data cambi radicalmente:

```
typedef struct {

    int giorno;
    char mese[4];
    int anno;

} Data ;
void stampaData(Data d) {

    printf("%d", d.giorno);
    printf("%d", d.mese);
    printf("%d", d.anno);

}
```

Abbiamo modificato la definizione del mese portandolo in modalità carattere. Non abbiamo quindi modificato l'interfaccia dato che la data resta composta sempre di tre campi: giorno, mese, anno. Cosa succede al codice di B? Che non funziona più e occorre modificarlo:

```
Data d;
...
if (d.giorno == 27)

    paga_stipendi();

if (strcmp(d.mese,"Dic"))
```

```
    paga_tredicesime()
...
```

Come abbiamo già visto nel capitolo precedente, le modifiche sono molto spesso costose: **una piccola modifica della struttura comporta tante piccole modifiche al codice** e - come già visto - **modifiche al codice possono introdurre degli errori**. Se aggiungiamo che **le modifiche alle strutture dati sono le più frequenti all'interno del codice per una questione di efficienza** ecco che occorre disciplinare questo problema.

3.3 Struttura dati ad accesso disciplinato

Il problema viene quindi risolto disciplinando o **impedendo l'accesso ai vari campi della struttura dati**. In parole povere non sarà più consentito scrivere

```
(d.mese==12)
```

Come fare per accedere ai campi della data? **Il responsabile della struttura dati deve fornire tutte le operazioni**. L'accesso alla struttura dati va quindi fornito unicamente tramite operazioni predefinite:

```
void inizializzaData(Data *d, int g, int m, int a) {
    d->giorno = g;
    d->mese = m;
    d->anno = a;
}
int leggi_giorno(Data d) {return(d.giorno);}
int leggi_mese(Data d){return(d.mese);}
int leggi_anno(Data d){return(d.anno);}
```

Il codice di B va quindi scritto in modo da utilizzare queste operazioni predefinite

```
Data d;
...
inizializzaData(*d, 14,12,2000);
if (leggi_giorno(d) == 27)
    paga_stipendi();
if (leggi_mese(d) == 12)
    paga_tredicesime();
...
```

Se quindi A va a modificare la struttura interna del proprio codice, **deve modificare anche le operazioni predefinite ma deve lasciare immutato il prototipo**.

```
int leggi_mese(Data d){
    if (strcmp(d.mese,"Gen"))
        return(1);
    ...
    else
    if (strcmp(d.mese,"Dic"))
        return(12);
}
void inizializzaData(Data *d, int g, int m, int a)) {
    d->giorno = g;
    if (m==1)
        d->mese = "Gen";
    ...
}
```

Abbiamo quindi incapsulato la struttura dati: **ogni modifica all'interno del codice resta confinata alla struttura dati** stessa.

3.4 Tipo di Dato Astratto (ADT)

Il tipo di dato astratto è un dato che è descritto quindi **non in base ai suoi campi, ma alle operazioni** per implementare oggetti di quel tipo. Le operazioni sono quindi le sole possibilità che ha chi usa quell'oggetto di accedervi. ADT esporta il nome del tipo e l'interfaccia, mentre **nasconde la struttura del tipo e l'implementazione delle operazioni**. L'interfaccia di un ADT deve quindi "promettere" che le operazioni saranno sempre valide e deve **includere tutte le modifiche utili**. Ad esempio:

```
void giorno_dopo(Data *d){
d->giorno++;
    if (d->giorno > 31){/*ma mesi di 30, 28, 29?*/
        d->giorno = 1;
d->mese++;
if (d->mese > 12)
    { d->mese = 1;
    d->anno++; }
}
```

Perché usiamo Java? Perché il C non ha costrutti linguistici diretti per definire ADT, **non possiamo impedire che si possa accedere ugualmente ai campi** della struttura dati. C manca inoltre di astrazioni che si ritrovano nei linguaggi Object-oriented.

3.5 Caratteristiche di Java

- **Object Oriented**. Permette di incapsulare le strutture dati all'interno di opportune dichiarazioni dette classi
- **Distribuito**
- **Indipendente dalla piattaforma.** Può essere eseguito su qualunque pc giri una Java Virtual Machine
- **Sicuro.** Viene eseguito in una sandbox che non può danneggiare l'host.
- Non esistono i tradizionali concetti di programma o sottoprogramma. Esistono le **classi** che raggruppano **dati privati e metodi**.
- In Java le variabili **possono essere dichiarate in ogni punto** del programma ma **vanno inizializzate** prima di utilizzarle.
- Tipi primitivi simili al C. Identiche strutture di controllo e dichiarazioni di variabili. Differenze per quanto riguarda gli array
- Classi, Interfacce, Packages, Information Hiding, Ereditarietà, Polimorfismo e binding dinamico

4 Programmi Java

4.1 Hello World

Un programma Java è **organizzato come un insieme di classi**: ogni classe corrisponde a una dichiarazione di tipo o a una collezione di funzioni. Ogni classe contiene dichiarazioni di variabili (attributi) e di funzioni (metodi). Il **programma principale è rappresentato da un metodo speciale di una classe, detto main**.

```
file: HelloWorld.java
public class HelloWorld {

    public static void main(String args[]){
        System.out.println("Hello world!");
    }

}
```

Questo è il codice di un Hello World in Java. Notiamo subito che

- Il sorgente deve stare in un file separato dalle classi pubbliche che ha lo stesso nome della classe principale
- Le classi vengono definite come public, ovvero accessibili a tutti coloro che utilizzano il codice

- Static indica che il metodo appartiene ad una classe
- Void main ha lo stesso significato del C
- String Args [] serve, come in C, per passare eventuali parametri da tastiera
- Per stampare utilizziamo la classe speciale system, predefinita dal programma

4.2 Esempio di classe in Java

```
class Data {

    private int giorno;
    private int mese;
    private int anno;
    ...
    public int leggi_giorno(){ //@ ensures (*restituisce il giorno*); }
    public int leggi_mese(){ //@ ensures (*restituisce il mese*); }
    public int leggi_anno(){ //@ ensures (*restituisce l'anno*); }

}
```

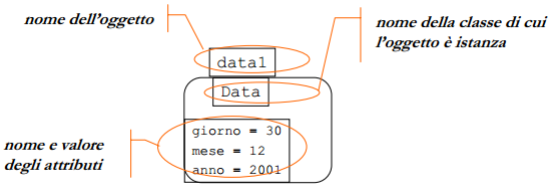
Possiamo vedere class come la struct del C. A differenza della struct però, possiamo definire come private l'accesso ai vari campi della struttura dati. I private sono invisibili all'esterno della classe, mentre i public no. Nell'esempio sopra riportato le funzioni sono definite appunto come public poiché devono essere adoperate per ottenere l'accesso. Una nuova classe viene definita in questo modo

```
<visibilità> class <nome classe>


    { <lista di definizioni di attributi, costruttori e metodi }
```

Una classe può essere vista come un tipo definito dall'utente che specifica le operazioni utilizzabili sul tipo stesso. Definisce quindi un vero e proprio ADT. Giorno, mese e anno sono detti **attributi** della classe.

4.3 Classi ed oggetti

In un programma Object Oriented si definiscono classi per poter definire degli oggetti. Un oggetto è un'area di memoria che contiene valori per ogni attributo definito nella classe. **Tutti gli oggetti nella stessa classe hanno quindi la stessa struttura**. Possiamo vedere la classe come **una sorta di "stampo" per creare oggetti simili ma distinti**.



Un oggetto è quindi **istanza** della classe che abbiamo usato come stampo per crearlo, ed ogni oggetto è caratterizzato da uno stato **definito dai valori delle variabili interne** all'oggetto. L'oggetto può quindi essere modificato e letto tramite delle operazioni interne alla classe che vengono chiamati **metodi della classe**. I metodi, se non si usa la parola chiave "static" appartengono agli oggetti e hanno come parametro implicito l'oggetto stesso a cui appartengono.

Ogni metodo è definito come segue:

```
<tipo val. rit.> <nome.>([<dic. par. formali>])
    {
        <corpo>
    }
```

Possiamo quindi vedere il metodo come quella che era la funzione nel linguaggio C.

