Sorgente : http://holsi.hasanaj.com/
Autore : Holsi Hasanaj
Professore: Paolo Merialdo
Corso di Ingegneria Informatica Roma Tre
Anno di produzione: 2011-2012

Programmazione orientata agli oggetti

Programmazione orientata agli oggetti — 02/03/12

Docente: Paolo Merialdo — Lezione I

merialdo@dia.uniroma3.it

Ricevimento: Martedì 12:00 → 13:00

Sito web: http://merialdo.dia.uniroma3.it

Strumenti: Eclipse IDE for JAVA EE Developers

Il paradigma OO corrisponde ad un interazione tra gli "oggetti"

↳ gli OGGETTI sono dotati di:
- COMPORTAMENTO: metodi (operazioni) che si possono invocare sull'oggetto
- STATO: informazioni memorizzate in variabili
- IDENTITÀ: possibilità di distinguere un oggetto da altri

CLASSE: contiene il comportamento e lo stato che un oggetto può assumere
- da una singola classe possono essere creati molti oggetti
  - → variabili d'istanza: consentono di memorizzare le informazioni di ciascun oggetto (stato dell'oggetto)
  - → costruttori: specificano come inizializzare lo stato di un nuovo oggetto
  - → metodi: specificano le operazioni che determinano il comportamento degli oggetti

ESERCIZIO: CLASSE ATTREZZO

```
public class Attrezzo {
private String nome;
private int peso;

public Attrezzo (String nome, int peso) {
  this.peso = peso;
  this.nome = nome; }

public String getNome () {
  return this.nome; }

public int getPeso() {
  return this.peso; }

public String toString() {
  return this.getNome() + "(" + this.getPeso() + "kg)"; } }
```

STRUTTURA BASE DI UNA CLASSE:

```
Public class ClassName {
  Variabili di istanza
  Costruttori
  Metodi }
```

modificatore di visibilità → Tipo → Nome della variabile

VARIABILI DI ISTANZA: private int peso;

COSTRUTTORI: this rappresenta un riferimento all'istanza che sto manipolando

! è indispensabile fare uso quando il nome del campo a cui si vuole accedere è nascosto dal nome di una variabile locale o di un parametro

→ I costruttori inizializzano lo stato di un oggetto: ① Memorizzano i valori iniziali negli attributi ② spesso ricevono parametri dall'esterno ③ I costruttori hanno lo stesso nome della classe

```
public Attrezzo (String nome, int peso) {
    this.nome = nome;
    this.peso = peso; }
```

→ per convenzione si utilizza sempre lo spazio this quando ci riferiamo a variabili d'istanza

OPERATORE: NEW

- Crea un nuovo oggetto della classe specificata
- Invoca l'esecuzione del COSTRUTTORE
- Ritorna un riferimento all'oggetto creato

ESEMPIO:
attrezzo1 = new Attrezzo ("pala", 7)

PARAMETRI:

sia i costruttori che i metodi ricevono dati attraverso parametri. Inoltre: parametri sono definiti nell'intestazione del costruttore o del metodo

↳ PARAMETRI FORMALI: per indicare i parametri nella definizione di un costruttore o di un metodo
↳ PARAMETRI ATTUALI: (o argomenti) per indicare gli argomenti che vengono passati al costruttore o al metodo quando è invocato

METODI: implementano "il comportamento" di un oggetto

hanno: - intestazione: • Modificatore di accesso
• Tipo valore restituito
• Nome del metodo
• Lista dei parametri formali
- corpo: • definizioni di variabili locali
• istruzioni

Un metodo può restituire un valore:
- void se non restituisce nessun valore

ESEMPIO:

```
public boolean hasAttrezzo (String nomeAttrezzo) {
    boolean attrezzoPresente;
    if(this.attrezzo.getNome().equals(nomeAttrezzo))
        attrezzoPresente = true;
    else
        attrezzoPresente = false;
    return attrezzoPresente; }
```

Programmazione orientata agli oggetti 02/03/12

Lezione I

Nel corpo di un metodo (o di un costruttore) si possono definire VARIABILI LOCALI:
- Servono a memorizzare informazioni utili alla esecuzione del metodo
- sono create al momento in cui il metodo viene chiamato, sono distrutte quando il metodo termina
- Ha VISIBILITÀ: di una variabile locale è il metodo o costruttore entro il quale è definita

! differenze tra una variabile locale e una variabile d'istanza:
- le variabili locali devono memorizzare informazioni che servono esclusivamente all'esecuzione del metodo; create all'invocazione del metodo, distrutte al termine di questo → non sono visibili al di fuori del metodo
- le variabili di istanza: memorizzano informazioni che rappresentano lo stato dell'oggetto; vivono per tutta la vita dell'oggetto; sono visibili a tutti i metodi della classe.

06/03/12

Lezione II

INFORMATION HIDING

SVILUPPO DEL SOFTWARE: → implementatore classi proprie
→ utilizzo di classi di libreria

quando utilizziamo classi di libreria siamo interessati a conoscere solo l'interfaccia: ① elenco di metodi che la classe è in grado di offrire dall'esterno ② modo per utilizzarli

! quando implementiamo una classe dobbiamo documentare l'elenco dei metodi offerti → in più dobbiamo rispettare l'implementazione della classe per garantire un uso corretto

RUOLO DEL PROGRAMMATORE

programmatore - realizzatore: interesse: implementare i servizi della classe
programmatore - utilizzatore: quando utilizza una classe scritta da altri
↳ gli interessa sapere cosa un oggetto può fare non come lo fa

MODIFICATORI DI ACCESSO: possiamo specificare ciò che è accessibile all'esterno e ciò che non lo è

↳ variabili di istanza devono essere sempre PRIVATE

```
private TipoAttributo nomeAttributo;
```

↳ esigenze per accedere in lettura → "accessore" pubblico

```
public TipoAttributo getNomeAttributo()
```

↳ se ci sono esigenze per "modificare"

```
public void setNomeAttributo(TipoAttributo nomeAttributo)
```

public → accesso consentito a chiunque
private → accesso consentito solo all'interno della classe stesso
package → accesso consentito alle classi definite all'interno del package
protected → accesso consentito alla classe stessa, alle sottoclassi e a tutte le classi del package

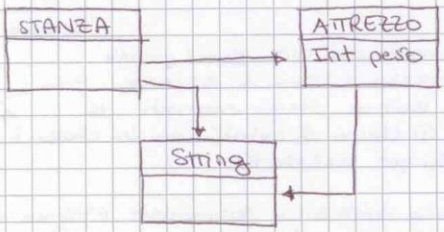
DIAGRAMMA DELLE CLASSI: descrive le caratteristiche principali delle classi che compongono l'applicazione → enfasi è la relazione tra le classi

DIAGRAMMA DEGLI OGGETTI: utilizzato per descrivere l'evoluzione di un programma

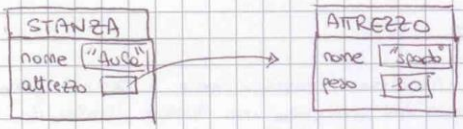
↳ mostra gli oggetti istanziati in memoria durante l'esecuzione dell'applicazione
- ogni oggetto ha un indirizzo di memoria
- le variabili di tipo riferimento memorizzano l'indirizzo dell'oggetto referenziato
- una rappresentazione grafica efficace di questi valori prevede l'uso di frecce che collegano la variabile riferimento all'oggetto referenziato

i valori memorizzati nelle variabili riferimento prevede l'uso di frecce che collegano la variabile riferimento all'oggetto referenziato

ESEMPIO: DIAGRAMMA CLASSI



ESEMPIO: DIAGRAMMA DEGLI OGGETTI



Le variabili possono memorizzare:
- tipi PRIMITIVI
- RIFERIMENTI AD OGGETTI

- Una generica variabile non memorizza direttamente l'oggetto, ma il riferimento dell'oggetto
- Nel caso dei tipi primitivi, il valore è memorizzato direttamente nella variabile

TIPI PRIMITIVI

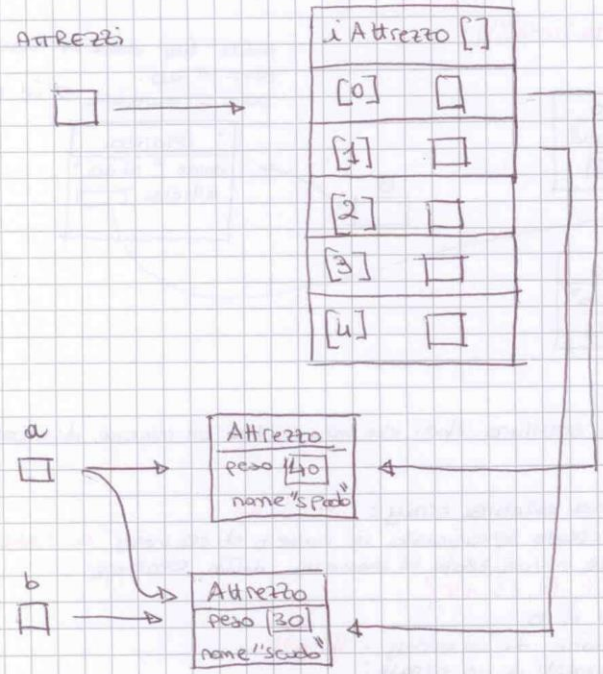
- boolean: vero (true) o falso (FALSE)
- char: caratteri UNICODE 2.1 (16 bit)
- byte: interi a 8 bit (con segno e in C2)
- short: interi a 16 bit " "
- int: interi a 32 bit " "
- long: interi a 64 bit " "
- float: numeri in virgola mobile a 32-bit
- double: numeri in virgola mobile a 64-bit

Programmazione orientata agli oggetti 09/03/12

Lezione III

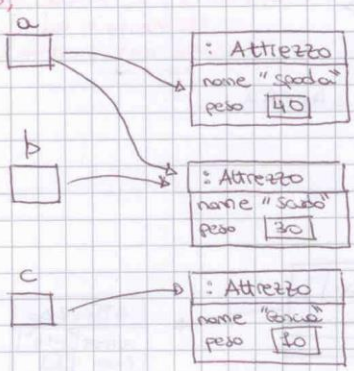
OSSERV:
→ costruire tanti scatolotti quanti sono ee new

ESERCIZIO



ESERCIZIO: → diagramma degli oggetti che rappresenta lo stato degli oggetti referenziati a, b, c al termine dell'esecuzione

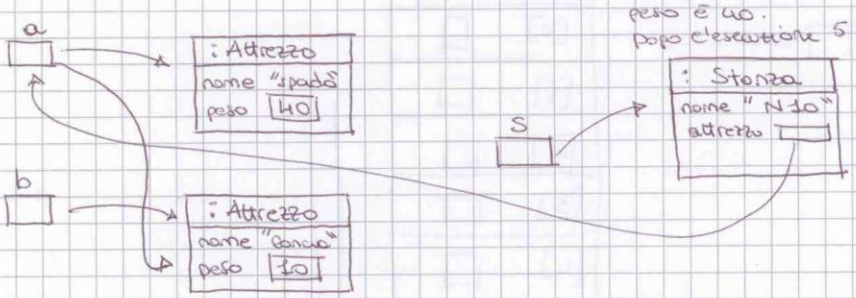
```
Attrezzo a = new Attrezzo ("spada", 40);
Attrezzo b = new Attrezzo ("scudo", 30);
Attrezzo c = new Attrezzo ("banco", 10);
a = b;
```



ESERCIZIO : → Programma oggetti → stato al termine della esecuzione linea 5
Quale valore ha il peso dell'attrezzo che si trova nello stato referenziato dalla variabile **s** al termine dell'istruzione 4 ? Al termine della istruzione 5 ?

```
1. Attrezzo a = new Attrezzo ("spada", 40);
2. Attrezzo b = new Attrezzo ("banco", 10);
3. Stanza s = new Stanza ("N10");
4. s.setAttrezzo (a);
5. a = b
6. System.out.println(s.toString());
```

alla fine dell'istruzione 4 il peso è 40.
dopo l'esecuzione 5 il peso è 10



ARRAY : definisce una struttura dati che memorizza un insieme di valori dello stesso tipo

- Dichiarazione di una variabile array: `int[] a;`
- L'oggetto array va creato specificando il numero di elementi `a = new int[10];`
- Un array può essere inizializzato al momento della creazione
  `int[] a = {21, 12, 23, 15};`
- Il primo elemento è 0
- Per avere la dimensione di un array `.length`
- Scansione degli elementi di un array:

```
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

(!) A PARTIRE DA JAVA 5

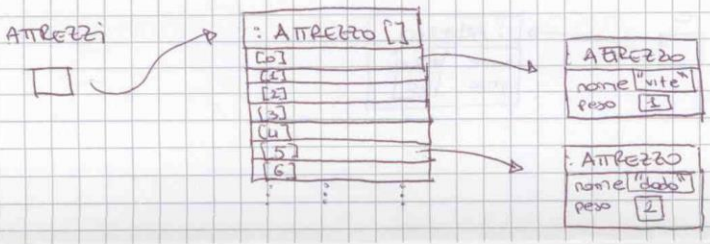
```
for (int elemento : a)
    System.out.println(elemento);
```

equivale ⇒

```
for (int i = 0; i < a.length; i++) {
    int elemento = a[i];
    System.out.println(elemento); }
```

ESEMPIO: DIAGRAMMA DEGLI OGGETTI : ARRAY

```
Attrezzo attrezzi [];
attrezzi = new Attrezzo [10]
attrezzi[0] = new Attrezzo ("vite", 1)
attrezzi[5] = new Attrezzo ("dado", 2)
```

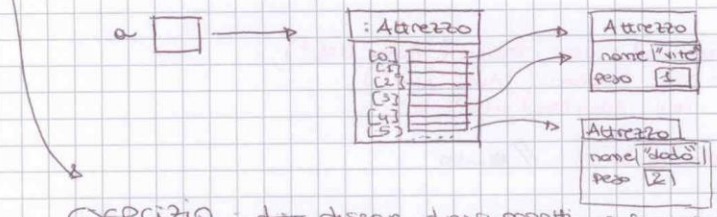


Programmazione orientata agli oggetti 09/03/12

Lezione III

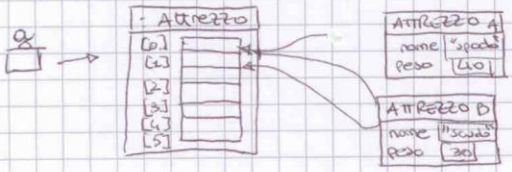
ESEMPIO : DIAGRAMMA DEGLI OGGETTI : ARRAY

```
Attrezzo attrezzi[];
attrezzi = new Attrezzo[10];
attrezzi[0] = new Attrezzo("vite", 1);
attrezzi[5] = new Attrezzo("dado", 2);
attrezzi[3] = attrezzi[0];
```



ESERCIZIO : ~~disg~~ disegn. diagr. oggetti → fino riga 7
quale valore ha ie peso dell'attrezzo referenziato con indice 0 dell'array al termine dell'istruzione 6 e al termine dello 7