4.4 Accesso ad attributi e metodi

L'accesso agli attributi e ai metodi avviene attraverso la notazione punto.

```
Data d;
int x;
...
//codice che inizializza d
x = d.leggi_giorno();
```

- Esegue leggi_giorno() su oggetto d:
  - restituisce valore del giorno della data d.
- E' come se leggi_giorno() avesse come argomento implicito d:
  - in C scriveremmo: leggi_giorno(d);
- Si dice anche che all'oggetto d **inviamo il messaggio** leggi_giorno

Questo porta a cambiare la sintassi di accesso alla struttura dati

```
Data d;
...
if (d.giorno_di_paga())
```

```
    paga_stipendi();
if (d.mese_13esima())
    paga_tredicesime();
...
```

L'accesso ad attributi e metodi di un oggetto si effettua quindi tramite la notazione punto. Lo **stato degli oggetti può inoltre cambiare nel tempo** chiamando i metodi opportuni, ma esistono oggetti immutabili come quelli della **classe predefinita string**. Per modificare lo stato, il metodo deve potere accedere ai campi dell'oggetto su cui è stato chiamato: **nella definizione di un metodo ci si può riferire direttamente (senza notazione punto) ad attributi e metodi dell'oggetto** sul quale il metodo sia stato invocato.

```
class Data {
    ...
    public void giorno_dopo()
        { //@ ensures (*incrementa la data*)
        giorno++;
        if (giorno > 31)
            { giorno = 1;
            mese++;
            if (mese > 12)
              { mese = 1;
              anno++;}
            }
        }
    ...
    Data d;
    ...
    d.giorno_dopo(); /*modifica lo stato dell'oggetto d*/
}
```

Attraverso i metodi public di una classe, è quindi possibile vedere quale sia lo stato dell'oggetto

```
Data d;
int x;
...
x = d.leggi_giorno();
```

Ma non è possibile accedere ai dati private al di fuori del codice di Data: il costrutto di classe consente quindi di definire un tipo di dato astratto.

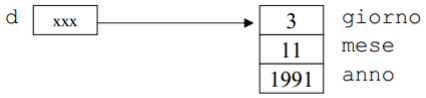
4.5 Programmazione in the small

4.5.1 Tipi primitivi e variabili

- Tipi numerici:
  - byte: 8 bit
  - short: 16 bit
  - int: 32 bit
  - long: 64 bit
  - float: 32 bit
  - double: 64 bit
- Altri tipi:
  - boolean: true false
  - char: 16 bit, carattere Unicode

I tipi numerici sono, a differenza del C, definiti da un numero fisso di bit, **indipendente dalla macchina**. I tipi sopra elencati, inoltre, sono detti tipi primitivi. **Tutte le variabili di tipo primitivo contengono il valore**. Tutte le altre contengono riferimenti al valore e sono dette per questo di **tipo riferimento**. In Java non esistono puntatori: gli oggetti referenziati dalle variabili sono allocate sullo heap e sono deallocate quando il sottoprogramma ritorna al chiamante. Le altre variabili sono allocate sullo stack.

Ogni variabile dichiarata con una classe contiene **un riferimento a un oggetto: un indirizzo di memoria**: il valore effettivo dell'indirizzo non è noto e non interessa.



La dichiarazione di una variabile con una classe, quindi, non alloca spazio per un oggetto ma solamente per il riferimento all'oggetto. A una variabile di tipo riferimento è assegnato inizialmente il riferimento **null**, per indicare che la variabile non è ancora associata ad alcun oggetto. Un parametro o una variabile locale **non possono comunque essere usati senza essere inizializzati** (compilatore rileva staticamente mancanza di inizializzazione). **Occorre costruire un oggetto e inizializzarlo esplicitamente**.

4.5.2 Costruzione di un oggetto

La costruzione di un oggetto si realizza dinamicamente tramite l'operatore new. Esempio

```
Data d = new Data();
```

Questa istruzione:

- Costruisce un nuovo oggetto di tipo Data
- Restituisce il riferimento all'oggetto appena creato
- Il riferimento **viene conservato nella variabile d** per potere usare l'oggetto
- Data() è un metodo particolare (**ha lo stesso nome della classe**, si riconosce come metodo dalle parentesi "()") chiamato **COSTRUTTORE**

Se eseguissimo questo codice:

```
Data data;
data d = new Data();
data d = new Data();
```

Otterremmo due oggetti di tipo data, ma nella variabile d **avremmo il riferimento solamente al secondo oggetto**, col primo che verrà perso e distrutto dal garbage collector per evitare memory leaks (viene eseguita in continuazione una routine di sistema che elimina tutti gli oggetti senza riferimenti, a discapito dell'efficienza) Se l'implementazione dell'oggetto è dichiarata in maniera privata, l'unico modo per costruire un oggetto è quello di definire all'interno della classe un metodo speciale detto costruttore che **ha lo stesso nome della classe.**

Il metodo costruttore **alloca le variabili e le inizializza**, eliminando una frequente fonte di errori, il suo codice contiene tutte le informazioni necessarie ad inizializzare gli attributi in via di creazione. Se non si definisce nessun costruttore, il compilatore fornisce il costruttore di default (senza parametri), che svolge le seguenti funzioni:

- Alloca lo spazio per gli attributi di tipo primitivo
- Alloca lo spazio per i riferimenti agli attributi di tipo definito dall'utente
- Inizializza a **null** tutti i **riferimenti**, a **0** tutte le variabili **numeriche**, a **false** tutti i **boolean**

4.6 Assegnamento

L'operatore di assegnamento "=", quando applicato a:

- Variabili e espressioni di tipi primitivi: copia il valore. **Es.**

```
int x,y;
x=5;
y=x;
```

  - In y viene copiato il valore contenuto in x;

- Variabili e espressioni di tipo riferimento: **copia il riferimento, non l'oggetto**. **Es.**

```
Data d1, d2;
d1=new Data(1,2,2004);
d1 = d2;
```

  - In d1 è copiato il **riferimento**, cioe' d1 e d2 **referenziano** (ossia condividono) **lo stesso oggetto**

4.7 Confronti tra variabili

L'operatore di confronto == confronta **i valori dei reference**. non gli oggetti!

```
Data d = new Data(1,12,2001);
Data d1= new Data(1,12,2001);
if (d==d1)
```

Questo if restituisce false, perché d e d1 **sono due variabili di tipo riferimento a due date diverse**. Come fare quindi i confronti? Si usa il metodo equals() che consente di verificare se due oggetti sono uguali (nel senso che **hanno lo stesso valore dello stato**). Caso particolare del metodo per **string**: contengono **la stessa sequenza di caratteri. Esempio di uso:**

```
String b = new String("Ciao");
String c = new String("Ciao");
if (b.equals(c))
```

4.8 Allocazione in memoria: Stack vs Heap

Le variabili sono **sempre allocate a run-time nello stack** e vengono deallocate quando **il sottoprogramma ritorna** al chiamante. Gli oggetti referenziati sono invece sempre allocati nello heap. Cosa vuol dire questo? Che **i tipi primitivi vengono deallocati** mentre **le modifiche fatte agli oggetti tramite un metodo restano**.

5 Dettagli sul linguaggio Java

5.1 Array

5.1.1 Definizione

Un array in java viene definito come:

```
T[];
```

Similmente sono dichiarati gli array multidimensionali:

```
T[][];
T[][][];
```

Esempi

```
int[];
float[][];
Persona[];
```

5.1.2 Dichiarazione e inizializzazione

Gli array vengono dichiarati in questo modo

```
int[] ai1, ai2;
float[] af1;
double ad[];
Persona[][] ap;
```

ai1, ai2 sono array di tipo int, af1 è un array di tipo float, ad è un array di tipo double, e ap è una matrice di riferimenti ad oggetti di tipo persona.

N.B. Le seguenti dichiarazioni sono del tutto equivalenti:

```
double ad[];
double[] ad;
```

Per quanto riguarda l'inizializzazione

```
int[] ai={1,2,3};
double[][] ad={{1.2, 2.5}, {1.0, 1.5}}
```

Ho definito in questo caso due tipi di array. Il tipo ai sarà un tipo di array di tre interi contenente 1, 2, 3 e ad sarà un tipo di matrice 2x2 contenente i valori nel codice.

5.1.3 Array vero e proprio (oggetto)

L'array vero e proprio si dichiara in questo modo, tramite il metodo costruttore:

```
new <tipo> [<dimensione>]
```

Esempi:

```
int[] i=new int[10], i={10,11,12};
float[][] f=new float[10][10];
Persona[] p=new Persona[30];
```

Nella prima dichiarazione dichiaro un array di 10 interi e inizializzo i primi tre a 10, 11, 12. Posso inizializzare anche dopo a differenza del C. Nel secondo caso dichiaro una matrice 10x10 di float, nel terzo caso dichiaro un array di 30 oggetti persona.

N.B. Come nel C gli array partono da 0 e arrivano a n-1. Se definisco un array di 10 elementi potrò usare le posizioni 0-9

Vediamo un altro esempio:

```
Person[ ] person; // riga 1
person = new Person[20]; // riga 2
person[0] = new Person( ) // riga 3
```

Nella riga 1 definisco la variabile "person" che conterrà un array di oggetti definiti nella classe "Person". L'array ancora non esiste: ho una variabile che **punta a null**. Nella riga 2 inizializzo l'array di 20 oggetti di tipo "Person", ma le 20 caselline punteranno ancora tutte a null. **Solo nella riga 3 riempio l'array con il primo oggetto** utilizzando il metodo costruttore della classe Person.

Caso diverso sarebbe invece stato qualora l'array fosse stato di tipo base:

```
float f[] = new float[10];
```

Essendo float un tipo primitivo viene allocato nella variabile f un array di 10 float i quali contengono il valore 0.0 - ma è appunto **un valore, non un riferimento** a null.

5.1.4 Loop Generalizzato per collezioni (foreach)

E' disponibile in Java un ciclo per scorrere più velocemente gli array detto foreach. Si dichiara in questo modo

```
for (T x: C)
```

Dove C è il **nome dell'array** e T il suo **tipo** ed è del tutto **equivalente** al vecchio

```
for (x=0, x<sizeof(C), x++)
```

Vediamo un esempio applicativo

```
int sum (int [ ] a) {
    int result = 0;
    for (int n : a)
        result += n;
    return result;
}
```

L'istruzione

```
result += n;
```

E' equivalente alle seguenti due

```
result = result + n;
n++;
```

5.2 Accesso ad attributi e metodi locali. La pseudo-variabile this

Nella definizione di un metodo ci si riferisce ad attributi e metodi dell'oggetto sul quale il metodo sia stato invocato direttamente. Esempio:

```
class Automobile {
    String colore;
    void dipingi(String col) {colore=col;}
    ...
}
```

Le variabili locali ad un metodo (o i parametri formali) possono però mascherare gli attributi della classe (in altre parole: posso avere una variabile locale in un metodo che si chiama esattamente come l'attributo). Si risolve il problema con la parola chiave **this**:

```
public class Automobile {
    private String colore, marca, modello;
    ...
    public void trasforma(String marca, String modello) {
        this.marca=marca; this.modello=modello;
        }
    }
    ... // in un'altra parte di codice (main?)
    Automobile a = new Automobile();
    a.trasforma("Ford", "T4");
```

this.marca si riferisce **al campo marca dell'oggetto corrente** così come this.modello si riferisce al campo modello dell'oggetto corrente mentre marca e modello alla destra dell'uguale sono **esattamente i due passati come parametri**. Dopo le due istruzioni finali a.marca diventa quindi Ford mentre a.modello diventa T4. This può anche essere usata per **restituire il riferimento all'oggetto.**

```
public class InsiemeDiInteri {
...
    public InsiemeDiInteri inserisci(int i) { ...
```

```
            // modifica this inserendovi l'elemento i return this;
            // restituisce l'insieme modificato
            ...
        }

...
}
...
InsiemeDiInteri x,y,z;
...
// qui x e y sono inizializzate opportunamente
z = (x.inserisci(2)).unione(y)
// utile che inserisci restituisca insieme
```

5.3 Oggetti Mutabili e Immutabili

Come abbiamo visto nel capitolo 4 **lo stato degli oggetti mutabili può cambiare nel tempo**, chiamando metodi opportuni. Esistono però alcuni casi in cui gli oggetti non sono mutabili, come è per esempio la classe predefinita **string**.

L'oggetto "abcd" (in quanto "String") **è immutabile**: il suo stato non cambia ovvero non esistono operazioni che consentono di modificare lo stato di un oggetto string. In generale, invece, **gli oggetti sono mutabili**:

```
int[] x = {0, 0, 0};
x[0] = 5;
```

5.4 La Classe String

Valgono due fondamentali regole base:

- La variabile di tipo String è un **oggetto**, manipolato con un **reference**
- Le stringhe sono **immutabili** (**non si possono aggiungere o togliere caratteri a una stringa**, ma occorre costruirne una nuova)

E' inoltre disponibile per le stringhe, a differenza del C, l'operazione di concatenamento **diretta** tramite il "+"

Sono definiti inoltre dei **metodi speciali** per la classe String:

```
int length()
```

restituisce la lunghezza di una stringa;

```
char charat(int index)
```

restituisce il char alla posizione index (il primo ha posizione 0)

```
String substring(int beginIndex)
```

restituisce l'indice nel quale inizia una sottostringa (parte da 0).

Vediamo un esempio

```
public class ProvaString {

    //OVERVIEW: ...
    public static void main (String argv[])
    {

        String a = new String(); //a è reference a stringa vuota
        String b = new String("Ciao“); //b è reference a stringa “Ciao”:

            //abbreviazione: String b = "Ciao“;
        String c = new String(b); //Ora c e' copia di b String d = b;
            //d e b sono alias
        System.out.println(b + " " + c + " “+ d);
    }

}
```

L'assegnamento d=b è **assegnamento dei reference**! Non si copia l'oggetto

5.5 Enumerazioni

Posso costruire in Java dei tipi che possono avere solamente un valore ben definito oppure null. Questi tipi sono detti **enumerazioni**.

```
enum Size {SMALL, MEDIUM, LARGE, EXTRA_LARGE};
Size s = Size.MEDIUM;
```

Size è una vera classe con quattro istanze. Per confrontare i valori è possibile usare == invece di equal. s può assumere solamente null o uno dei quattro valori enumerati. A una classe enumerata possono essere aggiunti costruttori e attributi

```
enum Size {


    SMALL(“S”), MEDIUM(“M”), LARGE(“L”), EXTRA_LARGE(“XL”)
    private String abbreviation;
    private Size(String abbreviation) {
        //il costruttore
        this.abbreviation=abbreviation
    }
    public String getAbbreviation() {return abbreviation;}
```

```
};
```

Nell'esempio viene restituita l'abbreviazione della classe col costruttore (che viene invocato passando il parametro. Esempio se invoco il costruttore per SMALL passo come parametro S e lo imposto come abbreviazione restituita.

Tutte le classi enumerate sono quindi considerate come eredi della classe enum, la quale offre i seguenti metodi:

```
// dà la costante enumerata della classe indicata che ha quel nome
static Enum valueOf(Class enumClass, String name)
// dà il nome della costante enumerata
String toString()
```

Altre operazioni

```
Scanner in = new Scanner(System,in);
String str = in.next(); //e.g. se letta stringa "SMALL"...
Size siz = Enum.valueOf(Size, str); //...dà costante Size.SMALL
System.out.println(Size.LARGE.toString()); //stampa "LARGE"
```

Ogni classe enumerata **ha un metodo static che restituisce un array contenente tutti i valori della classe.**

```
Size[] valori = Size.values;
```

5.6 Overloading di metodi

All'interno di una stessa classe possono esservi **più metodi con lo stesso nome purché si distinguano per numero e/o tipo dei parametri**

Attenzione: Il tipo del valore di restiuito non basta a distinguere due metodi (motivo: a volte un metodo può essere chiamato solo per i side-effect e valore restituito è ignorato)

```
class prova {
int max(int a, int b, int c) {...}
double max(double a, double b) {...}
int max(int a, int b) {...}
}
```

Ogni volta viene chiamata la funzione giusta. Vediamo un esempio di overloading **errato**.

```
class C {
int f() {...}
int f(int x) {...} // corretto
void f(int x) {...} // errato
}
```

Questo esempio è errato perché cambia solamente il tipo parametro restituito e non il prototipo (o signatura) del metodo.

5.6.1 Overloading dei costruttori

```
public class C {
    private int i1;
    private int i2;
    public C(int a1, int a2) {
        i1 = a1; i2 = a2;
    } public C(int a) {
        i1 = a; i2 = a;
    }
}
```

Cosa succede se definisco il metodo in questo modo?

```
public class Esempio {
    public static void main(String args[]) {
        C x, y, z;
        x = new C();
        y = new C(1);
        z = new C(1,4);
    }
}
```

La prima chiamata (x) restituirà un errore di compilazione dato che dovrebbe venire chiamato il metodo costruttore predefinito ma siccome **tale metodo non esiste avendone definiti altri due** non può essere chiamato. La seconda e la terza chiamata vanno a buon fine.

5.7 Argomenti a numero variabile

In Java 1.5 si possono creare metodi e costruttori con un numero variabile di argomenti

```
public void foo(int count, String... cards) {
    // body
}
```

... significa "zero o più argomenti". Tutti gli argomenti sono del tipo preceduto da ... e vengono memorizzati **in un array il cui nome è specificato dopo** (in questo esempio **cards**). E' quindi possibile iterare l'array mediante il foreach

```
for (String card : cards) {
    // loop body
}
```

5.8 Metodi static

Il metodo statico è un metodo che non è associato all'istanza di classe ma è **associato alla classe.** Si può accedere a un attributo di classe senza bisogno di creare un oggetto tramite la notazione:

```
<nome classe>.<nome attributo>
```

Il metodo static viene definito in questo modo

```
static <definizione dell'attributo o metodo>
```

Un metodo di classe può essere **invocato senza bisogno di creare un oggetto** tramite la notazione:

```
<nome classe>.<nome metodo>(<par. attuali>)
```

Un metodo static può accedere ai soli attributi e metodi static mentre un metodo convenzionale può accedere liberamente a metodi e attributi static.

Vediamo quindi il seguente esempio:

```
class Shape {


    static Screen screen=new Screen(); // si noti l'iniz. static
    void setScreen(Screen s) {screen=s;}
    void show(Screen s) {setScreen(s); ...}
}
...
Shape.setScreen(new Screen()); // corretto
Shape.show(); // errato, show è un metodo normale
Shape s1=newShape(), s2=new Shape();
Screen s=new Screen();
s1.setScreen(s);  // corretto, si possono chiamare metodi static su oggetti
// in questo punto s2.screen==s1.screen==s
```

Quindi i due oggetti s2 e s1 avranno il riferimento **allo stesso screen** nel campo screen. Se cambio il valore del campo screen **questo cambia in tutti gli oggetti**.