```
1. Attrezzo attrezzi[];
2. attrezzi = new Attrezzo[5];
3. attrezzo a = new Attrezzo("spada", 40);
4. attrezzo b = new Attrezzo("scudo", 30);
5. attrezzi[0] = a;
6. attrezzi[1] = b;
7. attrezzi[0] = attrezzi[1];
```



COSTANTI: si utilizzano quando vogliamo imporre che i valori di alcune variabili di istanza non possano essere cambiati

↳ si utilizza la parola CHIAVE `final`

VARIABILI DI CLASSE

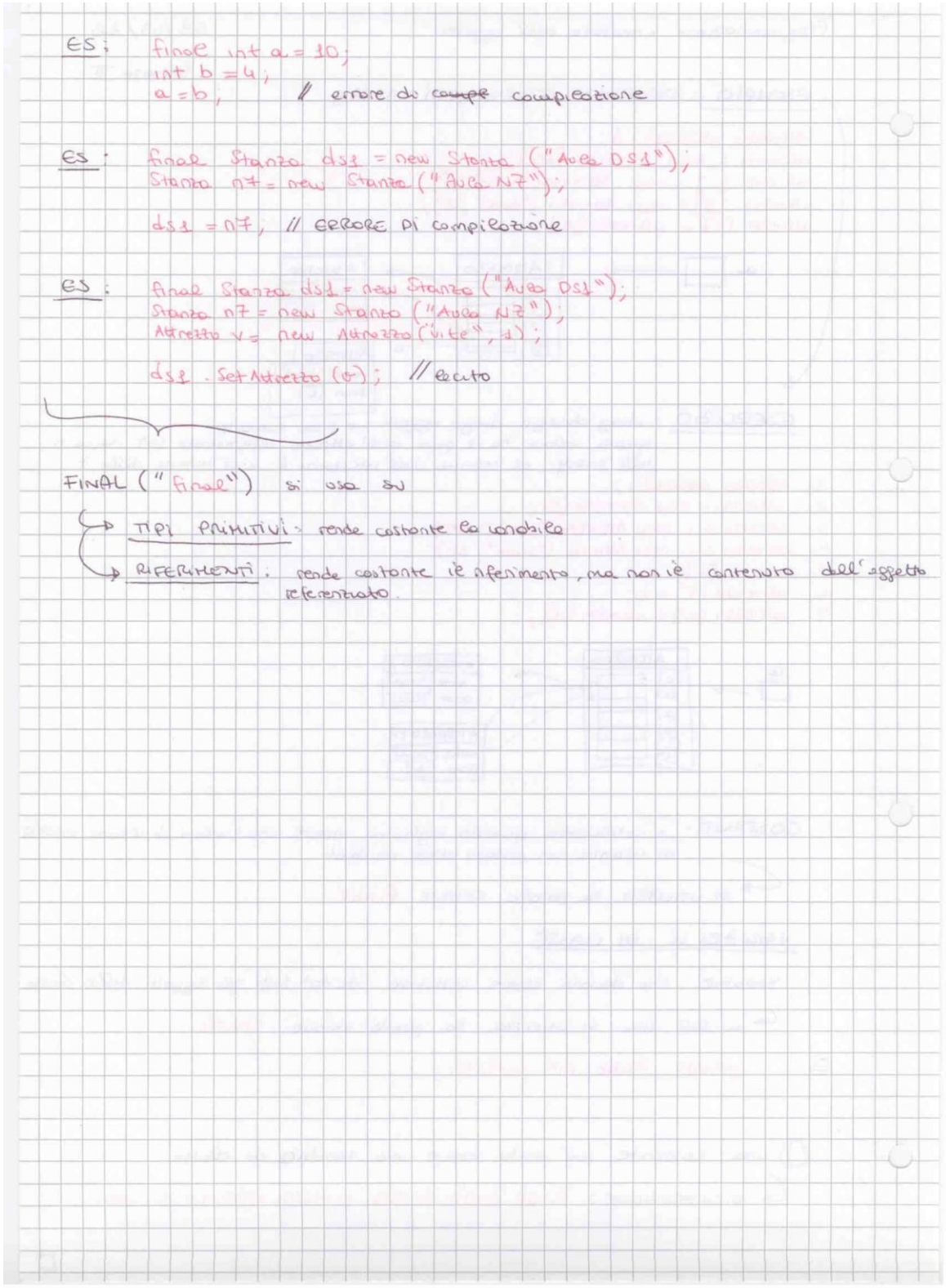
variabili che devono essere condivise da ~~dat~~ tutti gli oggetti della classe

↳ a tale fine si utilizza la parola chiave `static`

Es: `private static int perTutti;`

① una costante può anche essere una variabile di classe

↳ DICHIARAZIONE: `final static double NUMERO_MASSIMO_DI_RETTANGOLI = 4;`


ES: final int a = 10;
int b = 4;
a = b; // errore di compilazione
ES: final Stanza ds1 = new Stanza ("Aula DS1");
Stanza n7 = new Stanza ("Aula N7");
ds1 = n7; // ERRORE di compilazione
ES: final Stanza ds1 = new Stanza ("Aula DS1");
Stanza n7 = new Stanza ("Aula N7");
Attrezzo v = new Attrezzo ("vite", 4);
ds1 . setAttrezzo (v); // ecco
FINAL ("final") si usa su
TIPI PRIMITIVI: rende costante la variabile
RIFERIMENTI: rende costante il riferimento, ma non il contenuto dell'oggetto referenziato.

Programmazione orientata agli oggetti 13/03/12

Lezione IV

CLASSI & OGGETTI

OVERLOADING (SOVRACCARICAMENTO):

→ più metodi possono avere lo stesso nome purché abbiano una segnatura diversa
- Segnatura: nome del metodo e lista dei parametri formali

→ la distinzione del metodo sovraccarico, viene fatta a tempo statico sulla base del parametro attuale

- METODI SOVRACCARICHI -

Costanti letterali di tipo intero: se esiste un metodo che prende come argomento INT, allora viene usato quel metodo, altrimenti si passa a quello più piccolo tra quelli disponibili long, float, double

OVERLOADING DEI COSTRUTTORI:

ogni classe deve avere un costruttore → se non viene definito, le compilatore ne crea uno di default, senza argomenti

avere più costruttori offre più alternative su come inizializzare l'oggetto creato
Tutti i costruttori hanno lo stesso nome (quello della classe) è quindi frequente che i costruttori siano sovraccarichi

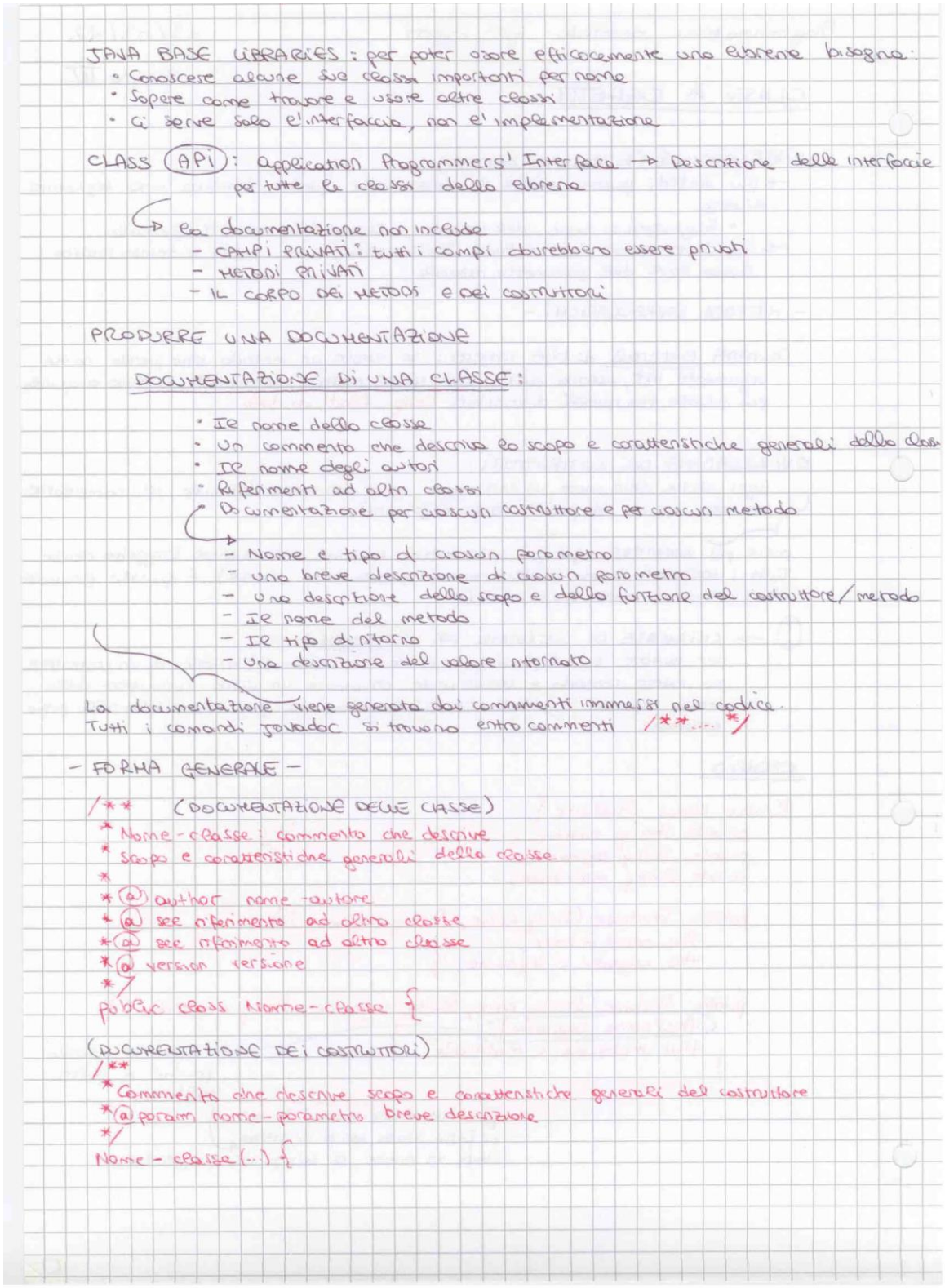
(!) → CHIAMATE DI COSTRUTTORI DAI COSTRUTTORI
per evitare la duplicazione di codice nella definizione di un costruttore può essere comodo e conveniente chiamare un altro costruttore della stessa classe → VINCOLO: la chiamata può essere solo nella prima istruzione

ESEMPIO

```
Public class Studente {
    private String nome;
    private String cognome;
    private String matricola;

    public Studente (String nome, String cognome) {
        this.nome = nome;
        this.cognome = cognome; }

    public Studente (String nome, String cognome, String matricola) {
        this(nome, cognome);
        this.matricola = matricola; }
}
```

→ segue le vincolo poiché è posta all'inizio

fare attenzione poiché se non ci fosse stato ~~non~~ ci sarebbe stato un errore a tempo di compilazione



JAVA BASE LIBRARIES: per poter usare efficacemente una libreria bisogna:
- Conoscere alcune sue classi importanti per nome
- Sapere come trovare e usare altre classi
- Ci serve solo l'interfaccia, non l'implementazione

CLASS (API): Application Programmers' Interface → Descrizione delle interfacce per tutte le classi della libreria

↳ La documentazione non include
- CAMPI PRIVATI: tutti i campi dovrebbero essere privati
- METODI PRIVATI
- IL CORPO DEI METODI E DEI COSTRUTTORI

PRODURRE UNA DOCUMENTAZIONE

DOCUMENTAZIONE DI UNA CLASSE:
- Il nome della classe
- Un commento che descrive lo scopo e caratteristiche generali della classe
- Il nome degli autori
- Riferimenti ad altri classi
- Documentazione per ciascun costruttore e per ciascun metodo
  - Nome e tipo di ciascun parametro
  - Una breve descrizione di ciascun parametro
  - Una descrizione dello scopo e della funzione del costruttore/metodo
  - Il nome del metodo
  - Il tipo di ritorno
  - Una descrizione del valore ritornato

La documentazione viene generata dai commenti immessi nel codice.
Tutti i comandi javadoc si trovano entro commenti /** ... */

- FORMA GENERALE -

```
/**   (DOCUMENTAZIONE DELLE CLASSE)
 * Nome-classe: commento che descrive
 * scopo e caratteristiche generali della classe
 *
 * @author nome autore
 * @see riferimento ad altra classe
 * @see riferimento ad altra classe
 * @version versione
 */
public class Nome-classe {

(DOCUMENTAZIONE DEI COSTRUTTORI)
/**
 * Commento che descrive scopo e caratteristiche generali del costruttore
 * @param nome-parametro breve descrizione
 */
Nome-classe (..) {
```

Programmazione orientata agli oggetti 16/03/12

Lezione

COMMENTAZIONE DEI METODI

```
/**
 * commmento che descrive scopo e caratteristiche generali del metodo
 *
 * @param nome-parametro breve descrizione
 * @return valore di ritorno, breve descrizione
 */
public type nome-metodo () {
```

PACKAGE: è una sorta di raggruppamento che consente di mantere assieme classi concettualmente e logicamente correlate
- → di creare spazi di nomi che evitino ~~ess~~ conflitti
- → fornire un dominio di protezione

Si può accedere alle classi pubbliche di un altro pacchetto in due modi:

1. usando il nome completamente qualificato di una classe
   `java.util.Scanner s = new java.util.Scanner(input);`
2. importando il package e scrivendo direttamente il nome della classe
   `import java.util.*;`
   `Scanner s = new Scanner(input);`

Le classi che appartengono ad un package → dichiarano ed farò appartenenza al package tramite
una classe può appartenere ~~a~~ `package nome-package;` → deve sempre comparire all'inizio del file
a più di un package

(!) Il nome di package deve essere univoco... a tal fine di solito il nome del package comprende il nome del dominio Internet dell'organizzazione scritto in ordine contrario

↳ una classe può essere usata ad di fuori del package solo se è dichiarato pubblico

```
package it.uniroma3.diadia;
public class Stanza {
```

→ la classe Stanza può essere usata ad di fuori del package (importandola)

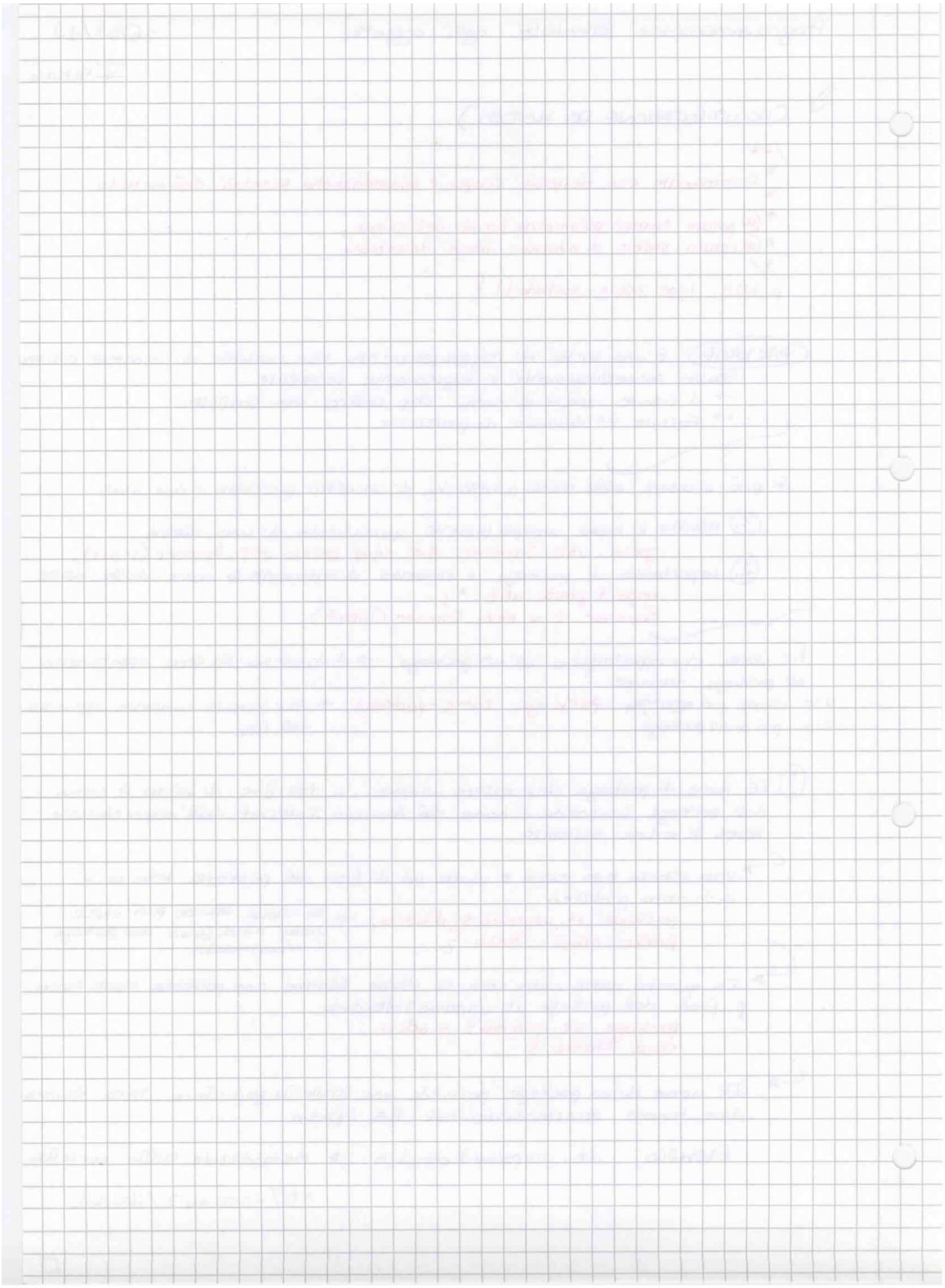
↳ In questo altro altro caso la classe Stanza non potrebbe essere usata di fuori del package it.uniroma3.diadia