5.9 Attributi costanti

È possibile definire attributi costanti tramite la notazione:

```
final <definizione di attributo>=<valore>
```

Vediamo un esempio

```
class Automobile {
```

```
    int colore;
    final int BLU=0, GIALLO=1,...;
    void dipingi(int colore) {this.colore=colore;}
}
...
Automobile a=new Automobile(); a.BLU=128; // errato
System.out.println("BLU="+a.BLU); // corretto
```

Il valore blu non può essere quindi cambiato.

5.10 Tipi riferimento per tipi primitivi

Sono dei tipi che permettono di utilizzare il tipo riferimento anche per i tipi primitivi per omogeneità con gli oggetti. Un esempio sono i tipi Integer e Character

```
Integer i; // qui i vale null!
i = new Integer(5); // i è un rif. a oggetto che contiene 5 Integer
x = i; // sharing: x e i sono stesso oggetto
int y = x.intValue(); // per accedere a valore di x
```

i, x e y sono tre variabili di tipo riferimento che puntano tutte ad x.

```
i = y
```

Boxing automatico: i diventa un riferimento al valore a cui è riferito y.

6 Packages e Information Hiding

6.1 Regole di visibilità

Tutte le classi in un programma Java sono raggruppate in uno o più Packages che **definiscono delle regole di visibilità**. Se visibile nel package A, infatti, **una entità X del package B** viene denotata come B.X (quindi si possono usare stessi nomi in packages diversi).

6.2 Struttura di un programma Java

6.2.1 Compilation Unit

Si definisce unità di compilazione o compilation unit ogni file sorgente che contiene **una o più classi delle quali una sola classe viene dichiarata pubblica**. Il file sorgente deve avere **lo stesso nome della classe pubblica**. C'è inoltre al più **un solo metodo di nome main** – Si può specificare il package di appartenenza di ogni classe nella compilation unit, se non lo si fa si assume un package senza nome di default.

6.2.2 Package

Il Package è quindi **una directory** che contiene tutte le compilation unit. Il package introduce un nuovo ambito di visibilità dei nomi: unit con lo stesso nome possono stare in package diversi. Ogni package contiene un insieme di classi pubbliche ed un insieme di classi private al package ("friendly") e **solo le classi pubbliche possono essere importate in altri package**.

6.2.3 Esempio

Compilation Unit A:

```
package myTools.text;
public class TextComponent { ... Zzz z; ... }
```

Compilation Unit B:

```
package myTools.text;
public class yyy { ... }
class Zzz{ ... }
```

Compilation Unit C:

```
package myFigs.planar;
public class DrawableElement { ... }
class xxx { ... }
```

In questo esempio le C.U. A e B appartengono allo stesso package, quindi A può usare la classe definita in B. C appartiene a un diverso package e può utilizzare solamente la classe yyy contenuta in B. A loro volta A e B non possono usare la classe xxx.

6.3 Information Hiding

6.3.1 Classi

Le classi possono essere:

- **Public:**
  - sono visibili a tutti tramite la funzione import
  - il file deve avere stesso nome della classe pubblica
  - può esserci al più una public class per ogni file
- **"friendly"** (quando non si indica nulla)
  - sono visibili solo all'interno dello stesso package/compilation unit
  - possono stare in file con altre classi
- Private
  - Possibile teoricamente (il compilatore non dà errore) ma non ha senso

6.3.2 Attributi e Metodi

Possono essere

- **Public**
  - sono visibili a **tutti quelli per cui la classe è visibile**
- **Private**
  - sono visibili solo ai metodi all'interno della stessa classe
- **"friendly"** (quando non si indica nulla):
  - sono visibili solo nelle classi all'interno dello stesso package/compilation unit: **per gli altri package, è come se fossero private**

6.3.3 Quando Public e quando Private?

Quando si dichiara public un attributo o un metodo di una public class, si fa una promessa, agli utilizzatori della classe, che **l'attributo o il metodo sarà disponibile e non cambierà,** perlomeno dal punto di vista degli utilizzatori della class. Tutti i metodi e attributi per cui ci **si vuole garantire la possibilità di modifica o eliminazione devono essere private** (o al massimo, ma solo se indispensabile, friendly. E' meglio quindi **usare private per tutti gli attributi della classe** (abbiamo detto al 3.3 che vogliamo impedirne l'accesso ai campi) e per tutti i metodi "helper" cioè **interni alla classe**: gli unici metodi da dichiarare public sono i metodi che devono essere utilizzati da **utilizzatori esterni** alla classe.

Nei programmi Java viene automaticamente importato un package standard che è il package java.lang che contiene tutte le classi di uso molto frequente. Non è necessario importare esplicitamente questo package per utilizzare, ad esempio, la classe String.

6.4 Ereditarietà

E' possibile stabilire una relazione di sottoclasse tra le classi di un programma Java $B \subseteq D$

La classe B è detta classe base, o antenato, o padre, o **sovraclasse** mentre la classe D è detta derivata, o discendente, o figlio, o erede, o **sottoclasse. La sottoclasse eredita tutta l'implementazione (attributi e metodi) della sopraclasse.** Una sottoclasse può quindi utilizzare tutti gli attributi e metodi della sovraclasse ma anche **implementarne di nuovi**. E' inoltre consentito **l'overriding** ovvero la ridefinizione nella sottoclasse di un metodo già definito nella sovraclasse che quindi viene sostituito. Per l'ereditarietà si utilizza la parola chiave extends, vediamo un esempio:

```
public class Automobile {
```

```
    private String modello;
    private boolean accesa;
    public Automobile(String modello) {

        this.modello=modello; this.accesa = false;
    }
    public void accendi() {
        accesa=true;
    }
    public boolean puoPartire() {
        return accesa;
    }
}
public class AutomobileElettrica extends Automobile {
    private boolean batterieCariche;
    public void ricarica() {
        batterieCariche=true;
    }
    public void accendi(); // da implementare
}
```

Gli oggetti creati con la classe AutomobileElettrica potranno quindi disporre di un attributo ulteriore (batteriecarche) e di un metodo ulteriore (ricarica) oltre a poter utilizzare tutti i metodi della classe Automobile. Nel caso invocassi il metodo accendi con un oggetto AutomobileElettrica viene chiamato il metodo accendi di AutomobileElettrica. E' possibile creare una gerarchia a più livelli mettendo classi che ereditano da classi che a loro volta ereditano da classi e così via. Ogni classe **può comunque ereditare da una sola classe.**

6.5 Costruzione di un software estendibile usando l'ereditarietà

Supponiamo di dover realizzare un editor di figure geometriche e prendiamo in considerazione le figure cerchio, triangolo e rettangolo. In C avremmo il seguente pseudo-codice:

```
typedef struct ... Figura;
Figura figure[100];
figure[1] = "rettangolo";
figure[2] = "triangolo";
figure[3] = "cerchio";
void disegnaTutto(Figura *figure) {
    for(i=0; i<100;i++) {
```

```
        if (figure[i] è "rettangolo")
            "disegna rettangolo"
        if (figure[i] è "triangolo")
            "disegna triangolo"
        if (figure[i] è "cerchio")
            "disegna cerchio"
    }
}
```

Supponiamo quindi che a questo punto del codice vogliamo aggiungere il trapezio. Cosa succede? Succede che la funzione disegnatutto deve cambiare per supportare il nuovo tipo

```
typedef struct ... Figura;
Figura figure[100];
figure[1] = "rettangolo";
figure[2] = "triangolo";
figure[3] = "cerchio";
figure[4] = "trapezio";
void disegnaTutto(Figura *figure) {
    for(i=0; i<100;i++) {
        if (figure[i] è "rettangolo")
            "disegna rettangolo"
        if (figure[i] è "triangolo")
            "disegna triangolo"
        if (figure[i] è "cerchio")
            "disegna cerchio"
        if (figure[i] è "trapezio")
            "disegna trapezio"
    }
}
```

Se volessimo invece sviluppare il medesimo programma in versione Object Oriented invece si definisce classe (astratta) Figura, con metodo (astratto) disegna(), e sue eredi Rettangolo, Cerchio, Triangolo che implementano disegna().

```
class Figura (); {
public void disegna(){ ... } }
class Cerchio extends Figura (); {
public void disegna(){ ... } }
class Rettangolo extends Figura (); {
public void disegna(){ ... } }
class Triangolo extends Figura (); {
public void disegna(){ ... } }
public static void disegnaTutto(Figura [] figure) {
```

```
        for (i= 0; i<100;i++)
            figure[i].disegna();
    }
```

La quantità di codice risulta essere più o meno la stessa di prima, ma questa volta il codice è meglio organizzato. Se volessimo estendere al trapezio infatti basterebbe modificare la parte di definizione degli oggetti aggiungendo la classe.

```
class Trapezio extends Figura (); {
public void disegna(){ ... } }
```

Il codice definito per la disegnaTutto infatti **non cambia**!

7 Polimorfismo

7.1 Cos'è?

Il polimorfismo è la capacità per un elemento sintattico di riferirsi ad elementi di tipo diverso. L'idea alla base è quella che gli oggetti abbiano due tipi di dichiarazioni: quella statica e quella del costruttore definita a run-time. In Java una variabile di un oggetto T, infatti, può definirsi ad un qualsiasi oggetto **il cui tipo sia T o un sottotipo di T**.

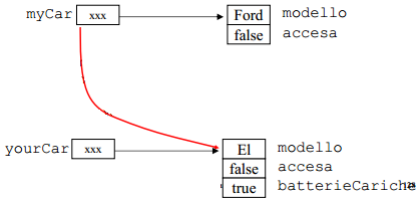
```
public class usaAutomobile {
    public static int partenza(Automobile p) {
        if (p.puoPartire()) p.accendi();
    }
    ...
    public static void main(String args[]) {

        Automobile myCar = new AutomobileElettrica("T"); \\ legale!!
        partenza(myCar); //funziona anche con AutomobileElettrica
    }
}
```

Nella variabile mycar di tipo automobile viene allocato il riferimento ad un oggetto di tipo Automobile ma costruito col costruttore di AutomobileElettrica: il tipo statico di myCar è **Automobile** quello **dinamico** è **AutomobileElettrica.** A run-time, l'automobile sarà quindi vista come una Automobile Elettrica utilizzando **i metodi e gli oggetti definiti per quella classe**. Il compilatore infatti verifica che ogni oggetto venga manipolato correttamente **solo in base al tipo apparente**. Vediamo un altro esempio:

```
class AutomobileElettrica extends Automobile {
```

```
    boolean batterieCariche;
    void ricarica() {
        batterieCariche=true;
    }
    void accendi() {
        if(batterieCariche)
            accesa=true;
        else
            accesa=false;
    }
}
... // in un'altra classe
Automobile myCar = new Automobile();
AutomobileElettrica yourCar = new AutomobileElettrica();
...
myCar = yourCar;
```

Mycar si riferisce al secondo oggetto, questo è lecito. L'assegnamento è possibile ed il tipo statico di myCar è Automobile, quello dinamico è AutomobileElettrica. Questo è quello che succede in memoria



Se fosse invece uguale il contrario, ovvero

```
yourCar = myCar;
```

Il tipo effettivo potrebbe non essere sottotipo del tipo apparente: quindi **non sarebbe usabile**.

7.2 Esempio di chiamata dei metodi:

```
Automobile myCar = new AutomobileElettrica();
... myCar.puoPartire(); //ok, chiama metodo di Automobile
myCar.accendi(); //ok, chiama metodo di AutomobileElettrica
myCar.ricarica(); //KO, ricarica non è metodo di Automobile
AutomobileElettrica yourCar = new AutomobileElettrica ();
yourCar.ricarica(); //OK, chiama metodo di AutomobileElettrica
```

Se avessimo avuto come prima riga di codice normalmente

```
Automobile myCar = new Automobile();
```

myCar.accendi avrebbe chiamato il metodo di Automobile!

In Java a fronte l'invocazione la chiamata del metodo **dipende quindi dal metodo dinamico** e non da quello statico. Il legame (binding) tra il metodo invocato e il metodo attivato è dinamico, **dipende dal tipo attuale dell'oggetto**. Dispatching è la tecnica usata per realizzare binding dinamico

```
public class usaAutomobile {

    public static void main(String args[]) {
        Automobile a1 = new Automobile("Ford");
        Automobile a2 = new AutomobileElettrica("T");
        a1.accendi(); // a run time chiama implementazione di Automobile
        a2.accendi(); // a run time si chiama implementazione di AutomobileElettrica
        partenza(a2); // solo a run time si può conoscere il tipo effettivo
    public static int partenza(Automobile a) {

        a.accendi(); //funziona anche con AutomobileElettrica
            ...
        }
}
```

Il compilatore **non ha modo di sapere quale metodo accendi** verrà chiamato. Il tipo del parametro è infatti noto solamente a run-time: il compilatore allora **non genera il codice per eseguire il metodo, ma genera codice che cerca qual'è l'implementazione giusta** di accendi **in base al tipo effettivo** di p e la esegue. Questa tecnica è detta **dispatching**.

7.3 Overloading e overriding

Il binding dinamico non va confuso con l'overloading. Esempi:

```
class Punto2D{

    public float distanza(Punto2D p) {...}

}
class Punto3D extends Punto2D {


    public float distanza(Punto 3D p){...} // OVERLOADING !!!

}
```

Il metodo distanza di Punto3D **ha una segnatura diversa da quella di distanza** dichiarato in Punto2D: **NON è overriding e non si applica binding dinamico**

```
Punto2D p = new Punto3D();
p.distanza(p); // chiama Punto2D.distanza(Punto2d)!!!
```

8 Classi e Tipi

8.1 Linguaggio fortemente tipizzato

L'obiettivo del Java è quello di evitare tutti i cosiddetti "errori di tipo". Quali sono gli errori di tipo? Essenzialmente possiamo dividerli in due categorie

- Assegnamenti tra oggetti di metodo incompatibile
- Chiamate di metodi incompatibili (numero e tipo parametri)

Se il compilatore può garantire che **NON sorgano errori di tipo** durante l'esecuzione si dice che il linguaggio è sicuro riguardo ai tipi (type safe). Quando ciò accade si dice che il linguaggio è **fortemente tipizzato**. Rispetto al C o al C++, infatti, in Java abbiamo già visto che viene fatta automaticamente **garbage collection** e viene controllato che **gli indici di un array non escano dai confini**.

8.2 Gerarchia di Tipi

- Una **classe** definisce un **tipo**.
  - Una **sottoclasse** definisce un **sottotipo**.
    - Un oggetto del **sottotipo** è sostituibile ad un oggetto del tipo.
- Si distingue tra
  - Tipo **statico**: il tipo dichiarato
  - Tipo **dinamico**: il tipo effettivo a run-time

Il compilatore verifica che ogni oggetto venga correttamente manipolato **in base al tipo statico**. Il linguaggio **garantisce che a run time non sorgono errori se invece si opera su un oggetto il cui tipo dinamico è un sottotipo del tipo statico.**

9 Dettagli sull'ereditarietà

9.1 Accesso ai membri private

Le sottoclassi non possono accedere agli attributi (e metodi) private delle sopraclassi! E' sbagliato scrivere:

```
public void accendi() {
    if(batterieCariche)
       accesa=true;
    ...;
}
```

perché accesa è private nella sovraclasse!

9.2 Overriding e pseudo-variabile super

All'interno (e solo all'interno) di un metodo della sottoclasse **ci si può riferire ai metodi della sopraclasse tramite la notazione**:

```
super.<nome metodo>(<lista par. attuali>)
```

Esempio:

```
public class AutomobileElettrica extends Automobile {

    ...
    public void accendi() { // OVERRIDING
        if(batterieCariche)
            super.accendi();
        else

            System.out.println("Batterie scariche");
    }
}
```

9.3 Accesso ed ereditarietà

Attributi e metodi di una classe possono essere:

- **public**
  - sono **visibili a tutti**
  - vengono ereditati
- **private**
  - sono **visibili solo all'interno della stessa classe**
  - non sono visibili nelle sottoclassi, ma sono ereditati
- **protected**
  - sono **visibili solo alle sottoclassi e alle classi nello stesso package**
  - vengono ereditati
- "**friendly**":
  - sono **visibili a tutte le classi nello stesso package**
  - vengono ereditati, ma sono visibili alle sottoclassi che stanno nello stesso package della classe

Riepilogando:

	Public	**Friendly**	**Protected**	**Private**
Ereditato in stcl	SI	SI	SI	SI
Visibile in stcl interna al package	SI	SI	SI	NO
Visibile in stcl esterna al package	SI	SI	NO	NO
Visibile a tutti	SI	NO	NO	NO

9.4 Ereditarietà dei costruttori

I costruttori **non sono ereditati** perche' occorre inizializzare anche i nuovi attributi (costruttore della superclasse non li inizializza)

Per inizializzare gli attributi private ereditati, all'interno di un costruttore **è possibile richiamare il costruttore della sopraclasse tramite la notazione:**

```
super(<lista di par. attuali>)
```

posta come **prima istruzione del costruttore.** Se il programmatore non chiama esplicitamente un costruttore della sopraclasse, **il compilatore inserisce automaticamente il codice che invoca il costruttore di default della sopraclasse** (che potrebbe non esistere!)

In AutomobileElettrica:

```
public AutomobileElettrica(String modello) {


    super(modello); //qui inizializza modello e accesa
    batterieCariche=false;

}
```

9.5 La classe Object

La classe Object **è la classe da cui tutti gli oggetti ereditano automaticamente**. Fornisce alcuni metodi tra cui i seguenti

9.5.1 Metodo Equals

```
public boolean equals(Object)
```

Dice se due oggetti sono equivalenti. Che cosa ciò esattamente significhi **dipende dal tipo dell'oggetto**. Per esempio, due set sono equivalenti **se contengono gli stessi elementi, indipendentemente dall'ordine di inserimento**, va pertanto **spesso ridefinita. L'implementazione fornita da Object coincide con "==", ovvero riferimenti uguali**.