```
package it.uniroma3.diadia;
class Stanza {
```

↳ Il nome di un package possiede una struttura gerarchica. Tale struttura deve trovare corrispondenza nel file system

ESEMPIO: it.uniroma3.diadia → memorizzate nella cartella

it/uniroma3/diadia



Programmazione orientato agli oggetto 20/03/12

Lezione VI

CONTROLLO DI VERSIONE

- SVN -

TEAM WORK → evita problemi di mettere insieme il codice

VERSIONAMENTO: si può tornare a più stadi diversi precedenti "salvataggi" in quanto spesso è richiesto accedere ad una versione precedente del Software → Bug introdotti nelle ultime versioni ☹ Perdita dei files causate da modifiche

SVN fornisce uno repository centrale

↳ assembla.com (consigliato)

COORDINATION: più persone lavorano sui stessi files
si vuole inviare le modifiche apportate al resto del team

SVN: o subversion è un sistema di controllo di versione open-source

Fornisce un repository centralizzato:
- Files System ad albero
- Versionamento dei files ad ogni cambiamento
- Accesso controllato

↳ ASSEMBLA: www.assembla.com → fornisce il servizio di repository

ECLIPSE PLUGIN → client SVN integrato all'IDE

- Si possono → condividere progetti dentro Eclipse nel repository SVN
- Import di un progetto condiviso all'interno del repository SVN da un altro membro del team

APPROCCI: se ne possono utilizzare

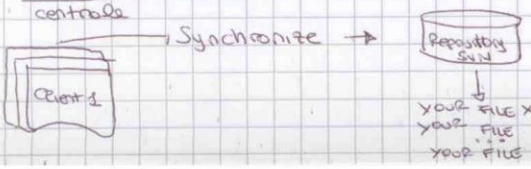
- Ottimistico: le team lavora su files differenti e non ci potranno essere "conflitti" (se si presentano fare il MERGE)
- Pessimistico: le team può lavorare sui stessi files (si usano lock-unlock)

↳ LOCK-UNLOCK: (coordination) - Richiedere i diritti su una risorsa
- Rilasciare i diritti sulla risorsa

COMMIT - UPDATE (COMMIT) & aggiornare la versione nella repository centrale di SVN
dopo le modifiche → Update del progetto di un altro membro del team
consentirà l'aggiornamento dei files alla nuova versione

UPDATE: Aggiornare la versione locale alla versione aggiornata della repository SVN

SYNCHRONIZE: sincronizzazione della versione locale con il repository centrale

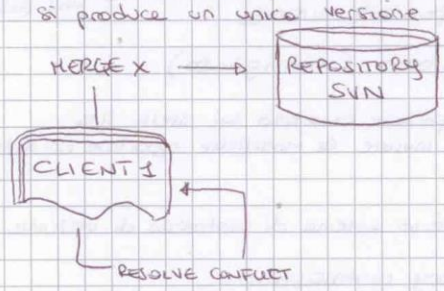


APPROCCIO OTTIMISTICO (MERGE):

Si genera quando ci possono essere conflitti causati dalla modifica dello stesso file da parte di più membri del team

- Dove ci sono conflitti eseguire il MERGE → date due versioni contrastanti come fondere le modifiche

MERGE = risolvere a mano il conflitto (prese due versioni differenti e in conflitto si produce un unica versione fusione)



Programmazione orientata agli Oggetti 20/03/12

Lezione VI

- QUALITÀ DI CODICE: COESIONE & ACCOPIAMENTO -

Il software evolve in continuazione → viene esteso, corretto, mantenuto → portato su altre piattaforme.
Se il costo dell'evoluzione è troppo alto, il software viene gettato.

QUALITÀ DEL CODICE: (dipende)

① **ACCOPIAMENTO**: due o più unità si dicono accoppiate, quando è impossibile, modificare una senza dovere modificare anche le altre
→ L'accoppiamento si riferisce ai legami tra unità separate e distinte di un programma
→ Per un codice di qualità dobbiamo puntare ad un basso accoppiamento
- BASSO ACCOPPIAMENTO: permette di capire il codice di una classe senza leggere i dettagli delle altre. Modificare una classe senza che le modifiche comportino conseguenze sulle altre classi

DUPLICAZIONE DEL CODICE: "è una forma elementare di accoppiamento"
- è sintomo di cattivo progetto
- porta facilmente alla propagazione di errori durante lo sviluppo
- rende difficile la manutenzione

② **COESIONE**: fa riferimento al numero e alla eterogeneità dei compiti di cui una singola unità è responsabile
! Se ciascuna unità è responsabile di un singolo compito, diciamo che tale unità ha un alta coesione: (si persegue un alta coesione)
→ si applica a metodi e classi

COESIONE METODI:
un metodo dovrebbe essere responsabile di un solo compito ben definito

COESIONI CLASSI:
Ogni classe dovrebbe rappresentare un singolo concetto ben definito

→ COESIONE ED ACCOPPIAMENTO: sono due facce della stessa medaglia
- L'alta coesione di una classe si ottiene perseguendo lo scarso accoppiamento di quella classe verso le altre classi
- Lo scarso accoppiamento di una classe verso le altre classi si ottiene perseguendo la alta coesione

REFACTORING: processo di riorganizzazione del codice che si ha poiché le classi e i metodi tendono così a diventare più lunghi e a perdere coesione ed ad aumentare l'accoppiamento
→ La riorganizzazione delle classi e dei metodi deve essere effettuata prima di introdurre nuove funzionalità
→ per assicurarsi di non aver introdotto errori, eseguiamo i test prima e dopo ogni azione di refactoring

LINEE GUIDA:

1. Un METODO è troppo lungo se è responsabile di più di un compito logico
2. una CLASSE è troppo complessa se rappresenta più di un concetto, se ha più di una responsabilità

- QUALITÀ DEL CODICE: INTRODUZIONE ALLE TECNICHE DI TESTING

ERRORI

- ERRORI di SINTASSI: ci vengono indicati dal compilatore
- ERRORI LOGICI: le compilatore non ci aiuta, sono noti anche come "bug"

MOTIVAZIONI DEL TESTING: Il compilatore ci aiuta sugli aspetti statici, tuttavia non dice nulla o quasi sugli aspetti dinamici

BUG: sono errori nell'evoluzione dinamica di un programma su cui il compilatore non ha potuto prevedere e dire nulla
- → il debugging è completamente a carico del programmatore
- → il costo del debugging è ritenuto di gran lunga la componente principale nel costo dei moderni software progetti software

→ COSTO DEL DEBUGGING: il costo di correzione di bug dipende da almeno due grandezze che ne determinano la fattibilità
- le "dimensioni" del contesto → n° di linee di codice in cui il bug può annidarsi
- il "tempo" che il bug impiega a manifestarsi → misura temporale di quanto dista la causa del bug ed il rilevamento dei suoi effetti

CODICE DI TESTING: accanto al codice di produzione si sviluppa sempre del codice di test il cui unico motivo di esistere è quello di verificare la correttezza a tempo di esecuzione del codice principale → il codice di test non fa parte del codice consegnato a fine progetto

TEST UNITARI - UNIT TESTING

- Test su frammenti di un sistema piuttosto che sull'intero sistema (si articola):
  - mettere un "frammento" del sistema in uno stato noto
  - inviare una serie di messaggi noti
  - controllare che alla fine il sistema si trovi nello stato atteso

AUTOMAZIONE DEI TEST → si scrive un programma in cui inizializziamo un certo numero di oggetti con sequenze di test
- Invochiamo il metodo sotto-test e verifichiamo che il risultato ritornato sia uguale a quello atteso

ESEMPIO

```
public static void main(String[] args) {
Sequenza positivi;
Sequenza negativi;
Sequenza negEpos;
Sequenza negEzero;
```

Programmazione orientata agli oggetti 20/03/12

Lezione VI

- QUALITÀ DEL CODICE: INTRO ALLE TECNICHE DI TEST -

ESEMPIO:

```
Sequenza negEzero;
Sequenza inPrimaPos;
Sequenza inUltimaPos;

positivi = new Sequenza(2);
positivi.setElemento(0,1);
positivi.setElemento(2,8);

negativi = new Sequenza(2);
negativi.setElemento(0,-6);
negativi.setElemento(1,-1);

negEpos = new Sequenza(2);
negEpos.setElemento(0,100);
negEpos.setElemento(3,1000)

negEzero = new Sequenza(2);
negEzero = setElemento(0,-1);
negEzero.setElemento(1,0);

inPrimaPos = new Sequenza(2);
inPrimaPos.setElemento(0,1000);
inPrimaPos.setElemento(1,0);

inUltimaPos = new Sequenza(2);
inUltimaPos.setElemento(0,1);
inUltimaPos.setElemento(1,1000);

Boolean esito = true;
esito &= (positivi.massimo() == 8);
System.out.println(positivi.massimo() == 8);
esito &= (negativi.massimo() == -1);

System.out.println(esito); }
```

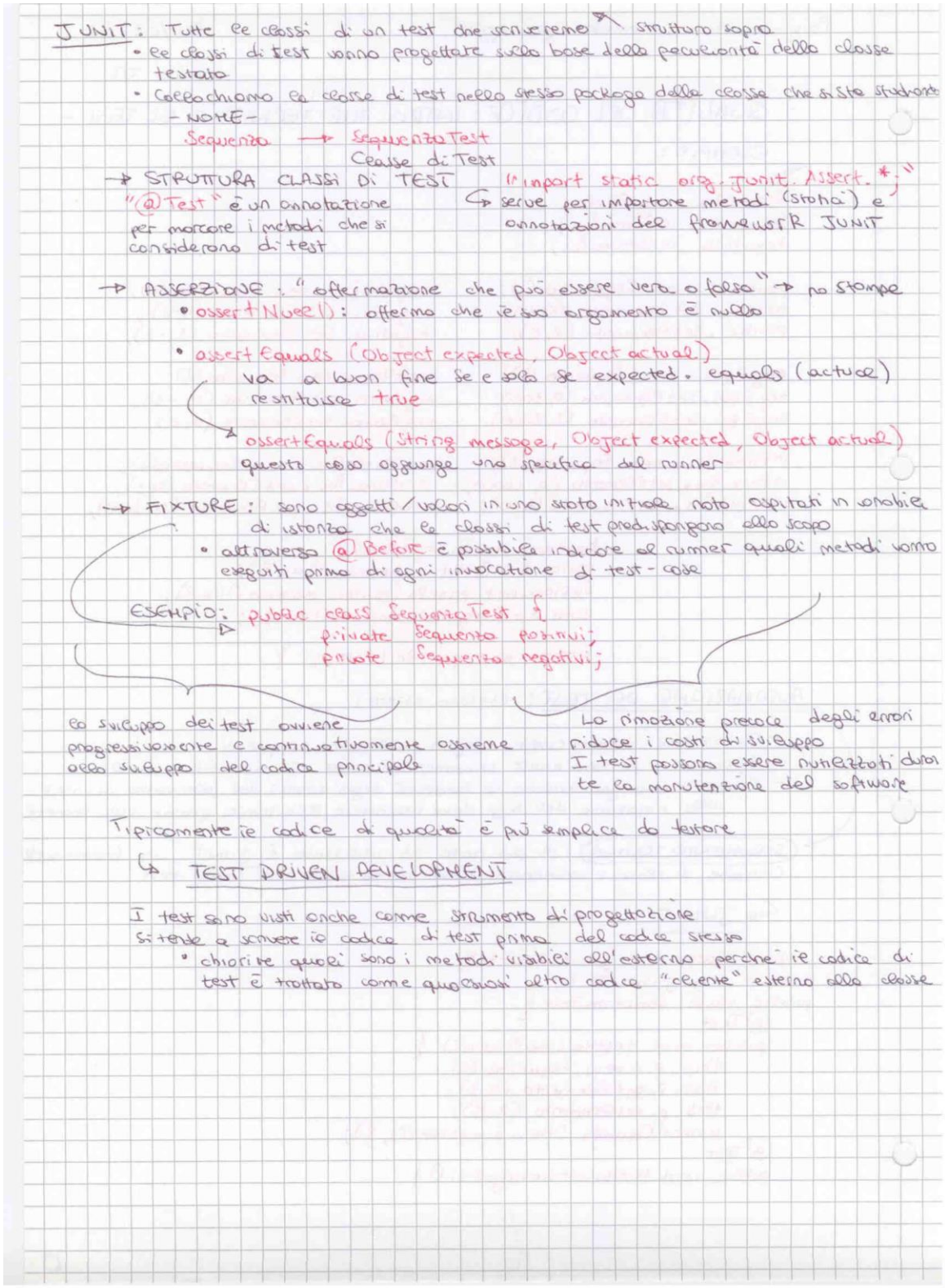
AUTOMAZIONE DEI TEST: (devono essere)

- Automatici: devono essere eseguiti molte volte ad esempio
- Efficienti: devono essere convenienti rispetto alle ispezioni manuali
- Isolati e che garantiscano la località degli errori: dal fallimento di un test alla rimozione del bug deve trascorrere poco tempo grazie alla località

STRUMENTO TESTING: il più noto ed utilizzato è JUNIT, un framework (insieme di classi e convenzioni d'uso) per la scrittura di classi test

↳ JUNIT

```
import static org.junit.Assert.*;
import org.junit.Test;
public class SequenzaTest {
    @Test
    public void testMassimoPositivi() {
        this.p = new Sequenza(2);
        this.p.setElemento(0,1);
        this.p.setElemento(2,8);
        assertEquals(this.p.massimo(),8);
    @Test
    public void testMassimoNegativi() {
```


JUNIT: Tutte le classi di un test che scriveremo → struttura sopra
le classi di test vanno progettate sulla base della pericolosità della classe testata
Collochiamo le classe di test nello stesso package della classe che si sta studiando
- NOME -
Sequenza → SequenzaTest
Classe di Test
→ STRUTTURA CLASSI DI TEST
"@Test" è un annotazione per marcare i metodi che si considerano di test
"import static org.junit.Assert.*;"
serve per importare metodi (statici) e annotazioni del framework JUNIT
→ ASSERZIONE: "affermazione che può essere vera o falsa" → no stampe
assertNull(): affermo che il suo argomento è nullo
assertEquals(Object expected, Object actual)
va a buon fine se e solo se expected.equals(actual) restituisce true
assertEquals(String message, Object expected, Object actual)
questo caso aggiunge una specifica del runner
→ FIXTURE: sono oggetti / valori in uno stato iniziale noto espliciti in modo di istanza che le classi di test predispongono allo scopo
attraverso @Before è possibile indicare al runner quali metodi vanno eseguiti prima di ogni invocazione di test - case
ESEMPIO:
public class SequenzaTest {
private Sequenza positivi;
private Sequenza negativi;
Lo sviluppo dei test avviene progressivamente e continuativamente assieme allo sviluppo del codice principale
La rimozione precoce degli errori riduce i costi di sviluppo
I test possono essere riutilizzati durante la manutenzione del software
Tipicamente il codice di qualità è più semplice da testare
→ TEST DRIVEN DEVELOPMENT
I test sono visti anche come strumento di progettazione
Si tende a scrivere il codice di test prima del codice stesso
chiarire quali sono i metodi visibili dell'esterno perché il codice di test è trattato come qualsiasi altro codice "cliente" esterno alla classe

Programmazione Orientata agli Oggetti. 27/03/12

Lezione VII

1/2

- INTERFACCE E POLIMORFISMO -

In JAVA i riferimenti sono tipati ovvero specificano il tipo dell'oggetto referenziato

ESEMPIO: Strumento S;

S è un riferimento ad un oggetto di tipo Strumento → questo significa che è possibile chiedere di eseguire i metodi offerti dal tipo Strumento all'oggetto referenziato da S

(1) In JAVA come in altri linguaggi orientati agli oggetti si possono definire nuovi tipi
- (1) DEFINENDO NUOVE CLASSI
- (2) CON IL COSTRUTTO interface

INTERFACE: non specifica i dettagli implementativi dei vari servizi → specifica solamente in che modo i servizi possono essere invocati

→ una INTERFACE consiste in una specifica delle segnature (e dei tipi restituiti dai metodi che il tipo può offrire

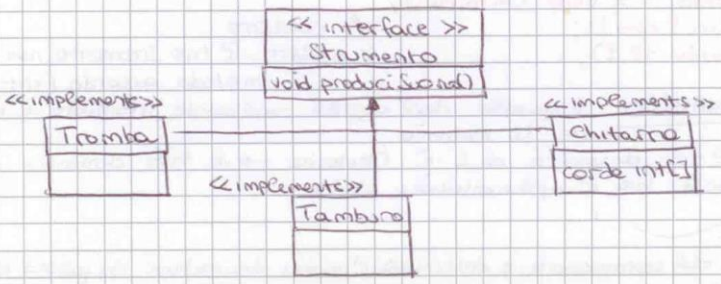
ES:
```
public interface Strumento {
    public void produciSuono(); }
```

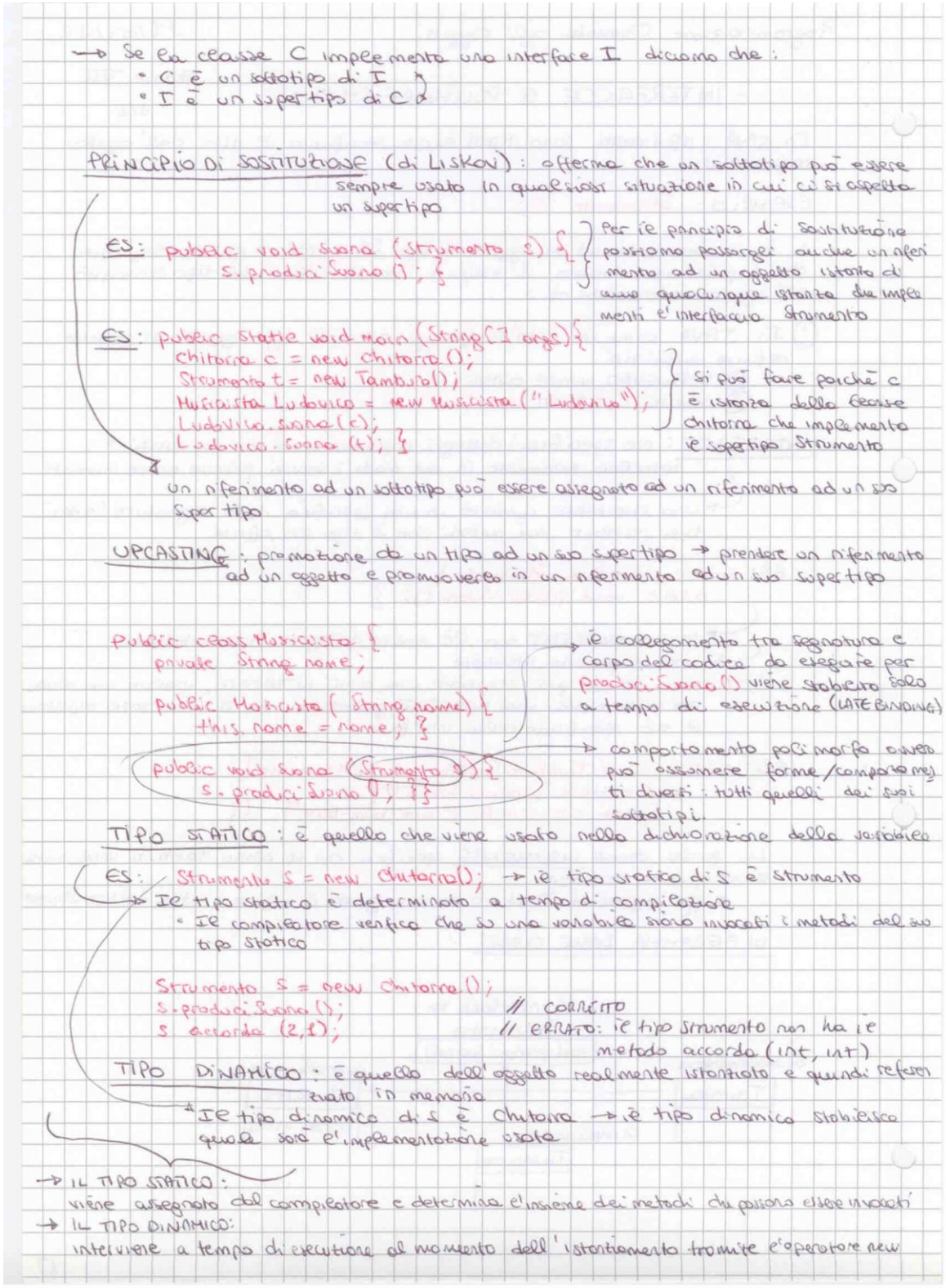
→ In una INTERFACE non c'è nessun dettaglio implementativo
→ non si possono istanziare
→ una classe può istanziare uno o più INTERFACE, inoltre una classe che implementa una INTERFACE garantisce che la sua istanza rispetterà il tipo specificato nella interface

ES:
```
public class Tamburo implements Strumento {
    public void produciSuono() {
        System.out.println("bum-bum-bum"); }}
```

La parola chiave IMPLEMENTS specifica che la classe tamburo implementa l'interfaccia Strumento
- gli oggetti di Tamburo sono in grado di offrire i metodi del tipo Strumento

DIAGRAMMA DELLE CLASSI:




→ Se la classe C implementa una interfaccia I diciamo che:
• C è un sottotipo di I
• I è un supertipo di C
PRINCIPIO DI SOSTITUZIONE (di Liskov): afferma che un sottotipo può essere sempre usato in qualsiasi situazione in cui ci si aspetta un supertipo
ES: public void suona (Strumento s) { s.produciSuono(); }
Per il principio di sostituzione possiamo passargli anche un riferimento ad un oggetto istanza di una qualunque istanza che implementi l'interfaccia Strumento
ES: public static void main (String[] args) {
Chitarra c = new Chitarra();
Strumento t = new Tamburo();
Musicista Ludovico = new Musicista("Ludovico");
Ludovico.suona(c);
Ludovico.suona(t); }
Si può fare perché c è istanza della classe Chitarra che implementa il supertipo Strumento
Un riferimento ad un sottotipo può essere assegnato ad un riferimento ad un suo supertipo
UPCASTING: promozione da un tipo ad un suo supertipo → prendere un riferimento ad un oggetto e promuoverlo in un riferimento ad un suo supertipo
public class Musicista {
private String nome;
public Musicista(String nome) {
this.nome = nome; }
public void suona (Strumento s) {
s.produciSuono(); }
è collegamento tra segnatura e corpo del codice da eseguire per produciSuono() viene stabilito solo a tempo di esecuzione (LATE BINDING)
→ comportamento polimorfo ovvero può assumere forme/comportamenti diversi: tutti quelli dei suoi sottotipi.
TIPO STATICO: è quello che viene usato nella dichiarazione della variabile
ES: Strumento s = new Chitarra(); → il tipo statico di s è Strumento
→ Il tipo statico è determinato a tempo di compilazione
• Il compilatore verifica che su una variabile siano invocati i metodi del suo tipo statico
Strumento s = new Chitarra();
s.produciSuono(); // CORRETTO
s.accorda(2,1); // ERRATO: il tipo Strumento non ha il metodo accorda(int, int)
TIPO DINAMICO: è quello dell'oggetto realmente istanziato e quindi referenziato in memoria
→ Il tipo dinamico di s è Chitarra → il tipo dinamica stabilisce quale sarà l'implementazione usata
→ IL TIPO STATICO:
viene assegnato dal compilatore e determina l'insieme dei metodi che possono essere invocati
→ IL TIPO DINAMICO:
interviene a tempo di esecuzione al momento dell'istanziamento tramite l'operatore new

Programmazione Orientata agli oggetti 27/03/12

Lezione VII

OVERLOADING: (metodi con lo stesso segnatura) problema viene risolto dal compilatore, quindi staticamente 2/2
→ se abbiamo un metodo sovraccarico il compilatore guarda il tipo statico dei parametri per decidere quale è il metodo da invocare

ESEMPIO