9.5.2 Metodo toString

```
public String toString();
```

Restituisce una rappresentazione testuale dell'oggetto. L'implementazione fornita da Object è quella di **una stringa che mostra il nome del tipo seguito da una rappresentazione testuale del valore dell'oggetto.**

9.5.3 Metodo Clone

```
public Object clone();
```

Il metodo clone **restituisce una copia dell'oggetto**, nello stesso stato. Viene fornita una **copia di tutti i campi**.

Attenzione: se si vuole che classe fornisca clone, occorre scrivere:

```
implements Cloneable
```

altrimenti la chiamata alla clone() fornita da Object provoca eccezione.

Se si vuole fare una copia "completa" dell'oggetto, il metodo clone va ridefinito: la clone fornita da Object fa una "copia superficiale", ovvero **se un attributo è un riferimento, viene copiato il riferimento**!

9.6 Classi e information hiding

Attributi e metodi di una classe

- se non dico nulla sono **visibili all'interno del package**: chiamato informalmente "friendly"
- se no, possono essere:
  - **public**:
    - visibili a tutti
  - **protected**:
    - visibili all'interno del package
    - visibili alle sottoclassi anche se esterne al package
  - **private**
    - visibili solo all'interno della stessa classe

9.7 Classi e metodi astratti

Un **metodo astratto** è un **metodo per il quale non viene specificata alcuna implementazione**.

Una classe **è astratta se contiene almeno un metodo astratto**. La classe astratta si definisce con la parola chiave "abstract" davanti a class.

Se una classe è astratta, **non è possibile creare istanze** per questa classe. Prendiamo l'esempio delle forme visto al 6.5:

```
abstract class Shape {

    static Screen screen;
    Shape(Screen s) {
        screen=s;
    }
    abstract void show();

}
class Circle extends Shape {

    void show() {
        ...
    }

}
...
Shape s=new Shape(); // errato
Circle c=new Circle(); // corretto
```

Perché questo? Perché **se io non definissi un metodo show nella superclasse, potrei invocare il metodo show nella sottoclasse** da cui questa eredita. Ma io voglio ugualmente definire il metodo nella classe perché **so che questo metodo deve essere ereditato da tutte le sottoclassi.** So anche, però, che se questo metodo, che va specificato nelle sottoclassi venisse definito semplicemente con

```
void show();
```

vuota nella superclasse, potrebbe accadere che se mi dimentico di implementarlo in una sottoclasse, il compilatore compili non rilevando errori ma il metodo non fa nulla in quanto vuoto: **l'abstract quindi ci "forza" all'overriding del metodo in tutte le sottoclassi.**

9.8 Classi e metodi final

Se **vogliamo impedire che sia possibile creare sottoclassi di una certa classe** la definiremo final.

```
final class C {...}
class C1 extends C // errato
```

Similmente, se **vogliamo impedire l'overriding di un metodo** dobbiamo definirlo final

```
class C {
    final void f() {...}
}
class C1 extends C {
    void f() {...} // errato
}
```

10 Limiti dell'ereditarietà semplice

10.1 Le interfacce

L'ereditarietà semplice non permette la descrizione di numerose situazioni reali:

Supponiamo per esempio di avere una classe Giocattolo ed una classe Automobile: **in assenza di ereditarietà multipla** non posso definire la classe AutomobileGiocattolo. Java **distingue tra ereditarietà semplice e multipla utilizzando il costrutto interface**.

Una interfaccia è come una classe che può avere solo attributi costanti e **i cui metodi sono tutti pubblici ed astratti**.

```
interface <nome> {
<lista di definizione di attributi costanti e metodi privi di corpo>
}
```

Gli attributi di una interfaccia sono quindi implicitamente **visibili alla classe che la implementa e immodificabili** (static final). Vediamo un esempio di definizione interfaccia con tre attributi statici e un metodo implicitamente astratto:

```
interface scalable {
    int SMALL=0,MEDIUM=1,BIG=2; //static e final
void setScale(int SIZE);
}
```

10.2 Gerarchia di implementazione

Una sottoclasse **può estendere una sola classe, ma può implementare infinite interfacce.** Essendo i metodi delle interfacce metodi astratti, se la classe non è astratta **deve fornire una implementazione per tutti i metodi presenti nelle interfacce** che implementa, altrimenti la classe è astratta. I metodi con la stessa signatura definiti in più interfacce vanno ovviamente ridefiniti una sola volta.

```
class <nome> extends <nome0>
    implements <nome1 >,...,<nomen> {...}
```

Nel caso particolare di ereditarietà singola, quando conviene ereditare da una classe astratta, che possiamo vedere del tutto equivalente ad una interfaccia, e quando, appunto implementare una interfaccia? Tendenzialmente **conviene ereditare da una classe qualora vi sia almeno un pezzetto di codice comune a tutte le sottoclassi**.

11 Conversioni automatiche di tipo

11.1 Promozioni automatiche

Java effettua automaticamente le seguenti promozioni di tipo qualora vi siano delle operazioni coinvolte

```
byte -> short, int, long, float, double
short -> int, long, float, double
int -> long, float, double
long -> float or double
float -> double
char -> int, long, float, double
```

Riepilogandole in tabella:

Tipo	short	int	long	float	double
byte	SI	SI	SI	SI	SI
short	-	SI	SI	SI	SI
int	NO	-	SI	SI	SI
long	NO	NO	-	SI	SI
float	NO	NO	NO	-	SI
char	NO	SI	SI	SI	SI

Queste vengono effettuate in maniera automatica dal compilatore considerato che i tipi sono compatibili tra loro. Facciamo qualche esempio

```
int a=2, c=0;
float b=2.1;
c = a + b;
```

C varrà 2.1 e diventa di tipo float.

Attenzione alle conversioni: ad esempio nella divisione tra interi

```
int a=2, b=5;
float c=0.0;
c = b/a;
```

Essendo c il risultato di una divisione tra interi, **nonostante sia dichiarato float, varrà 2**. Se vogliamo essere sicuri che il risultato sia float, possiamo agire in questo modo:

```
int a=2, b=5;
float c=0.0;
c = b*1.0/a;
```

In questo modo il numeratore sarà float che diviso per un int, **mantenendo il tipo più "grande", darà il risultato float**.

11.2 Casting Esplicito

E' possibile forzare una conversione di tipo attraverso l'operatore di casting:

```
(<tipo>)<espressione>
```

Tra operatori primitivi sono consentite le seguenti operazioni di casting:

```
short -> byte, char
char -> byte, short
int -> byte, short, char
long -> byte, short, char, int
float -> byte, short, char, int, long
double -> byte, short, char, int, long, float
byte -> char
```

Riepilogandole in tabella:

Tipo	**byte**	**short**	**int**	**long**	**float**	**char**
short	SI	-	NO	NO	NO	SI
int	SI	SI	-	NO	NO	SI
long	SI	SI	SI	-	NO	SI
float	SI	SI	SI	SI	-	SI
double	SI	SI	SI	SI	SI	SI
char	SI	SI	NO	NO	NO	-

Riepiloghiamo quindi le conversioni di tipo possibili tra i diversi tipi. C = casting, A = automatica.

Tipo	byte	short	int	long	float	double	char
byte	-	A	A	A	A	A	C
short	C	-	A	A	A	A	C
int	C	C	-	A	A	A	C
long	C	C	C	-	A	A	C
float	C	C	C	C	-	A	C
double	C	C	C	C	C	-	C
char	C	C	A	A	A	A	-

È possibile **forzare esplicitamente la conversione da un tipo riferimento T ad un sottotipo T1** purché **il tipo dinamico dell'espressione che convertiamo sia un sottotipo di T1**. Esempio:

```
Animale a = ...;
Gatto mao = ...;
a = mao; // OK, assegnazione polimorfica
mao = (Gatto)a; // corretto (casting) perché a è un gatto
//se non fosse un Gatto è eccezione (vedremo)
```

11.3 Metodo istanceof

Possiamo evitare errori a run-time utilizzando il metodo **instanceof**:

```
Object a;
int i = System.in.read();
if (i>0)

    a = new String();

else

    a = new Integer();

if (a instanceof String)

    return a.equals("abcd");
```

Il metodo confronta l'oggetto in a e il suo tipo con l'oggetto definito a destra eseguendo l'operazione solamente se a è String. Vediamone un utlizzo nella ridefinizione della classe equals (vedi 9.5.1)

```
class Data {

    private int giorno;
    private int mese;
    private int anno;
    ...
    public int leggi_giorno(){
        ...
```

```
    }
    public boolean equals(Object o) {
        if (!(o instanceof Data)
            return false;
        Data d=(Data) o;
    return (giorno==d.giorno && mese==d.mese && anno==d.anno)
    }
}
```

Perché ridefinire l'equals così, e non ad esempio, così?

```
public boolean equals(Data o) {
return (giorno==d.giorno && mese==d.mese && anno==d.anno)
}
```