```
interface Edificio {
    public int altezza();
}
public class Palazzo implements Edificio {
    private int altezza;
    public Palazzo(int altezza) { this.altezza = altezza; }
    public int altezza() { return this.altezza; }

public class Colloratore {
    public void collora(Edificio e) {
        System.out.println("Colorato Edificio"); }
    public void collora(Palazzo p) {
        System.out.println("Colorato Palazzo"); }

public static void main(String args[]) {
    Palazzo p = new Palazzo(4);
    Edificio e = new Palazzo(3);
    Colloratore c = new Colloratore();
    c.collora(p);
    c.collora(e);
```

tipo statico di p è Palazzo

il tipo statico di e è Edificio

INTERFACCE AS A RUOLO: visto che una classe può implementare più di una interfaccia si può dunque dire che ogni interfaccia, una classe rappresenta un ruolo specifico
- ragionare sui ruoli ci aiuta a produrre codice estremamente riutilizzabile

DOWNCASTING: forzatura sul tipo del parametro. Quando si forza il downcasting la macchina virtuale effettua un controllo a tempo dinamico per verificare che l'operazione sia possibile
- verifica che l'oggetto appartenga al sottotipo a cui si sta forzando il cast

→ in caso contrario errore: java.lang.ClassCastException

INTERFACCE ESTENSIONE
CLASSE OBJECT

13/04/12

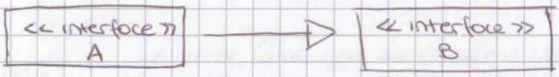
Lezione VIII

Talvolta può essere utile definire una nuova interfaccia a partire da una interfaccia esistente. In JAVA è possibile definendo una interfaccia come una estensione di un'altra interfaccia

ES: `public interface B extends A { public int b2(); }`

↳ Stiamo dicendo che ① B è un sottotipo di A ② B offre tutti i metodi di A più il metodo b2
- cosa quindi il principio di sostituzione

→ e' estensione è un sofisticato meccanismo per definire sottotipi



CREAZIONE DEI TIPI

INTERFACCE: permettono di definire nuovi tipi senza definire l'implementazione dei metodi che formano la specifica del tipo

CLASSI: permettono di definire nuovi tipi; ma richiedono l'implementazione di tutti metodi che formano la specifica del tipo

CLASSI ASTRATTE: Strumento intermedio che permette di lasciare qualche metodo "estratto", ovvero senza implementazione, per consentendone l'implementazione di altri

ESTENSIONE DI INTERFACCE:

- Nuove interfacce definite come estensioni di altre
- L'insieme dei metodi dell'interfaccia estesa comprende quelli della interface base più altri di nuova definizione
- Nessun metodo possiede implementazione delle due interfacce

ESTENSIONE CLASSI

- Nuove classi definite come estensioni di altre
- L'insieme dei metodi della classe estesa comprende quelli della classe base non privati, più altri di nuova definizione.
- I metodi ereditano anche l'implementazione, che se necessario può essere sovrascritta (ovverride) nella classe estesa

- EREDITARIETÀ -

o estensione, si può definire una nuova classe a partire da una classe esistente

- Aggiungendo campi a quelli della classe originale
- Ridefinendo metodi della classe originale

- Rispetto alla classe base, la classe estesa di solito: può avere qualche membro in aggiunta; può definire il comportamento di qualche metodo
- La classe base viene considerata un supertipo della classe estesa

VALIDO IL PRINCIPIO DI SOSTITUZIONE: un'istanza della classe estesa può essere considerata anche come un'istanza della superclasse (ie sottotipo può essere usato al posto di un Supertipo)

- Le variabili di istanza e i metodi vengono ereditati

① Le istanze della classe estesa hanno le stesse variabili della classe base più quelle eventuali aggiunte

② Tutti i metodi pubblici della classe base sono disponibili nella nuova classe senza la necessità di definirli

! solo i membri (variabili di istanza o metodi) pubblici sono accessibili della classe estesa

OVERRIDING: si ha quando alcuni metodi della classe base possono essere non adatti alla classe estesa → comportamento diverso da quello della classe base (riscrittura del metodo)

① Se i metodi della classe estesa vogliono accedere ai campi della classe base devono usare l'interfaccia pubblica della classe base

ES:

```
public class Studente extends Persona {
    private String matricola;
public void setMatricola (String matricola) {
    this.matricola = matricola; }
public String toString () {
    return this.getNome() + " " + this.getMatricola(); }
```

Programmazione Orientata agli Oggetti 13/06/12

Lezione VIII

INTERFACE ESTENSIONE

CLASSE OBJECT

Un'istanza della classe estesa può essere usata al posto di una istanza della classe base → si manifesta il polimorfismo e le legame al codice avviene a tempo di esecuzione. SI PRENDE SEMPRE L'IMPLEMENTAZIONE DEL TIPO DINAMICO

→ COSTRUTTORE CLASSE ESTESA: Deve inizializzare:
- le variabili di istanza ereditate della classe base
- le variabili di istanza proprie

Il costruttore della classe estesa deve delegare l'inizializzazione degli attributi della classe base ad un costruttore della classe base

Quest'operazione si effettua chiamando il costruttore della classe base mediante la parola chiave super() specificando i parametri attuali del costruttore della classe base

EX
```
public class Studente extends Persona {
    private String matricola;
    public Studente(String nome, String matricola) {
        super(nome);
        this.matricola = matricola; }
```
deve sempre essere messo al primo posto

! i costruttori di una classe estesa devono sempre avere una chiamata al costruttore della classe base: se super → se non si effettua esplicitamente la chiamata, il completatore inserisce una chiamata al costruttore "NO-ARG"

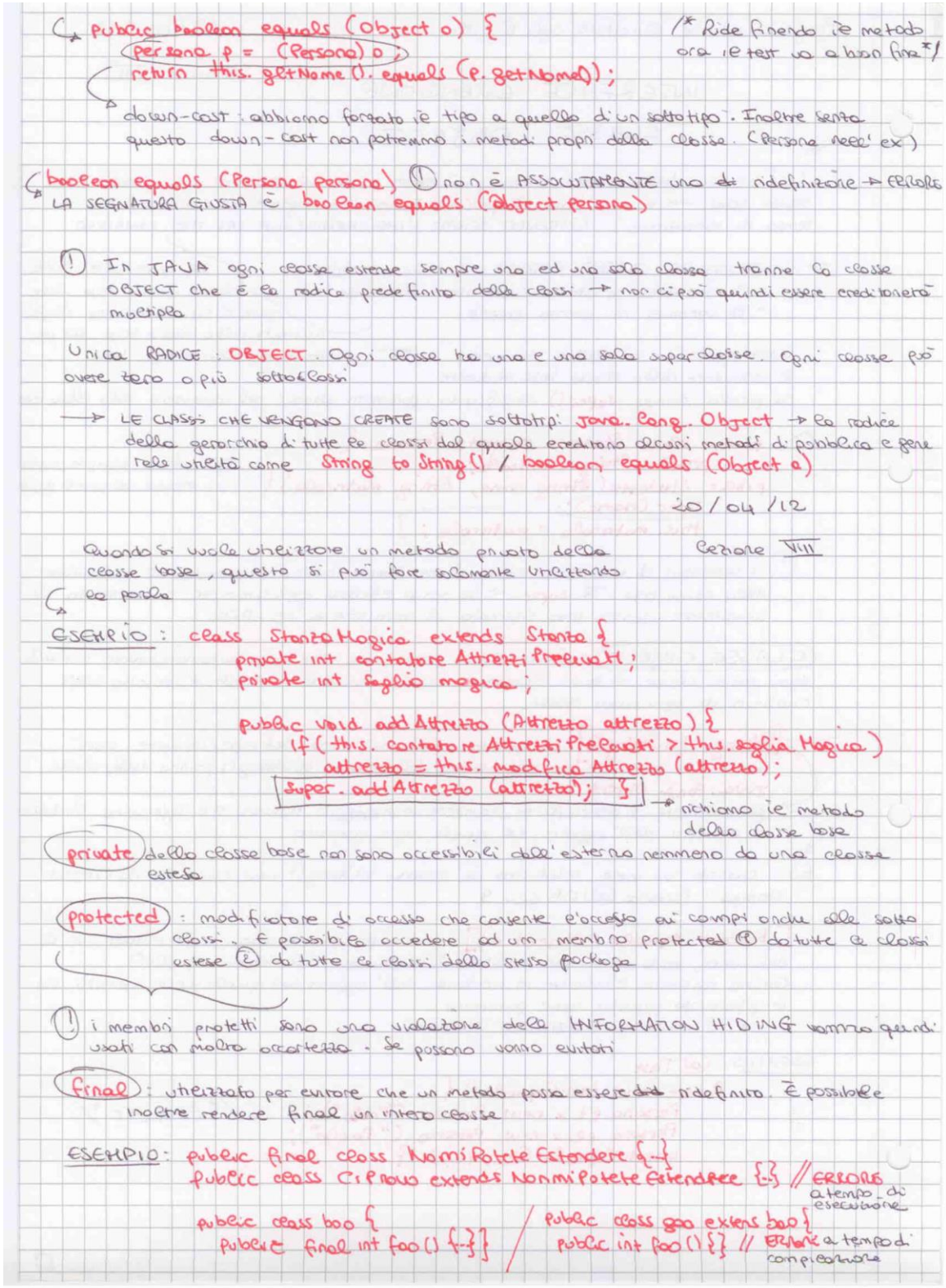
CLASSE OBJECT: è una classe predefinita, che viene automaticamente estesa da ogni nuova classe → ha un insieme di metodi, che sono ereditati e possono essere ridefiniti da ogni nuova classe.

[String toString()]: Se questo metodo non viene det esplicitamente, verrà comunque ereditato la definizione del metodo toString() proprio della classe java.lang.Object
generalmente si preoccupa di stampare un messaggio testuale che rappresenta l'indirizzo in memoria dell'oggetto sul quale viene invocato

ES: Quando non viene ridefinito il metodo toString() viene ereditato da Object
Stampa: Persona@10b62c9

[boolean equals(Object o)]: anche per questo caso se il metodo non è definiamo, verrà comunque ereditato la definizione del metodo equals()
Questo confronta l'indirizzo in memoria dell'oggetto sul quale viene invocato con il riferimento passato come parametro.
→ È essenziale RIDEFINIRLO POICHÉ:

ESEMPIO:
```
@Test
public void testEquals() {
    Persona p1 = new Persona("Paolo");
    Persona p2 = new Persona("Paolo");
    assertEquals(p1, p2); ?
```
/* FALLISCE */


public boolean equals (Object o) {
/* Ride finendo ie metodo ora ie test va a buon fine */
Persona p = (Persona) o;
return this.getNome().equals(p.getNome());
down-cast: abbiamo forzato ie tipo a quello di un sottotipo. Inoltre senza questo down-cast non potremmo i metodi propri della classe (Persona neel' ex)
boolean equals (Persona persona) ! non è ASSOLUTAMENTE una ridefinizione → ERRORE
LA SEGNATURA GIUSTA È boolean equals (Object persona)
! In JAVA ogni classe estende sempre una ed una sola classe tranne la classe OBJECT che è la radice predefinita delle classi → non ci può quindi essere ereditarietà multipla
Unica RADICE: OBJECT. Ogni classe ha una e una sola superclasse. Ogni classe può avere zero o più sottoclassi
→ LE CLASSI CHE VENGONO CREATE sono sottotipi: java.lang.Object → la radice della gerarchia di tutte le classi dal quale ereditano alcuni metodi di pubblica e generale utilità come String toString() / boolean equals (Object o)
20/04/12
Lezione VIII
Quando si vuole ridefinire un metodo privato della classe base, questo si può fare solamente ridefinendo la parola
ESEMPIO: class StanzaMagica extends Stanza {
private int contatoreAttrezziPrelevati;
private int sogliaMagica;
public void addAttrezzo (Attrezzo attrezzo) {
if (this.contatoreAttrezziPrelevati > this.sogliaMagica)
attrezzo = this.modificaAttrezzo(attrezzo);
super.addAttrezzo(attrezzo); }
richiamo ie metodo della classe base
private della classe base non sono accessibili dall'esterno nemmeno da una classe estesa
protected: modificatore di accesso che consente l'accesso ai campi anche alle sotto classi. È possibile accedere ad un membro protected ① da tutte le classi estese ② da tutte le classi dello stesso package
! i membri protetti sono una violazione della INFORMATION HIDING vanno quindi usati con molta accortezza. Se possono vanno evitati
final: utilizzato per evitare che un metodo possa essere ridefinito. È possibile inoltre rendere final un intero classe
ESEMPIO: public final class NonMiPotete Estendere {..}
public class CiProvo extends NonMiPotete Estendere {..} // ERRORE a tempo di esecuzione
public class boo {
public final int foo() {..}}
public class goo extens boo {
public int foo() {} // Errore a tempo di compilazione

Programmazione Orientata agli Oggetti 20/04/12

INTERFACCE ESTENSIONE

Lezione VIII

CLASSE OBJECT

In Java una classe può essere estesa da molte classi, ma ogni classe estende sempre una ed una sola classe (tranne OBJECT) è la radice predefinita delle classi
- NON CI PUÒ ESSERE EREDITÀ MULTIPLA

Se un membro è definito in entrambe le classi base, da quale delle due lo eredita?
Se le due classi base a loro volta estendono una superclasse comune?

Problema legato alle implementazioni che vengono ereditate, per questo una classe può implementare tante interfacce ma può estendere una sola classe

GENERICS

24/04/12

Lezione IX

I GENERICS sono uno strumento per scrivere classi parametriche rispetto ad uno tipo o più di tipi

ESEMPIO: Implementare una classe generica COPPIA

```
Pubblic class Coppia {
        private Object primo;
        private Object secondo;
        pubblic Coppia() {}
        pubblic Coppia(Object primo, Object secondo) {
                this.primo= primo;
                this.secondo = secondo; }
        pubblic Object getPrimo() {
                return this.primo ; }
        pubblic Object getSecondo() {
                return this.secondo; }
        pubblic void setPrimo (Object primo) {
                this.primo = primo; }
        pubblic void setSecondo (Object secondo) {
                this.secondo = secondo; } }
```

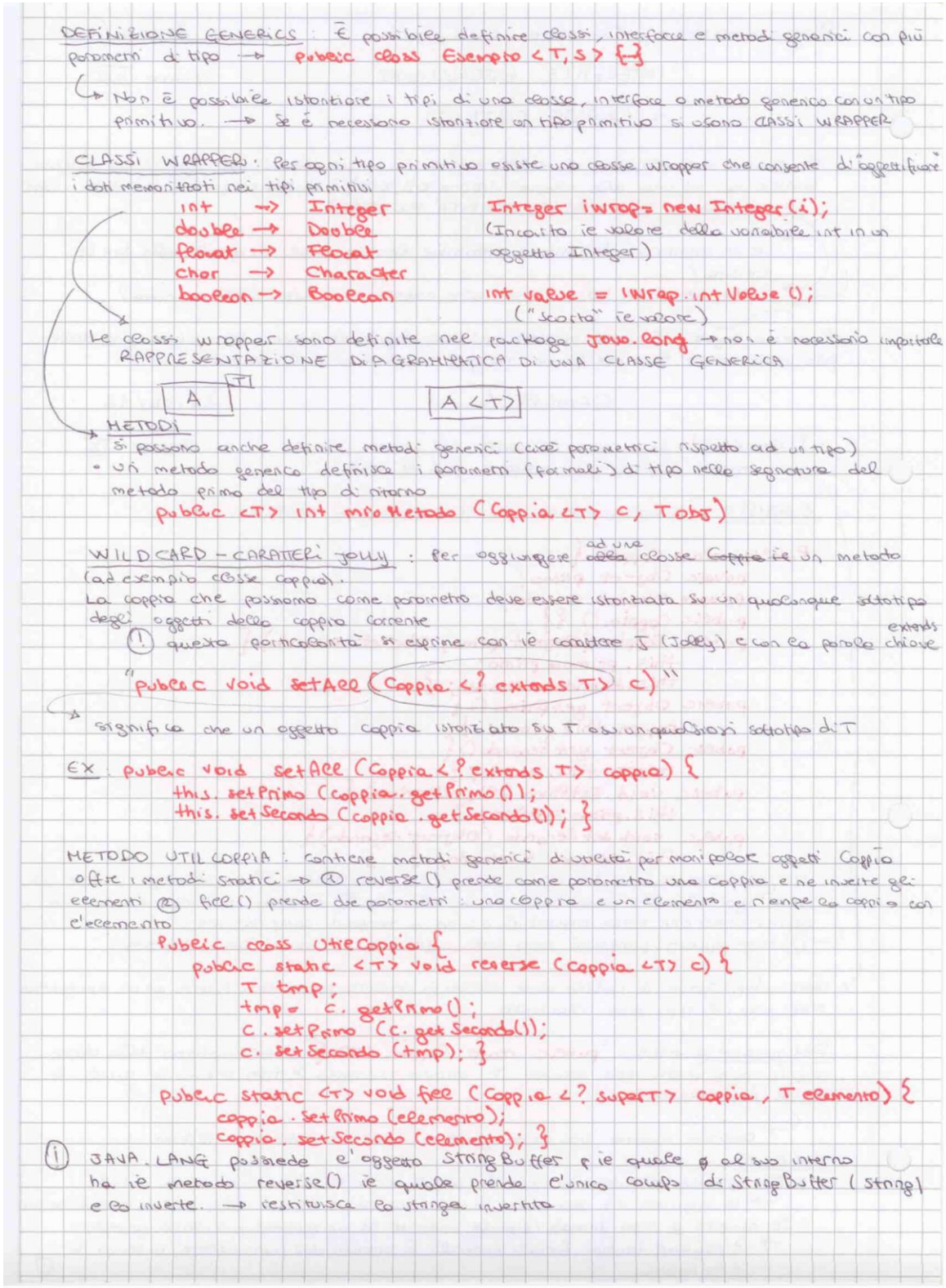
Un controllo laxo dei tipi a tempo di compilazione ① ci costringe a fare un cast ogni volta che accediamo ad un elemento ② rimanda a tempo di esecuzione alcuni errori che erano rilevabili anche a tempo di compilazione.
(!) tutti questi problemi sono stati affrontati e risolti con i Generics

Δ- Nella definizione, il codice viene scritto in maniera parametrica rispetto ad un tipo generico
- Nell'uso, il tipo viene istanziato

Definizione generics "pubblic class Coppia <T> {" → all'interno della classe coppia ogni volta che usiamo T, stiamo indicando il tipo secondo il quale la classe è parametrica

→ Quando usiamo una classe generica, dobbiamo istanziare il tipo

ANALOGIE:
- Il concetto di parametro formale/attuale inerente l'invocazione dei metodi
  → il legame tra parametri formali/attuali è operato dalla JVM a tempo di esecuzione
- Il concetto di tipo formale/attuale inerente la tipizzazione di classi generiche
  → il legame tra tipi formali/attuali è operato dal compilatore a tempo di compilazione



DEFINIZIONE GENERICS: È possibile definire classi, interfacce e metodi generici con più parametri di tipo → public class Esempio<T,S> {..}

↳ Non è possibile istanziare i tipi di una classe, interfaccia o metodo generico con un tipo primitivo. → Se è necessario istanziare un tipo primitivo si usano CLASSI WRAPPER

CLASSI WRAPPER: Per ogni tipo primitivo esiste una classe wrapper che consente di "oggettificare" i dati memorizzati nei tipi primitivi

int →	Integer
double →	Double
float →	Float
char →	Character
boolean →	Boolean

Integer iWrap = new Integer(i);
(Incapsula il valore della variabile int in un oggetto Integer)

int value = iWrap.intValue();
("scarta" il valore)

Le classi wrapper sono definite nel package java.lang → non è necessario importarle

RAPPRESENTAZIONE DI A GRAFICA DI UNA CLASSE GENERICA

METODI
Si possono anche definire metodi generici (cioè parametrici rispetto ad un tipo)
- Un metodo generico definisce i parametri (formali) di tipo nella segnatura del metodo prima del tipo di ritorno

public <T> int mioMetodo (Coppia<T> c, T obj)

WILDCARD - CARATTERI JOLLY: Per aggiungere ad una classe Coppia un metodo (ad esempio classe Coppia).
La coppia che passiamo come parametro deve essere istanziata su un qualunque sottotipo degli oggetti della coppia corrente
(!) questa parametrizzazione si esprime con il carattere ? (Jolly) e con la parola chiave extends

"public void setAcc (Coppia<? extends T> c)"

↳ significa che un oggetto Coppia istanziato su T o su un qualsiasi sottotipo di T

EX: public void setAcc (Coppia<? extends T> coppia) {
this.setPrimo (coppia.getPrimo());
this.setSecondo (coppia.getSecondo()); }

METODO UTILCOPPIA: Contiene metodi generici di utilità per manipolare oggetti Coppia
Offre i metodi statici → ① reverse() prende come parametro una coppia e ne inverte gli elementi ② fill() prende due parametri: una coppia e un elemento e riempie la coppia con l'elemento

```
Public class UtileCoppia {
    public static <T> void reverse (Coppia<T> c) {
        T tmp;
        tmp = c.getPrimo();
        c.setPrimo (c.getSecondo());
        c.setSecondo (tmp); }

    public static <T> void fill (Coppia<? superT> coppia, T elemento) {
        coppia.setPrimo (elemento);
        coppia.setSecondo (elemento); }
```

(i) JAVA.LANG possiede l'oggetto StringBuffer che al suo interno ha il metodo reverse() il quale prende l'unico campo di StringBuffer (String) e lo inverte. → restituisce la stringa invertita

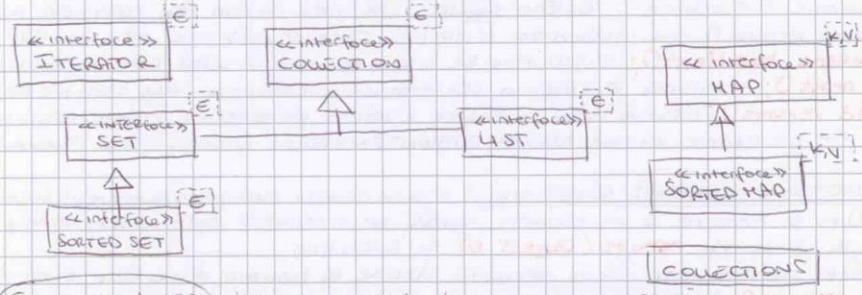
Programmazione Orientata agli Oggetti 24/04/12

COLLEZIONI LISTE GENERICHE

Lezione IX

Gli ARRAY sono uno strumento di basso livello:
- La dimensione di una collezione in genere non è nota a priori a differenza degli ARRAY

Nella libreria di base di JAVA abbiamo un package che offre un vasto insieme di interfacce ed classi per la gestione di collezione di oggetti
→ Il package ha subito una sostanziale modifica in seguito all'introd. dei Generics

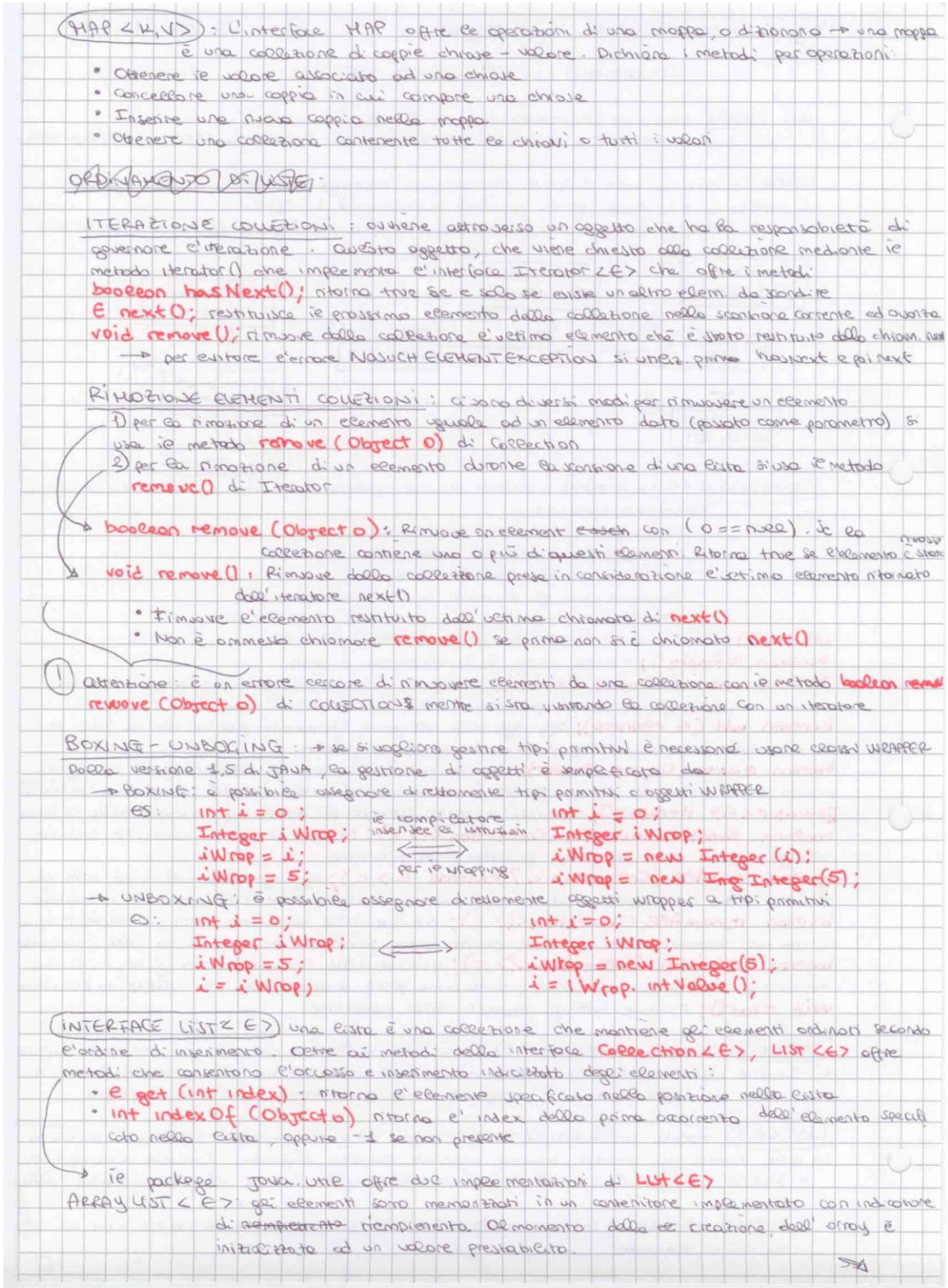


COLLECTION<E>: dichiara i metodi di una generica collezione e permettono di svolgere le seguenti operazioni:
- Aggiungere un elemento alla collezione
- Verificare la dimensione della collezione
- Verificare se la collezione è vuota
- Aggiungere tutti gli elementi di un'altra collezione
- Ottenere un iteratore con cui scorrere la collezione

`int size();` ritorna il n° di elementi presenti nella collezione
`boolean isEmpty();` ritorna true se la collezione è vuota
`boolean contains(Object element);` ritorna true se la collezione contiene un elemento uguale a quello passato come parametro → (usa internamente equals)
`boolean add(E element);` aggiunge alla collezione l'elemento passato; ritorna true se la collezione è cambiata dopo la chiamata a questo metodo
`boolean remove(Object element);` rimuove dalla collezione gli elementi uguali all'oggetto passato come parametro → uguaglianza è verificata dal metodo equals; ritorna true se la collezione è cambiata
`Interator<E> iterator();` restituisce un oggetto Iterator per iterare sugli elementi della collezione
`boolean const containsAll(Collection<?> c);` ritorna true se la collezione contiene tutti gli elementi della collezione passata come parametro
`boolean addAll(Collection<? extends E> c);` aggiunge alla collezione tutti gli elementi della collezione passati come parametro → ritorna true se la collezione è cambiata
`boolean removeAll(Collection<?> c);` rimuove dalla collezione tutti gli elementi uguali a quelli che sono contenuti nella collezione passata come parametro → true se la coll è cambiata
`boolean retainAll(Collection<?> c);` rimuove dalla collezione tutti gli elementi che non sono presenti nella collezione passata come param → ritorna true se la collezione è cambiata
`void clear();` rimuove tutti gli elementi della collezione

SET<E>: estende Collection<E> → è una collezione che non può contenere duplicati. Inoltre offre tutti e soli i metodi della interface Collection con la restrizione che le classi che la implementano si impegnano a non ammettere la presenza di duplicati

LIST<E> estende Collection<E> e corrisponde ad una sequenza → collezione ordinata di elementi
Le liste rispetto agli insiemi, possono contenere elementi duplicati. Inoltre offre operazioni:
- ACCESSO POSIZIONALE: permette di accedere agli elementi in base alla loro posizione nella lista
- RICERCA: permette di ricercare un elemento nella lista e ritorna la sua posizione all'interno della sequenza



MAP<K,V>: L'interfaccia MAP offre le operazioni di una mappa, o dizionario → una mappa è una collezione di coppie chiave - valore. Dichiara i metodi per operazioni:
- Ottenere il valore associato ad una chiave
- Cancellare una coppia in cui compare una chiave
- Inserire una nuova coppia nella mappa
- Ottenere una collezione contenente tutte le chiavi o tutti i valori

ORDINAMENTO DI LISTE:

ITERAZIONE COLLEZIONI: avviene attraverso un oggetto che ha la responsabilità di governare l'iterazione. Questo oggetto, che viene chiesto alla collezione mediante il metodo iterator() che implementa l'interfaccia Iterator<E> che offre i metodi:
boolean hasNext(); ritorna true se e solo se esiste un altro elem. da scandire
E next(); restituisce il prossimo elemento della collezione nella scansione corrente ed avanza
void remove(); rimuove dalla collezione l'ultimo elemento che è stato restituito dalla chiam. next
→ per evitare l'errore NOSUCHELEMENTEXCEPTION si usa prima hasNext e poi next

RIMOZIONE ELEMENTI COLLEZIONI: ci sono diversi modi per rimuovere un elemento
1) per la rimozione di un elemento uguale ad un elemento dato (passato come parametro) si usa il metodo remove(Object o) di Collection
2) per la rimozione di un elemento durante la scansione di una lista si usa il metodo remove() di Iterator

→ boolean remove(Object o): Rimuove un elemento e con (o==null). Se la collezione contiene uno o più di questi elementi. Ritorna true se l'elemento è stato rimosso
→ void remove(): Rimuove dalla collezione presa in considerazione l'ultimo elemento ritornato dall'iteratore next()
- Rimuove l'elemento restituito dall'ultima chiamata di next()
- Non è ammesso chiamare remove() se prima non si è chiamato next()

(!) Attenzione: è un errore cercare di rimuovere elementi da una collezione con il metodo boolean remove(Object o) di COLLECTION mentre si sta visitando la collezione con un iteratore

BOXING - UNBOXING: → se si vogliono gestire tipi primitivi è necessario usare classi WRAPPER
Dalla versione 1.5 di JAVA, la gestione di oggetti è semplificata da:
→ BOXING: è possibile assegnare direttamente tipi primitivi a oggetti Wrapper
es:
```
int i = 0;                          int i = 0;
Integer iWrap;      il compilatore  Integer iWrap;
iWrap = i;          inserisce le    iWrap = new Integer(i);
iWrap = 5;          istruzioni      iWrap = new Integer(5);
                    per il wrapping
```
→ UNBOXING: è possibile assegnare direttamente oggetti wrapper a tipi primitivi
es:
```
int i = 0;                          int i = 0;
Integer iWrap;                      Integer iWrap;
iWrap = 5;                          iWrap = new Integer(5);
i = iWrap;                          i = iWrap.intValue();
```

INTERFACE LIST<E>: una lista è una collezione che mantiene gli elementi ordinati secondo l'ordine di inserimento. Oltre ai metodi della interfaccia Collection<E>, LIST<E> offre metodi che consentono l'accesso e inserimento indicizzato degli elementi:
- E get(int index): ritorna l'elemento specificato nella posizione nella lista
- int indexOf(Object o) ritorna l'index della prima occorrenza dell'elemento specificato nella lista, oppure -1 se non presente

→ le package Java.util offre due implementazioni di LIST<E>
ARRAYLIST<E>: gli elementi sono memorizzati in un contenitore implementato con un array di dimensione. Al momento della creazione dell'array è inizializzato ad un valore prestabilito.

Programmazione Orientata agli Oggetti 24/04/12

COLLEZIONI: LISTE GENERICHE Lezione IX

ARRAYLIST<E> : quando il numero di elementi è prossimo alla capacità dell'ARRAY viene istanziato un nuovo array (di dim doppia) nel quale vengono copiati i vecchi

↳ COSTRUTTORI : ① ArrayList() costruisce una lista vuota con capacità iniziale di 10
② ArrayList(Collection<? extends E> c) : costruisce una lista contenente gli elementi specificati nella collezione ③ ArrayList(int initial capacity)

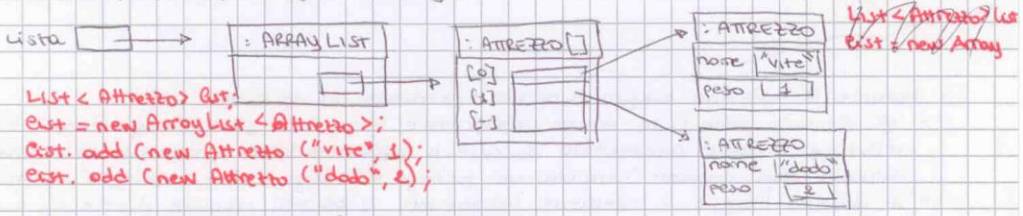
LINKED LIST : Gli elementi sono memorizzati in una lista concatenata. Ogni elemento della lista contiene: un riferimento all'elemento successivo; un riferimento all'oggetto memorizzato. Inoltre non è necessario stabilire una capacità iniziale

↳ COSTRUTTORI : ① LinkedList<E> costr. una lista vuota. ② LinkedList<E>(Collection<? extends E> c) costr. una lista contenente gli elementi nella specifica collezione. → dato dall'ordine dell'iteratore

ITERAZIONE : FOR-EACH : Per iterare su tutti gli elementi di una collezione è possibile usare la forma for-each

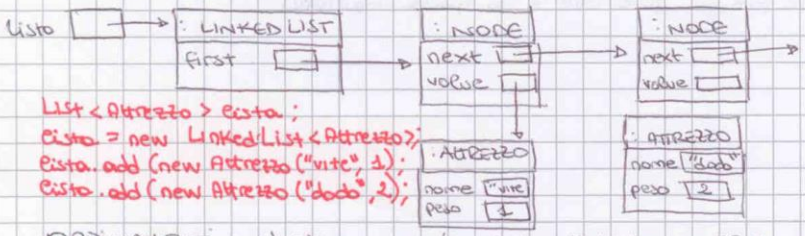
```
for (Tipo elemento : collezione)
    istruzione_su elemento
```

DIAGRAMMA DEGLI OGGETTI : ARRAYLIST