Il motivo è semplice: il primo equals **effettua overriding sull'equals della classe object, e può essere chiamato per ogni tipo di oggetto**. Restituisce false qualora l'oggetto non sia una data. Con il codice appena descritto, invece, se chiamassi equals per un due oggetti che non sono date, **andrei a chiamare l'equals della classe object da cui tutte le classi ereditano** e che **me li confronterebbe indipendentemente dal tipo.**

12 Array in una lista

12.1 Arraylist

Gli array in Java, come tutti i linguaggi di programmazione, **hanno una dimensione fissa**. Se a un certo punto un array non basta più a contenere gli oggetti, **non si può "allungarlo": occorre (deallocarlo e) allocarne uno nuovo più capiente: SCOMODO.** Per questo Java definisce gli Arraylist che sono **contenitori estensibili e accorciabili dinamicamente**. Esempio di definizione:

```
ArrayList <Person> team = new ArrayList<Person>();
```

Per estendere un arraylist, non potendo accedere direttamente agli elementi del vettore, si usa il metodo add:

```
team.add(new Person("Bob",...));
team.add(new Person("Joe",...));
```

E' disponibile, nella classe arraylist, un metodo che restituisce automaticamente la dimensione dell'array:

```
int team.size();
```

Non esiste per gli arraylist la notazione a[i] che permette di accedere direttamente all'elemento, ma è necessario usare i metodi get e set:

```
team.get(1); //dà la Persona di nome Joe
team.set(0, new Person("Mary",...)); //sostituisce Mary a Bob
```

Il metodo set **NON deve essere usato per inserire un elemento in una posizione che non c'è, ma solamente per sostituire un elemento.** E' possibile anche rimuovere un elemento col metodo **remove**.

```
team.remove(0); // rimuove Mary, ora ci sono Sue e Bob
```

Le arraylist **sono implementate come array di puntatori**, con tutto ciò che ne consegue per quanto riguarda la complessità. Ad esempio il metodo add e remove esegue una traslazione dell'array (complessità$O(n)$).

12.2 Foreach per le collezioni

Essendo gli arraylist una collezione, è possibile scandirla col ciclo foreach

```
for(Person p: team){

    //fa qualcosa con la persona p

}
```

è equivalente al tradizionale for

```
for(int i=0; i<team.size(); i++){


    Person p = team.get(i); //fa qualcosa con la persona p

}
```

La differenza tra i due rimane evidente: **il primo ciclo accede direttamente all'oggetto, il secondo vi accede tramite contatore**.

12.3 Wrappers e Autoboxing

Gli arraylist (come tutti gli altri contenitori) **possono contenere solo oggetti, NON valori di tipi primitivi**: per memorizzare valori primitivi **bisogna includerli in oggetti delle classi**

```
wrapper ArrayList<Integer> intList = newArrayList<Integer>();
```

In particolare l'istruzione

```
intList.add(7);
```

Viene accettato perchè tradotto (dal compilatore) in

```
intList.add(new Integer(7));
```

questo si chiama AUTOBOXING: viene memorizzato il riferimento ad un oggetto che contiene l'intero "7" e non l'intero 7. Simmetricamente si può scrivere

```
int n = intList.get(3);
```

perchè **il metodo per estrarre il valore primitivo viene aggiunto dal compilatore** che produce

```
int n = intList.get(3).intValue();
```

12.4 Vector nelle versioni precedenti a Java 1.5

Coi Vector (nelle versioni precedenti di Java) **si potevano memorizzare solo oggetti Object**, quindi in aggiunta occorreva **anche un casting prima di effettura un "unwrap"**

```
Vector myVector = new Vector ();
myVector.add(new Integer(5));
int n = ((Integer)myVector.get(0)).intValue();
```

13 Input / Output da Tastiera

13.1 Classe System

Java offre come in C un input / output da tastiera, realizzato tramite la classe System.

13.1.1 Input

Per input: **si costruisce uno scanner collegato allo "standar Input Stream" System.in**, poi si usano i **metodi della classe Scanner:**

```
nextLine();
```

legge la prossima riga,

```
next();
```

legge la prossima stringa fino a uno spazio

```
nextInt();
```

legge il prossimo int

```
nextDouble();
```

legge il prossimo double. Ovviamente valgono anche per float, long, short, char...

```
hasNext();
```

dice se c'è una stringa in input,

```
hasNextInt();
```

dice se c'è una stringa di tipo int in input. Ovviamente analoghi metodi sono definiti per float, long, short, char.

13.1.2 Output

System.out fornisce tre importanti metodi: il primo è **analogo al printf** di C, con le convenzioni di formattazione note. Vi sono in più **print** e **println** per **stringhe** dove è possibile usare l'operatore di concatenazione tra stringhe "+" e dove eventuali numeri vengono convertiti automaticamente in string.

13.2 Esempio di Input / Output

```
import java.util.*;
...
Scanner in = new Scanner(System.in);
System.out.println("Come ti chiami?");
String nome = in.nextLine();
System.out.println("Quanti anni hai?");
int eta = in.nextInt();
System.out.println("Ciao "+nome+" tra un anno avrai "+(eta+1)+" anni");
/*oppure: System.out.printf("Ciao %s tra un anno avrai %d anni", nome, eta+1); */
```

14 Gestione delle Eccezioni

14.1 Definizione di Eccezione

Una procedura (utente o di libreria) **deve poter segnalare l'impossibilità di produrre un risultato significativo o la propria terminazione scorretta**. Ad esempio errore durante l'apertura di un file (il file non esiste) o il calcolo della radice quadrata di un numero negativo.

14.2 Gestione Tradizionale dell'Errore

14.2.1 Terminazione del Programma

E' soluzione comune la terminazione del programma alla prima eccezione. Tuttavia questa soluzione risulta troppo drastica: possono infatti esservi alcuni casi in cui l'eccezione può essere gestita. La gestione delle eccezioni, a rigore, dovrebbe inoltre spettare al chiamante e non al chiamato: la fopen, ad esempio non ha elementi per decidere come continuare.

14.2.2 Restituzione di un Valore Convenzionale che rappresenti l'errore

E' possibile, come soluzione alternativa, restituire un valore di rappresentazione dell'eccezione. Ad esempio in C la fopen restituisce il valore -1 se il file non esiste. Il problema è che questo può non essere fattibile o perché la funzione non restituisce il valore di ritorno, oppure semplicemente perché **qualsiasi valore di ritorno è ammissibile**. Inoltre, in generale, dà poche informazioni riguardo all'errore e condiziona il chiamante, ad esempio, per una funzione fattoriale:

```
z = x + fact(y)
```

Non può essere considerata valida se questa restituisce il valore -1, ma va implementata così:

```
int r = fact(y);
if (r>0)
    z = x + r;
else
    ...
```

14.2.3 Altre soluzioni meno comuni

E' possibile portare il programma in uno stato d'errore usando una variabile globale error. La procedura chiamante potrebbe non accorgersi che il programma è stato portato in uno stato "scorretto“ (dovrebbe verificare ERROR ogni volta che chiama una procedura o esegue un'operazione che potrebbe dare errore), oppure usare una funzione ERROR che in generale diminuisce la leggibilità del programma dato che dovrebbe essere verificata ogni volta.

14.3 Soluzione: gestione esplicita delle eccezioni a livello di linguaggi

Una procedura (definita dall'utente o predefinita) può terminare normalmente (con un risultato valido) o "sollevare un'eccezione" che è un oggetto speciale restituito dalla funzione: **il chiamante può "catturare l'eccezione" all'interno di un blocco di codice** e gestirla ad-hoc ovvero da eseguire solo in caso di eccezioni: le eccezioni possono contenere dati che danno indicazioni al chiamante o al programmatore sul problema incontrato.

Ogni eccezione ha un tipo (classe): ad esempio DivisionByZeroException, NullPointerException sono classi predefinite di Java – le eccezioni possono quindi essere definite dall'utente (personalizzazione) tramite la definizione di nuove classi.