```
List<Attrezzo> lista;
lista = new ArrayList<Attrezzo>();
lista.add(new Attrezzo("vite", 1));
lista.add(new Attrezzo("dado", 2));
```

DIAGRAMMA DEGLI OGGETTI : LINKEDLIST




```
List<Attrezzo> lista;
lista = new LinkedList<Attrezzo>;
lista.add(new Attrezzo("vite", 1));
lista.add(new Attrezzo("dado", 2));
```

ORDINAMENTI : l'ordinamento ha senso se esiste una relazione d'ordine tra gli elementi della lista: In altri termini, gli elementi della lista devono sapersi confrontare

oppure ci deve essere un oggetto esterno che sa come confrontare due oggetti della lista

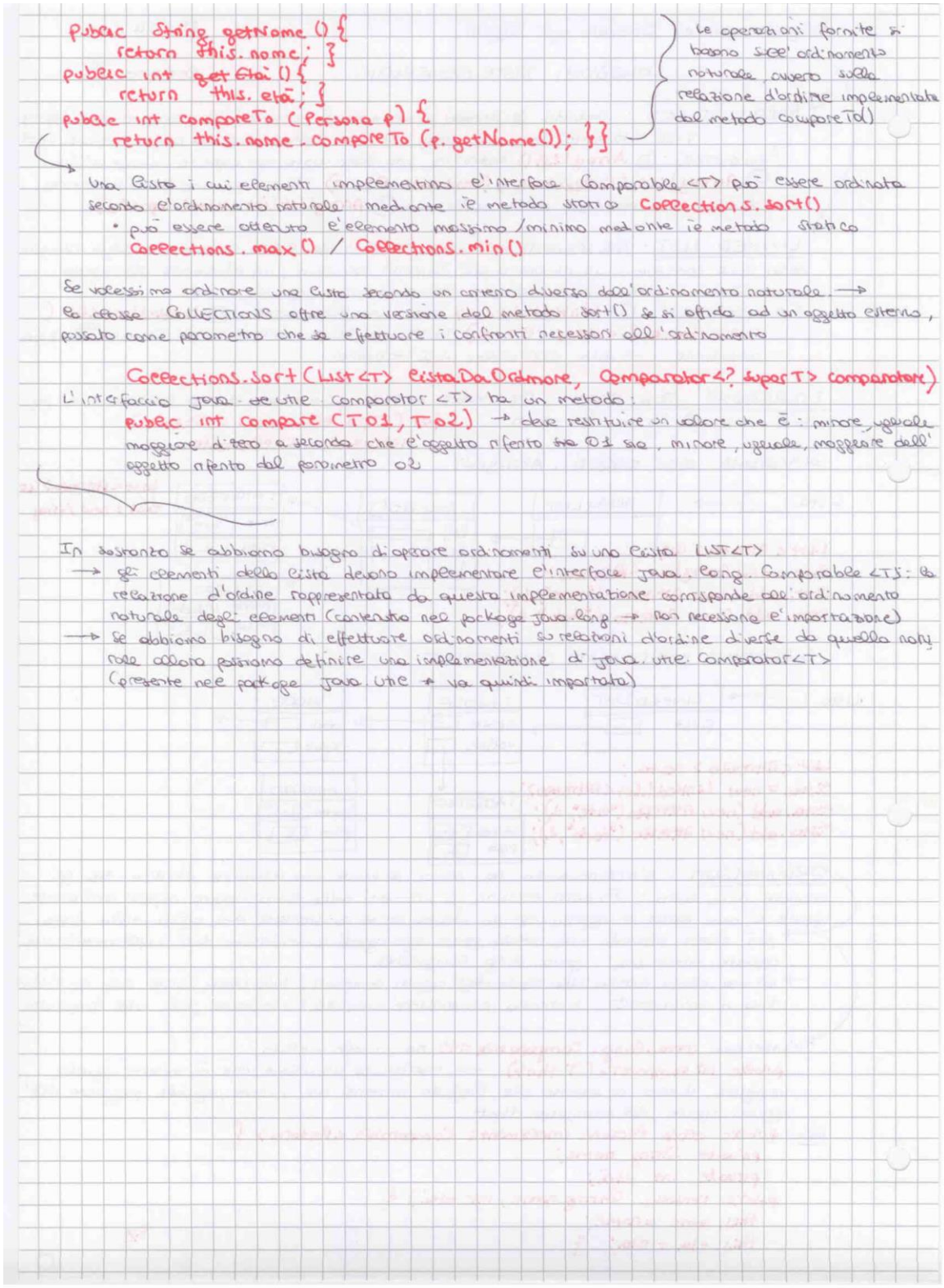
→ può essere affidata alla stessa classe degli oggetti contenuti che deve implementare una apposita interfaccia java.lang.Comparable

→ ad una classe esterna alla classe degli oggetti contenuti; tale classe esiste solo con l'obiettivo di confrontarli; si chiama comparatore e rispetta l'interfaccia java.util.Comparator

e'interface java.lang.Comparable<T> ha un solo metodo

public int compareTo(T that) → restituisce un valore che è minore, uguale, maggiore di zero a seconda che l'oggetto corrente sia minore, uguale, maggiore dell'oggetto riferito dal parametro that

Es
```
public class Persona implements Comparable<Persona> {
    private String nome;
    private int età;
    public Persona(String nome, int età) {
        this.nome = nome;
        this.età = età; }
```



```
public String getNome() {
    return this.nome; }
public int getEtà() {
    return this.età; }
public int compareTo(Persona p) {
    return this.nome.compareTo(p.getNome()); }}
```

Le operazioni fornite si basano sull'ordinamento naturale offerto sulla relazione d'ordine implementata dal metodo compareTo()

Una lista i cui elementi implementano l'interfaccia Comparable<T> può essere ordinata secondo l'ordinamento naturale mediante il metodo statico **Collections.sort()**

- può essere ottenuto l'elemento massimo/minimo mediante il metodo statico **Collections.max() / Collections.min()**

Se volessimo ordinare una lista secondo un criterio diverso dall'ordinamento naturale → la classe COLLECTIONS offre una versione del metodo sort() se si affida ad un oggetto esterno, passato come parametro che sa effettuare i confronti necessari all'ordinamento

Collections.sort(List<T> listaDaOrdinare, Comparator<? super T> comparatore)

L'interfaccia Java generica Comparator<T> ha un metodo:

public int compare(T o1, T o2) → deve restituire un valore che è: minore, uguale maggiore di zero a seconda che l'oggetto riferito da o1 sia, minore, uguale, maggiore dell'oggetto riferito dal parametro o2

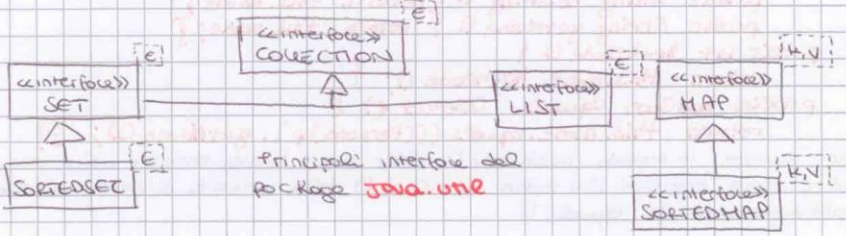
In sostanza se abbiamo bisogno di operare ordinamenti su una lista List<T>

→ gli elementi della lista devono implementare l'interfaccia Java.lang.Comparable<T>: la relazione d'ordine rappresentata da questa implementazione corrisponde all'ordinamento naturale degli elementi (contenuto nel package java.lang → non necessario l'importazione)

→ se abbiamo bisogno di effettuare ordinamenti su relazioni d'ordine diverse da quella naturale allora possiamo definire una implementazione di java.util.Comparator<T> (presente nel package java.util → va quindi importata)

Programmazione orientata agli oggetti 27/04/12

COLLEZIONI: INSIEMI GENERICI Lezione X



SET<E>: un insieme (Set) è una collezione che non contiene duplicati

↳ estende l'interfaccia Collection<E> oltre tutti i suoi metodi della interfaccia COLLECTION<E> con la restrizione che le classi che la implementano si impegnano a non ammettere duplicati

- CRITERI DI EQUIVALENZA: necessari al fine di rilevare l'inserimento di duplicati

→ Nelle librerie JAVA Collection FrameWork sono presenti due implementazioni di SET<E>

① HASHSET<E>: richiede che le classi ridefiniscano opportunamente due metodi (ereditati da Object) che servono a verificare ed evitare la presenza di duplicati

- il metodo equals: `public boolean equals(Object that)`
- il metodo hashCode: `public int hashCode()`

Una funzione di hash è una funzione che calcola un numero intero, codice hash, a partire dai dati di un oggetto, in modo che sia molto probabile che oggetti non uguali abbiano codici diversi

- Due o più oggetti possono avere lo stesso codice hash: situazione che genera una collisione
- Una buona funzione di HASH deve minimizzare le collisioni

! OSS: se i codici hash sono diversi allora le stringhe non sono uguali; il contrario non è necessariamente vero

Se non li definiamo, questi metodi vengono ereditati da java.lang.Object secondo una semantica tutta: hashCode() restituisce l'indirizzo in memoria, equals() equivale ad un semplice ==

→ V classe va definito accuratamente il metodo equals() e deve soddisfare:
Riflessività: per qualunque riferimento x, x.equals(x) deve restituire true
Simmetria: V riferimento x e y, x.equals(y) deve restituire true <=> y.equals(x) restituisca true
Transitività: V riferimento x, y e z => x.equals(y) restituisca true e y.equals(z) restituisca true => x.equals(z) deve restituire true
V riferimento x non nullo, x.equals(null) restituisca false
Se due riferimenti x, y sono identici (x == y) => x.equals(y) deve restituire true

In pratica per verificare la presenza di duplicati HASHSET<E> usa equals() in combinazione con il valore restituito dal metodo hashCode()
In particolare per verificare se 2 oggetti sono identici uguali HashSet: prima verifica se il loro codice hash è identico: solo in caso affermativo (collisione) invoca il metodo equals()
Es: `poco public int hashCode() { return 0; }` → hash code con collisioni

! Ogni qual volta si scrive un metodo equals() è OBBLIGATORIO scrivere anche il corrispondente metodo hashCode()

! Per implementare efficacemente il metodo hashCode() nella produzione del codice hash conviene usare le stesse informazioni usate dal metodo equals()

es:
```
pubeic ceass Persona {
    private String nome;
    Persona (String nome)  { this.nome = nome; }
    pubeic String toString () { retorn. this.nome ; }
    pubeic String getNome () { retorn this.nome; }
pubeic int hashCode () {
    return this.nome.hashcode (); }
pubeic booeean equals (Object p) {
    return this.nome.equals (((Persona)p).getNome ()); }}
```

→ Per scrivere il metodo hashCode() delle nostre classi conviene usare una combinazione dei codici hash restituiti dai metodi hashCode() delle variabili di istanza utilizzate nell'implementazione di equals()

(2) TREESET<E>: quest'implementazione garantisce (oltre all'assenza di duplicati) che gli elementi sono ordinati in accordo a:
- l'ordinamento naturale degli elementi
- un ordinamento stabilito da un comparatore esterno, noto all'insieme stesso

COSTRUTTORI:
TreeSet(): costruisce una nuova albero vuoto ordinato in base all'ordinamento naturale
TreeSet(Collection<? extends E> c): costruisce un nuovo albero contenente gli elementi della lista specificata → ordinamento naturale
TreeSet(Comparator<? super E> comparator) costruisce una nuova albero vuota, ordinata in accordo con lo specifico comparatore

L'implementazione TREESET è basata su alberi di ricerca binaria dove gli elementi dell'albero sono nodi che hanno più riferimenti ad altri nodi
→ In ogni albero binario ogni nodo ha dei figli, nodo destro e nodo sinistro che rispettano:
- i valori dei dati di tutti i discendenti alla sinistra di qualunque nodo sono inferiori al valore del dato memorizzato in quel nodo, mentre tutti i discendenti alla destra contengono valori maggiori
- per stabilire se un nodo deve andare a destra o a sinistra dobbiamo sapere se i suoi dati sono minori o maggiori dei dati del nodo genitore → gli elementi devono implementare Comparable

es:
```
Pubeic ceass Persona implements Comparable<Persona> {
    private String nome;
    private (String nome) { this.nome = nome; }
    pubeic String toString () { return this.nome; }
    pubeic String getNome () { return this.nome; }
    pubeic int hashCode () { return this.nome.hashCode (); }
    pubeic int compareTo (Persona p) { return this.nome.compareTo (p.getNome()); }
    pubeic booeean equals (Object p) {
        return this.nome.equals (((Persona)p).getNome ()); }}
```

es:
```
~~pubei~~
pubeic ceass Persona {
    private String nome;
    Persona (String nome) { this.nome = nome; }
    pubeic String toString () { return this.nome ; }
    pubeic String getNome () { return this.nome ; } }

    pubeic ceass ComparatorePersone implements Comparator<Persona> {
        pubeic int compare (Persona p1, Persona p2) {
            return p1.getNome().compareTo (p2.getNome ()); }}
```

(1) Per verificare la presenza di duplicati → TREESET<E> usa il metodo compareTo() quindi gli oggetti che appartengono alla collezione devono implementare COMPARABLE<E>
Oppure l'oggetto TREESET<E> deve essere creato passando al costruttore un oggetto COMPARATORE

Programmazione orientata agli oggetti 4/05/12

COLLEZIONI : INSIEMI GENERICI lezione X

(!) le implementazioni di compareTo(), hashCode() e equals() devono avere una semantica coerente
- ogni volta che definiamo il metodo equals() dobbiamo definire anche il metodo hashCode()
- se la classe implementa Comparable<E> → definiamo il metodo compareTo() questo è bene che sia coerente con equals()
  → dati due riferimenti ad oggetti x ed y allora
  x.compareTo(y) restituisce 0 <=> x.equals(y) restituisce true

Quale implementazione conviene utilizzare ?

① TREESET<E> : mantiene ordinata la collezione ma è meno efficiente → va usata solo solamente se è l'esigenza di mantenere ordinato la collezione
② HASHSET<E> → molto più efficiente

COLLEZIONI : MAPPE GENERICHE 8/05/12 lezione XI

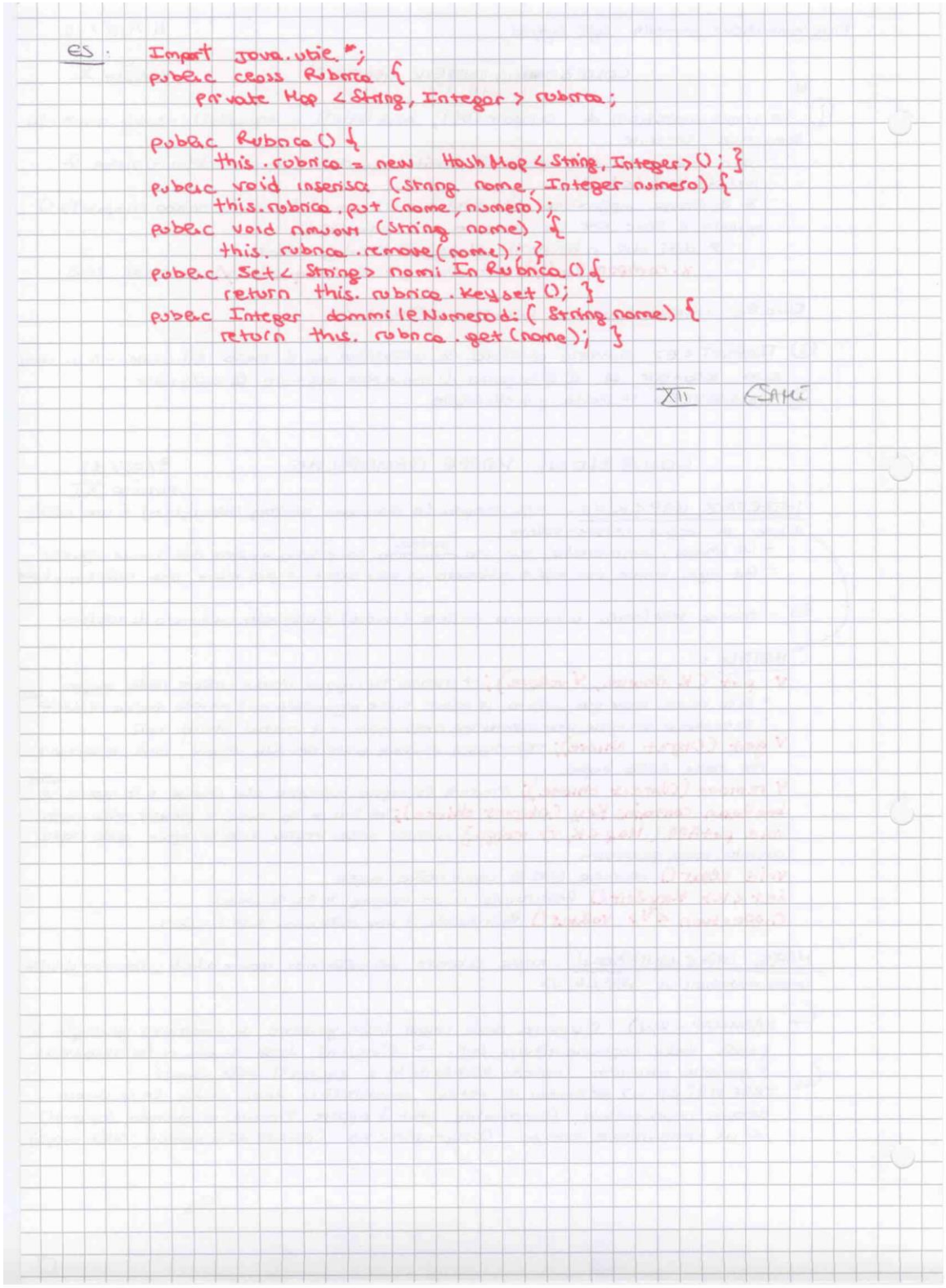
INTERFACE MAP<K,V> : una mappa (o dizionario, o array associativo) è una collezione di coppie chiave-valore
- le chiavi, sono uniche → in una mappa non possono esistere due chiavi uguali
- ad ogni chiave può essere associato un solo valore → può essere una collezione (valori)

Es : rubrica telefonica a ciascuna persona (chiave) è associato un numero di telefono

METODI :
- V put(K chiave, V valore); → inserisce la coppia chiave-valore nella mappa.
  - se la chiave esiste già, allora il valore viene aggiornato e il metodo ritorna il vecchio valore
  - se la chiave non esiste viene inserita una nuova coppia, e il metodo ritorna null
- V get(Object chiave); restituisce il valore associato alla chiave; null se la chiave non esiste nella mappa
- V remove(Object chiave); rimuove la coppia associata alla chiave e ritorna il valore
- boolean containsKey(Object chiave); verifica se la chiave è presente nella mappa
- void putAll(Map<K,V> mappa) inserisce nella mappa tutte le coppie della mappa passata come parametro
- void clear() elimina tutte le coppie della mappa
- Set<K> keySet() restituisce in un insieme tutte le chiavi
- Collection<V> values() restituisce in una collezione tutti i valori

MAPPE IMPLEMENTAZIONI : nelle librerie del collection FrameWork, abbiamo diverse implementazioni di MAP<K,V>

→ HASHMAP<K,V) : l'unicità delle chiavi viene garantita da meccanismi analoghi a quelli delle implementazioni Set. → l'unicità delle chiavi in HashMap<K,V> è garantita attraverso i metodi hashCode() e equals() delle chiavi
→ TREEMAP<K,V> attraverso il metodo compareTo() delle chiavi (e le chiavi devono implementare Comparable<K>) oppure tramite il metodo compare() di un comparatore esterno Comparator<K> (passato al costruttore della mappa)



es.

```
Import java.util.*;
public class Rubrica {
    private Map <String, Integer> rubrica;

public Rubrica() {
        this.rubrica = new HashMap <String, Integer>(); }
public void inserisci (String nome, Integer numero) {
        this.rubrica.put (nome, numero);
public void rimuovi (String nome) {
        this.rubrica.remove(nome); }
public Set< String> nomiInRubrica(){
        return this.rubrica.keySet(); }
public Integer dammiIlNumeroDi( String nome) {
        return this.rubrica.get(nome); }
```

XII ESAMI

NOME: _____ MATRICOLA: _____

COGNOME: _____

Dobbiamo ristrutturare una applicazione per la gestione di un magazzino. La classe **Magazzino** (vedi codice Java) modella un magazzino: nella implementazione corrente è previsto che in un magazzino possano essere due tipologie di prodotti: libri e dischi. Le classi **Libro** e **Disco**, modellano le tipologie di prodotti libro e disco, rispettivamente: ogni tipologia di libro (disco) ha un codice identificativo, ed un costo unitario (per semplicità trascuriamo altri dettagli specifici). Le tipologia di prodotti presenti in magazzino sono memorizzati in due mappe (una per i libri e una per i dischi) che associano ad ogni libro (disco) presente in magazzino la quantità. Ad esempio supponendo che nel magazzino siano presenti 50 libri con codice "*AAA*", 30 libri con codice "*BBB*", 8 dischi con codice "*ZZZ*" le mappe **libro2quantita** e **disco2quantita** memorizzerebbero le seguenti informazioni (per chiarezza indichiamo i codici del libro e del disco al posto dei riferimenti agli oggetti **Libro** e **Disco**):

libro2quantita	disco2quantita
AAA → 50	ZZZ → 8
BBB → 30	

Domanda 1 (peso 35%)
Ci si aspetta (come che nel magazzino possono essere molte tipologie di prodotti oltre a libri e dischi (ad esempio, giochi di società, DVD, etc.) sarebbe molto svantaggioso introdurre una mappa per ciascuna tipologia. Un programmatore esperto suggerisce di agire come segue: le classi **Libro**, **Disco** (e tutte quelle che serviranno a modellare nuove tipologie di prodotti) devono implementare l'interface **Articolo**, che offre i metodi **int getCostoUnitario()** e **String getCodice()**. Nella classe magazzino sarà quindi presente una sola mappa **Map<Articolo,Integer> articolo2quantita**

- scrivere il codice della interface **Articolo**
- mostrare come devono essere modificate le classi **Libro** e **Disco**, affinché implementino l'interface **Articolo**
- riscrivere tutto il codice della classe **Magazzino** sostituendo le due mappe con la mappa **Map<Articolo,Integer> articolo2quantita**. In particolare va riscritto il costruttore, e i metodi **aggiungiLibro()** e **aggiungiDisco()** vanno eliminati e rimpiazzati da un unico metodo **aggiungiArticolo()**

Domanda 2 (peso 15%)
Scrivere il codice del metodo **int calcolaValoreMagazzino()** della classe **Magazzino** (riscritta secondo le indicazioni della Domanda 1) che calcola il valore complessivo degli articoli presenti in magazzino. In particolare il valore complessivo degli articoli presenti nel magazzino è calcolato sommando il costo complessivo di ogni articolo presente in magazzino. Il costo complessivo di un articolo è calcolato come il costo unitario dell'articolo moltiplicato per la quantità presente in magazzino. Ad esempio supponendo che nel magazzino siano presenti 50 libri con codice "*AAA*" (costo unitario 10) e 8 dischi con codice "*ZZZ*" (costo unitario 5), il costo complessivo del magazzino sarebbe 10*50 + 5*8 = 540

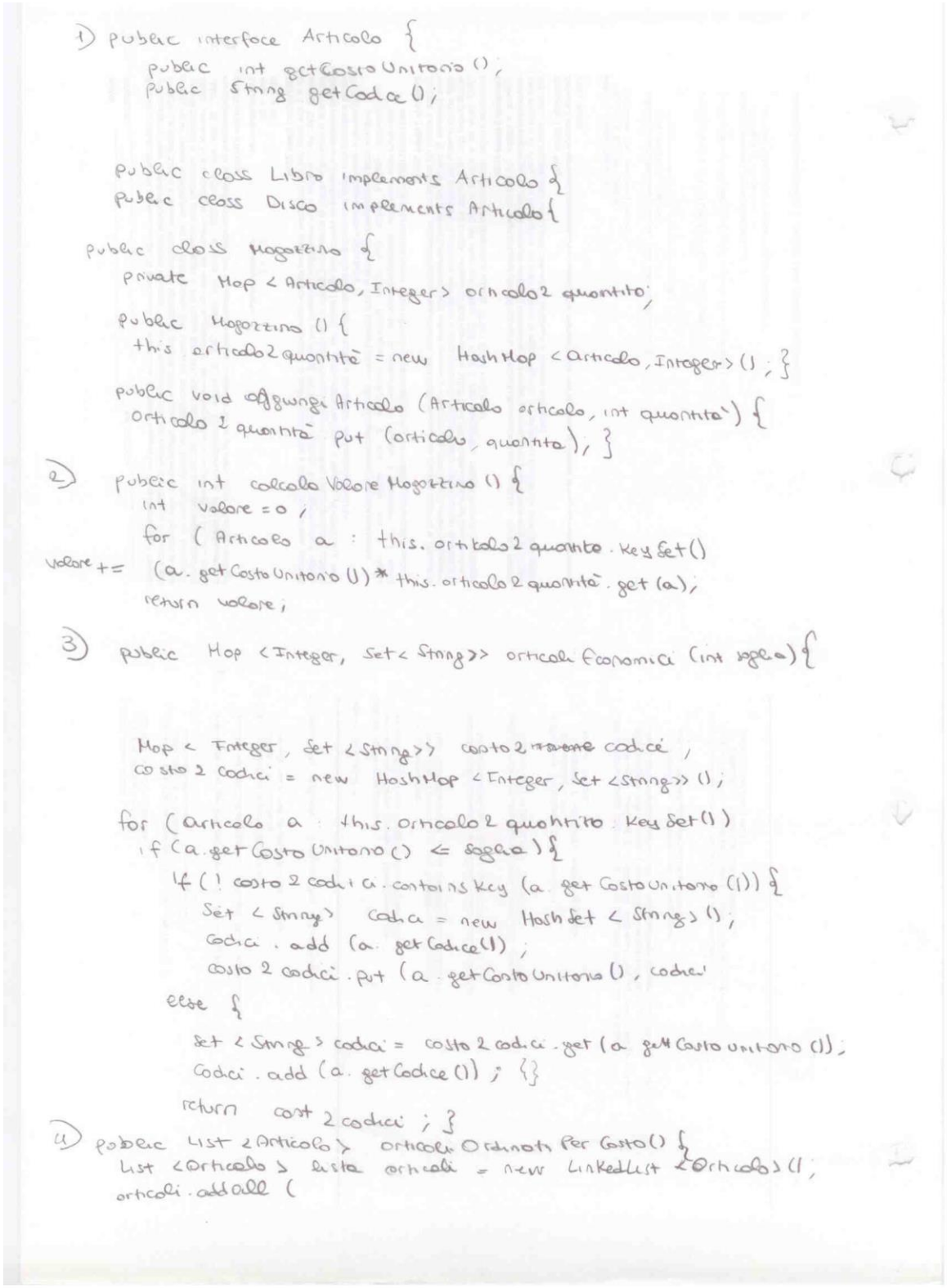
Domanda 3 (peso 35%)
Scrivere il codice del metodo **Map<Integer, Set<String>> articoliEconomici(int soglia)**: questo metodo restituisce in una mappa gli articoli il cui costo unitario è inferiore ad una soglia (passata come parametro al metodo). Chiave della mappa è un intero, che rappresenta un costo unitario; la mappa associa alla chiave un insieme dei codici degli articoli che hanno costo unitario pari a quello della chiave (che comunque è inferiore alla soglia)

Domanda 4 (peso 15%)
Scrivere il metodo **articoliOrdinatiPerCosto()** che restituisce una lista degli articoli presenti in magazzino ordinata per costo unitario

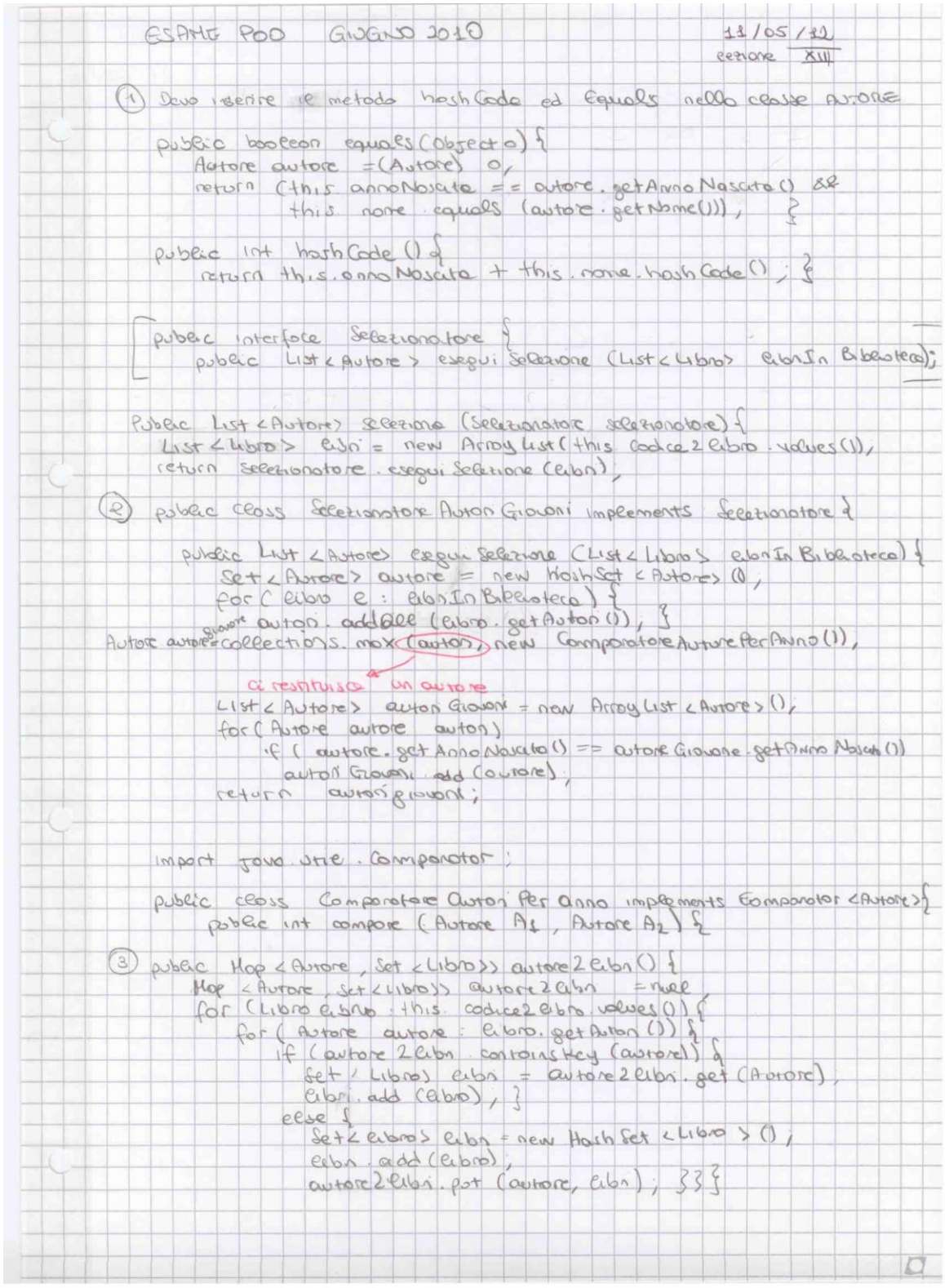
```
public class Disco {
    private String codice;
    private int costoUnitario;

    public Disco(String codice, int costoUnitario) {
        this.codice = codice;
        this.costoUnitario = costoUnitario;
    }

    public String getCodice(){ return this.codice; }

    public int getCostoUnitario(){ return this.costoUnitario; }

    public boolean equals(Object o) {
        Disco d = (Disco)o;
        return this.codice.equals(d.getCodice());
    }

    public int hashCode() { return this.codice.hashCode(); }
}
```

```
public class Libro {
    private String codice;
    private int costoUnitario;

    public Libro(String codice, int costoUnitario) {
        this.codice = codice;
        this.costoUnitario = costoUnitario;
    }

    public String getCodice(){ return this.codice; }

    public int getCostoUnitario(){ return this.costoUnitario; }

    public boolean equals(Object o) {
        Libro l = (Libro)o;
        return this.codice.equals(l.getCodice());
    }

    public int hashCode() { return this.codice.hashCode(); }
}
```

```
import java.util.*;

public class Magazzino {
    private Map<Libro, Integer> libro2quantita;
    private Map<Disco, Integer> disco2quantita;

    public Magazzino(){
        this.libro2quantita = new HashMap<Libro,Integer>();
        this.disco2quantita = new HashMap<Disco,Integer>();
    }

    public void aggiungiDisco(Disco disco, int quantita){
        disco2quantita.put(disco, quantita);
    }

    public void aggiungiLibro(Libro libro, int quantita){
        libro2quantita.put(libro, quantita);
    }

    public int calcolaValoreMagazzino() {
        int valore = 0;
        // DOMANDA 2: scrivere codice mancante
        return valore;
    }

    public Map<Integer,Set<String>> articoliEconomici(int soglia){
        // costo -> insieme di codici di articoli con il costo della chiave
        Map<Integer,Set<String>> costo2insiemeCodiciArticoli;
        costo2insiemeCodiciArticoli = new HashMap<Integer,Set<String>>();
        // DOMANDA 3: scrivere codice mancante
        return costo2insiemeCodiciArticoli;
    }

    public List<Articolo> articoliOrdinatiPerCosto() {
        List<Articolo> listaArticoli = new LinkedList<Articolo>();
        // DOMANDA 4: scrivere codice mancante
        return listaArticoli;
    }
}
```



```
1) public interface Articolo {
     public int getCostoUnitario();
     public String getCodice();

   public class Libro implements Articolo {}
   public class Disco implements Articolo {}

   public class Magazzino {
     private Map<Articolo, Integer> articolo2quantita;

     public Magazzino() {
       this.articolo2quantita = new HashMap<Articolo, Integer>(); }

     public void aggiungiArticolo(Articolo articolo, int quantita) {
       articolo2quantita.put(articolo, quantita); }