14.4 Gestire le eccezioni in Java

14.4.1 Try e Catch

Un'eccezione può essere catturata e gestita attraverso il costrutto:

```
try {<blocco>} catch(ClasseEccezione e) {<codice di gestione>}
```

Vediamo un esempio

```
try{
x = x/y;
}
catch (DivisionByZeroException e){
//e e' l'oggetto eccezione
//codice per gestire l'eccezione
//Qui è possibile usare e
}
//istruzione successiva, da eseguire se non c'e' stata eccezione o se
//catch è riuscito a "recuperare
```

Se viene lanciata una eccezione durante l'esecuzione del codice nella try **il codice si interrompe.** Se è possibile catturare l'eccezione si esegue il codice nella catch relativa. E' possibile scrivere più di una catch per catturare più di un tipo di **eccezione**. Un ramo catch(Ex e) **può gestire un'eccezione di tipo T se T è di tipo Ex o T è un sottotipo di Ex**

```
try{
    int y = Integer.parseInt(stdin.readLine());
    x = x/y;
}
catch(NumberFormatException e) {
    //qui cattura eccezione generabilie da parseInt
}
catch (ArithmeticException e) {

    //qui cattura tutte le eccezioni aritmetiche, inclusa DivisionByZero
}
catch (IOException e) {
    //qui cattura le eccezioni di IO
}
```

14.4.2 Ramo Finally

E' possibile definire un ramo di codice che viene eseguito ugualmente sia se la try ha successo, sia se viene lanciata una eccezione ed eseguita una delle catch. Tipico esempio: essere sicuri in uscita di chiudere un file, sia che ci sia o che non ci sia un errore.

```
class Prova {
    static void read(String fileName) {
        try {

            FileInputStream f=new FileInputStream(fileName);
            ... // use f
        }
        catch(IOException ex) {
        ...
        }
        finally {
            f.close();
        }
    }
}
```

14.5 Propagazione delle Eccezioni

Se invocando una procedura si verifica un'eccezione:

- si termina l'esecuzione del blocco di codice in cui si è verificata l'eccezione e...
  - se il blocco di codice corrente è un blocco try/catch ed esiste un catch in grado di gestire l'eccezione, si passa il controllo al primo di tali rami catch e, completato questo, alla prima istruzione dopo il blocco.
  - altrimenti si risalgono eventuali blocchi di codice più esterni fino a trovare un blocco try/catch che contenga un ramo catch che sia in grado di gestire l'eccezione, altrimenti...
  - l'eccezione viene propagata nel contesto del chiamante
  - la propagazione continua fino a che si trova un blocco try/catch che gestisce l'eccezione
    - se tale blocco non si trova nemmeno nel main(), il programma termina

14.6 Progettare Procedure con Eccezioni

14.6.1 Dichiarare Eccezioni

E' possibile definire procedure e metodi che costruiscono eccezioni e le restituiscono al chiamante: il fatto che una procedura possa terminare sollevando un'eccezione è **dichiarato nella sua interfaccia per mezzo della clausola "throws"**.

Utilità:

- segnalare un comportamento anomalo incontrato durante l'esecuzione di un'istruzione

```
public static int leggiInteroDaInput () throws IOException
```

- notificare che una precondizione su parametri di un'operazione è stata violata

```
public static int fact (int n) throws NegativeException
```

- restituire un valore convenzionale

```
public static int search (int[] a, int x)
throws NullPointerException, NotFoundException
//@esnures (*restituisce la posizione di a, se esiste, dove si trova x*)
```

14.6.2 Sollevare eccezioni

Per **sollevare esplicitamente un'eccezione, si usa il comando throw,** seguito dall'oggetto (del tipo dell'eccezione) da "lanciare" al chiamante

```
public static int fact (int n) throws NegativeException{
    if (n<0)
        throw new NegativeException();
    else if (n==0 || n==1)
        return 1;
           else
        return (n * fact(n-1));
    }
```

14.7 Eccezioni Checked e Unchecked

L'albero delle eccezioni in Java è strutturato come segue:

- Throwable

- Error
- Exception
  - **Checked Exceptions**
  - Runtime Exceptions
    - **Unckecked Exceptions**

Le eccezioni Checked sono quelle **definite dalla procedura che può sollevarle** (errore a compile-time). Quando una procedura P1 invoca un'altra procedura P2 che può sollevare un'eccezione di tipo Ex (checked), almeno una delle due seguenti affermazioni deve essere vera:

- l'invocazione di P2 in P1 avviene **internamente ad un blocco try/catch** che gestisce eccezioni di tipo Ex (quindi, P1 gestisce l'eventuale eccezione)
- il tipo Ex (o un suo sopra-tipo) fa parte delle eccezioni **dichiarate nella clausola throws della procedura P1** (quindi, P1 propaga l'eventuale eccezione

Le eccezioni unchecked possono invece propagarsi **senza essere dichiarate in nessuna signature di procedura** e **senza essere gestite da nessun blocco try/catch**: può essere meglio includerle comunque in throws, per renderne esplicita la presenza (ma per il compilatore e' irrilevante)

14.8 Definizione di nuove eccezioni

Gli oggetti di un qualunque tipo T definito dall'utente possono essere usati per sollevare e propagare eccezioni, **a condizione che T sia definito come sotto-tipo della classe Exception** (o RuntimeException)

- La definizione della classe che descrive un'eccezione **non differisce dalla definizione di una qualsiasi classe definita dall'utente**. In particolare può possedere attributi e metodi propri (usati per fornire informazioni aggiuntive al gestore dell'eccezione) (per ereditarietà)
- Definizione tipica:

```
public class NewKindOfException extends Exception {

    public NewKindOfException(){
        super();
    }
    public NewKindOfException(String s){ super(s); }

}
```

I due costruttori (il primo è quello normale, il secondo se si vuole passare una stringa di errore) **richiamano semplicemente i costruttori di exception. Esempio di uso:**

```
throw new NewKindOfException(“problema!!!”)
```

Esempio di gestione:

```
try{
    ....
}
catch(NewKindOfException ecc){
    String s = ecc.toString( );
    System.out.printls(s);
}
```

14.8.1 Eccezioni con un costruttore

E’ possibile definire le eccezioni, ad esempio, nel metodo costruttore, per verificare la correttezza del programma. La throw lancia una nuovo costruttore dataillegale

```
public class ProvaEcc {
    public static void main(String[] args) {
        int g,m,a; ... // leggi g, m, a
        try {
            d=new Data(g,m,a);
        }
        catch(Data.DataIllegale e) {

            System.out.println("Inserita una data illegale");
            System.exit(-1);
        }
    }
}
public class Data {


    private int giorno, mese, anno; private boolean corretta(int g,int m,int a) {
    ...
    }
    public class DataIllegale extends Exception {}; //inner class
    public Data(int g, int m, int a) throws DataIllegale {
    if(!corretta(g,m,a)) throw new DataIllegale();
    giorno=g; mese=m; anno=a;
    }
}
```

14.9 Progettare le Eccezioni

14.9.1 Quando utilizzarle

Si può progettare una eccezione per:

- segnalare un comportamento anomalo incontrato durante l'esecuzione di un'istruzione
- notificare che una precondizione su un'operazione è stata violata
- restituire un valore convenzionale

In particolare è sempre meglio utilizzare una eccezione **checked**, per il semplice motivo che **vincola il programmatore a gestirla** una volta implementato un metodo in cui questa è dichiarata nella signature. L'uso delle eccezioni unchecked dovrebbe essere limitato a:

1. Si tratta di eccezioni di tipo aritmetico/logico
2. C'è un modo conveniente e poco costoso di evitare l'eccezione
   (a) Es: le eccezioni aritmetiche: si può sempre, se serve, controllare prima di eseguire il calcolo
   (b) Es: per gli array, le eccezioni di tipo OutOfBoundException possono essere evitate controllando in anticipo il valore dell'attributo length dell'array
3. L'eccezione è usata solo in un contesto ristretto

14.9.2 Masking

Definisco masking la gestione dell'eccezione all'interno del metodo (l'eccezione non si propaga al chiamante). Dopo la gestione dell'eccezione, l'esecuzione continua seguendo il normale flusso del programma

```
public static boolean sorted (int[] a) {

    int prev;
    try {
       prev=a[0];
    }
    //lancia eccezione se array e' vuoto
    //(era meglio check diretto su a)
    catch (IndexOutOfBoundsException e){
       return true;
    }
    for (int i=1; i<a.length; i++) {
       if (prev <= a[i])
```

```
            prev=a[i];
        else
            return false;
    }
    return true;
}
```

Si noti che viene fatto restituire True alla funzione perché un array vuoto è ordinato per definizione.

14.9.3 Reflecting

La gestione dell'eccezione può comportare la propagazione di un'ulteriore eccezione (dello stesso tipo o di tipo diverso). Nell'esempio viene mostrato una eccezione all'interno del codice che cambia il metodo che passa al chiamante:

```
public static int min (int[] a)
throws NullPointerException, EmptyException{

    int m;
    try {
        m=a[0];
    }
    catch (IndexOutOfBoundsException e){
        throw new EmptyException("Arrays.min");
    }
    for (int i; i<a.length; i++) if (a[i] < m) m=a[i];
    return m;

}
```

14.10 Consigli Utili

- Aggiungere ai dati correlati con l'eccezione l'indicazione della procedura che l'ha sollevata (in modo da facilitare l'individuazione delle cause)

  ```
  public static int fact (int n)
  throws NotFoundException{

      ...
      throw new NotFoundException("fact");
      ...

  }
  ```

- Nel caso in cui la gestione di un'eccezione comporti un'ulteriore eccezione (reflecting), conservare le informazioni

```
catch (NotFoundException e){

    throw new NewKindOfException(“procedure.name” + e.toString());
}
```

- sebbene sia possibile scegliere liberamente i nomi delle nuove eccezioni definite, è buona convenzione farli terminare con la parola Exception (**NotFoundException** piuttosto che **NotFound**) •
- Può essere talvolta utile prevedere un package contenente tutte le nuove eccezioni definite (facilita il riuso delle eccezioni). A volte invece conviene definire eccezioni come inner classes