2)   public int calcolaValoreMagazzino() {
       int valore = 0;
       for (Articolo a : this.articolo2quantita.keySet())
         valore += (a.getCostoUnitario()) * this.articolo2quantita.get(a);
       return valore;

3)   public Map<Integer, Set<String>> articoliEconomici(int soglia) {

       Map<Integer, Set<String>> costo2codici;
       costo2codici = new HashMap<Integer, Set<String>>();

       for (Articolo a : this.articolo2quantita.keySet())
         if (a.getCostoUnitario() <= soglia) {
           if (!costo2codici.containsKey(a.getCostoUnitario())) {
             Set<String> codici = new HashSet<String>();
             codici.add(a.getCodice());
             costo2codici.put(a.getCostoUnitario(), codici);
           else {
             Set<String> codici = costo2codici.get(a.getCostoUnitario());
             codici.add(a.getCodice()); }}
         return costo2codici; }
4)   public List<Articolo> articoliOrdinatiPerCosto() {
       List<Articolo> listaArticoli = new LinkedList<Articolo>();
       articoli.addAll(
```



ESAME POO GIUGNO 2010

11/05/12
Lezione XIII

① Devo riscrivere il metodo hashCode ed Equals nella classe AUTORE

```
public boolean equals(Object o) {
    Autore autore = (Autore) o;
    return (this.annoNascita == autore.getAnnoNascita() &&
            this.nome.equals(autore.getNome()));
}

public int hashCode() {
    return this.annoNascita + this.nome.hashCode(); }
```

```
public interface SelezionatoreAutori {
    public List<Autore> eseguiSelezione(List<Libro> libriInBiblioteca);
}

Public List<Autore> selezione(SelezionatoreAutori selezionatore) {
    List<Libro> libri = new ArrayList(this.codice2Libro.values());
    return selezionatore.eseguiSelezione(libri);
```

② public class SelezionatoreAutoriGiovani implements SelezionatoreAutori {

```
    public List<Autore> eseguiSelezione(List<Libro> libriInBiblioteca) {
        Set<Autore> autori = new HashSet<Autore>();
        for(Libro l : libriInBiblioteca) {
            autori.addAll(l.getAutori()); }
Autore autore = Collections.max(autori, new ComparatoreAutoriPerAnno());
```

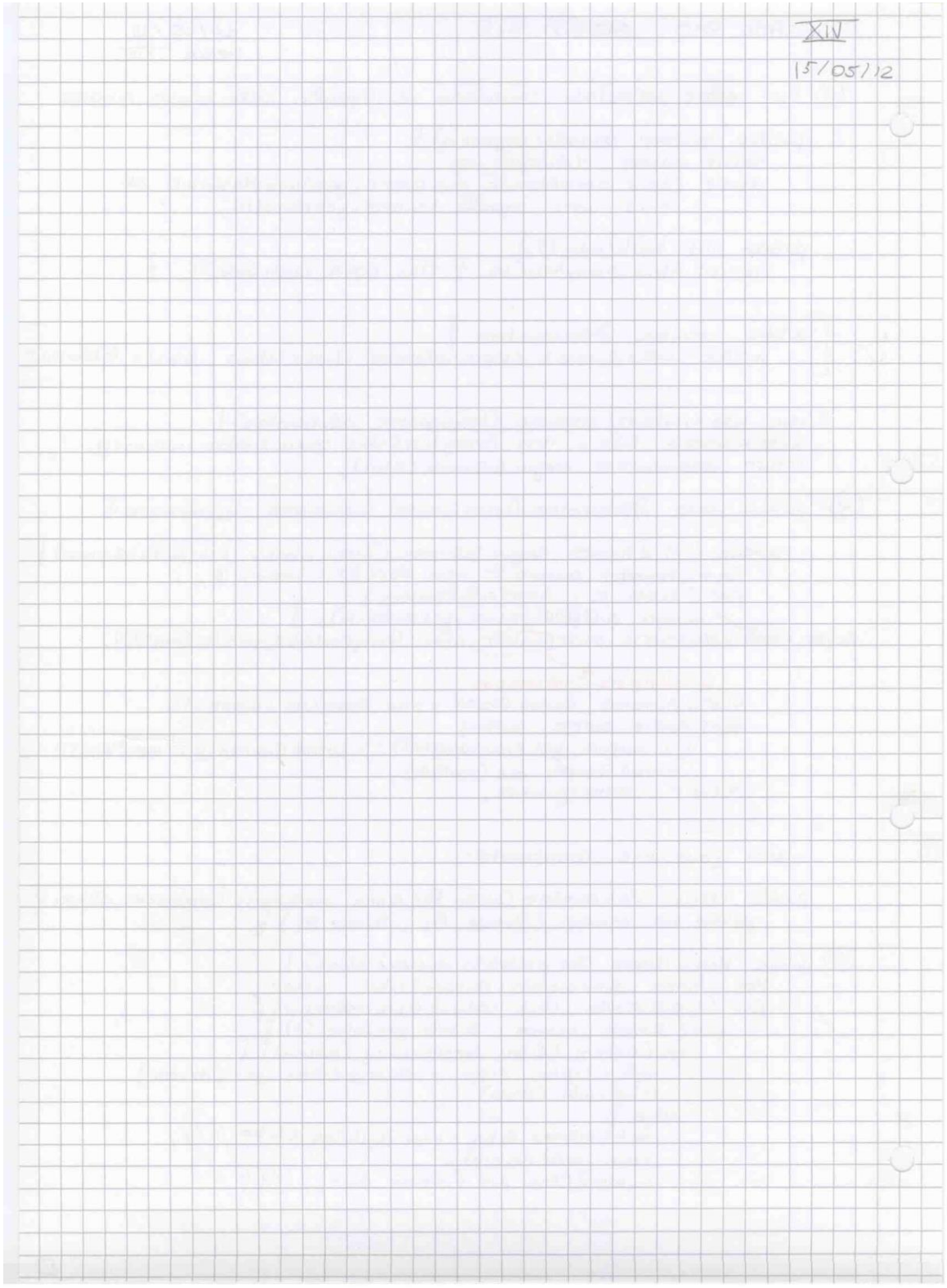
(autori: Set<Autore>) — ci restituisce un autore

```
        List<Autore> autoriGiovani = new ArrayList<Autore>();
        for(Autore autore : autori)
            if (autore.getAnnoNascita() == autoreGiovane.getAnnoNascita())
                autoriGiovani.add(autore);
        return autoriGiovani;
```

```
import java.util.Comparator;

public class ComparatoreAutoriPerAnno implements Comparator<Autore>{
    public int compare(Autore A1, Autore A2) {
```

③ public Map<Autore, Set<Libro>> autore2Libri() {

```
    Map<Autore, Set<Libro>> autore2Libri = null;
    for(Libro libro : this.codice2Libro.values()) {
        for(Autore autore : libro.getAutori()) {
            if (autore2Libri.containsKey(autore)) {
                Set<Libro> libri = autore2Libri.get(Autore);
                libri.add(libro); }
            else {
                Set<Libro> libri = new HashSet<Libro>();
                libri.add(libro);
                autore2Libri.put(autore, libri); }}}
```


XIV
15/05/12

Programmazione Orientata agli Oggetti

15/05/12

Lezione XV

CLASSI ASTRATTE

- L'estensione può essere uno strumento utile per il riuso del codice → le classi estese ereditano infatti anche l'implementazione della classe base
In alcuni casi può essere utile definire classi base che sono pensate solo per essere estese → queste classi contengono una implementazione parziale

CLASSE ASTRATTA:

1) Contiene una definizione parziale della implementazione
2) Non può essere istanziata, ma possono essere istanziate le classi che la estendono
3) Vale la relazione sotto-tipo - super-tipo e quindi il principio di sostituzione: le istanze delle classi che estendono una classe astratta possono essere usate ogni qual volta abbiamo un riferimento alla classe base.

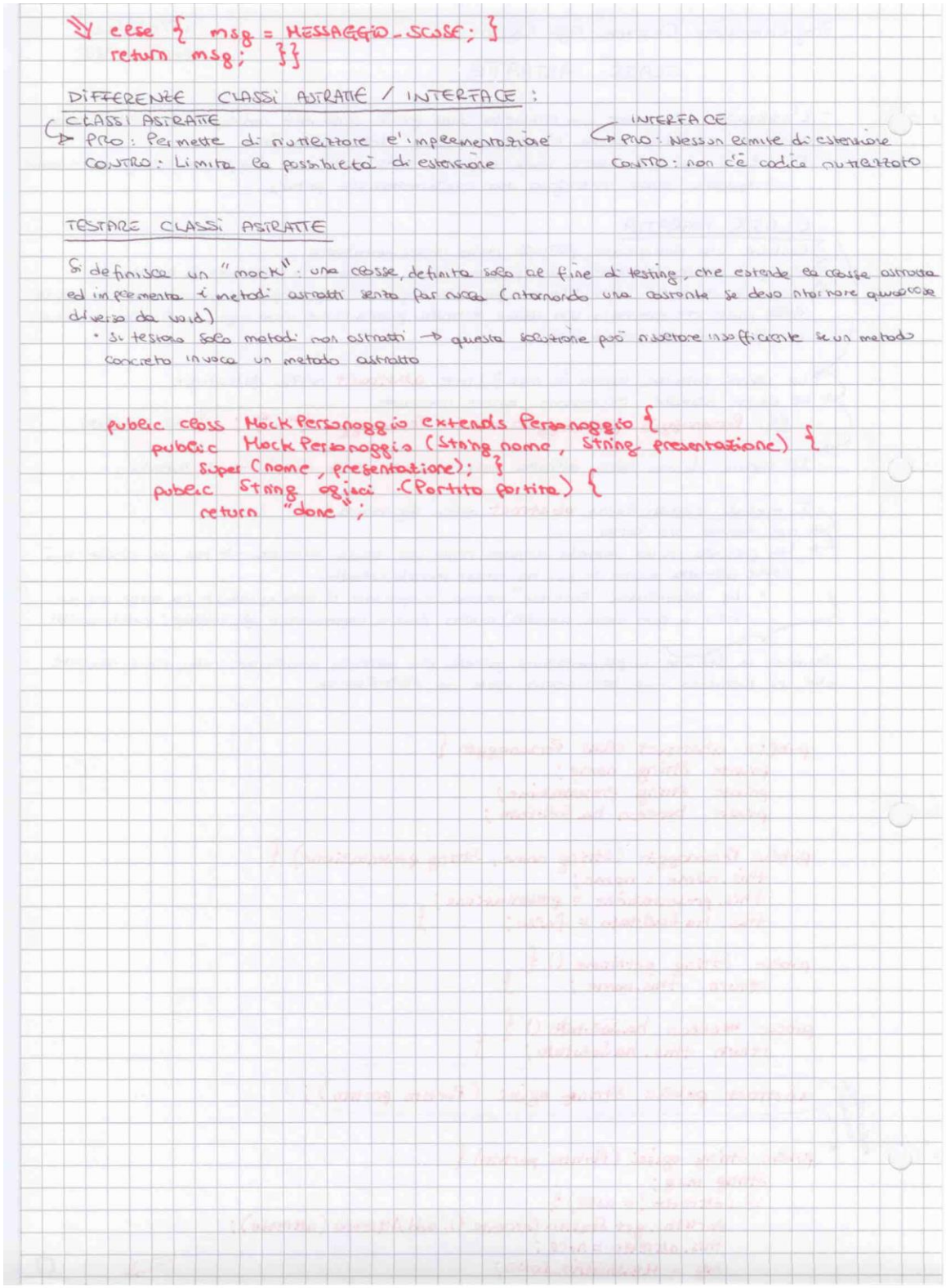
- Le classi astratte hanno il modificatore **abstract** nella definizione
- Le classi astratte non possono essere istanziate
  es: `Personaggio personaggio = new Personaggio ("_");` // non compilato
- Il codice di una classe astratta viene ereditato dalle classi che la estendono

- I metodi astratti hanno **abstract** nella segnatura
  - non hanno un corpo
  - La presenza di un metodo astratto rende una classe astratta → ma una classe può essere astratta anche se non ha nessun metodo astratto
    - Le sottoclassi "concrete" devono completare l'implementazione (a meno che non siano a loro volta astratte) ovvero devono implementare gli eventuali metodi astratti

Servono a definire implementazioni parziali che verranno completate nelle classi concrete che la estendono → sono molto usate nei FRAMEWORK

```
public abstract class Personaggio {
    private String nome;
    private String presentazione;
    private boolean haSalutato;

public Personaggio (String nome, String presentazione) {
    this.nome = nome;
    this.presentazione = presentazione;
    this.haSalutato = falso;        }

public String getNome () {
    return this.nome;        }

public boolean haSalutato () {
    return this.haSalutato;        }

abstract public String agisci (Partita partita);

public String agisci (Partita partita) {
    String msg;
    if (attrezzo != null) {
        partita.getStanzaCorrente().addAttrezzo (attrezzo);
        this.attrezzo = null;
        msg = MESSAGGIO_DONO;
```

nella classe concreta



else { msg = MESSAGGIO_SCUSE; }
return msg; }}

DIFFERENZE CLASSI ASTRATTE / INTERFACE :

CLASSI ASTRATTE
PRO: Permette di riutilizzare e implementazione
CONTRO: Limita la possibilità di estensione

INTERFACE
PRO: Nessun limite di estensione
CONTRO: non c'è codice riutilizzato

TESTARE CLASSI ASTRATTE

Si definisce un "mock": una classe, definita solo al fine di testing, che estende la classe astratta ed implementa i metodi astratti senza far nulla (ritornando una costante se devo ritornare qualcosa di diverso da void)

- Si testano solo metodi non astratti → questa soluzione può risultare inefficiente se un metodo concreto invoca un metodo astratto

```
public class MockPersonaggio extends Personaggio {
    public MockPersonaggio (String nome, String presentazione) {
        super(nome, presentazione); }
    public String agisci (Partita partita) {
        return "done";
```

FEBBRAIO 2008
Lezione XVI

NOME: HOLSI
COGNOME: HASANAJ
MATRICOLA: _____

Vogliamo ristrutturare una libreria per la gestione di oggetti grafici. La classe `Disegno` (vedi codice) modella un disegno. Un disegno è formato da una sovrapposizione di forme elementari: ad ogni forma è associato un livello (o layer) che rappresenta l'ordine con cui devono essere sovrapposte le varie forme. Nella implementazione attuale è previsto che un disegno possa essere composto da due tipologie di forme elementari: cerchi e rettangoli, modellati dalle classi `Cerchio` e `Rettangolo`.
Ogni forma ha un colore (istanza della classe `Colore`) ed un insieme di attributi specifici relativi alla forma (vedi codice). Nella classe `Disegno` le forme che compongono un disegno sono memorizzate in due mappe, una per i rettangoli e una per i cerchi, che associano ad ogni livello una lista di rettangoli e una lista di cerchi, rispettivamente.

Ad esempio supponendo che un disegno sia formato da un rettangolo *R1* e un cerchio *C2* entrambi al livello *2*, sovrapposti da un cerchio *C1* al livello *4*, e da due rettangoli *R2* ed *R3* al livello *5*, le mappe `livello2rettangoli` e `livello2cerchi` memorizzerebbero le seguenti informazioni (per chiarezza indichiamo i nomi dei rettangoli e del cerchi al posto dei riferimenti agli oggetti `Rettangolo` e `Cerchio`):

`livello2rettangolo`	`livello2cerchio`
2 → *{R1}*	*2* → *{C2}*
5 → *{R2, R3}*	*4* → *{C1}*

Domanda 1 (peso 5%)
Scrivere il codice del metodo `aggiungiRettangolo(Rettangolo rettangolo, int livello)` della classe `Disegno`.

Domanda 2 (peso 35%)
Ci si è resi conto che in un disegno potrebbero esserci altre forme oltre a rettangoli e cerchi (ad esempio, ellissi, triangoli, etc.): sarebbe molto svantaggioso introdurre una mappa per ciascuna forma. Un programmatore esperto suggerisce di agire come segue: le classi `Rettangolo`, `Cerchio` (e tutte quelle che serviranno a modellare nuove forme) dovranno estendere una classe astratta `Forma`. Nella classe `Disegno` sarà quindi presente una sola mappa, `Map<Integer, List<Forma>> livello2forme`.

- scrivere il codice della classe astratta `Forma`, che astrae il concetto di forma geometrica
- riscrivere completamente il codice delle classi `Rettangolo` e `Cerchio`, affinché estendano la classe astratta `Forma`
- riscrivere completamente il codice della classe `Disegno` sostituendo le due mappe con la mappa `Map<Integer, List<Forma>> livello2forme`. In particolare va riscritto il costruttore, ed i metodi `aggiungiRettangolo(Rettangolo rettangolo, int livello)` e `aggiungiCerchio(Cerchio cerchio, int livello)` vanno eliminati e rimpiazzati da un unico metodo `aggiungiForma(Forma forma, int livello)`

Domanda 3 (peso 15%)
Scrivere il codice del metodo `Set<Colore> coloriPresentiNelDisegno()` della classe `Disegno` (riscritta secondo le indicazioni della Domanda 2) che restituisce in un insieme i colori di tutte le forme che compongono un disegno. Se necessario è possibile modificare il codice della classe `Colore`; non è possibile introdurre nuove classi.

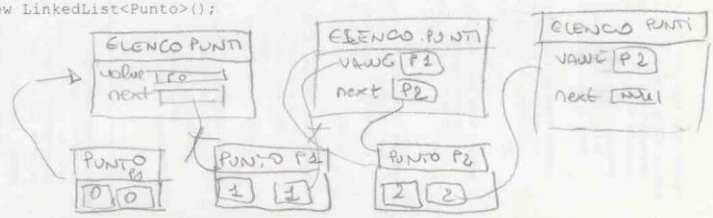
Domanda 4 (peso 10%)
Scrivere il codice del metodo `Map<Colore, List<Forma>> colore2forme()` che restituisce una mappa che associa ad un colore la lista, ordinata in ordine crescente di superficie, delle forme del disegno con tale colore. Non è possibile introdurre altre classi; se necessario è possibile modificare il codice della classe astratta `Forma`.

Domanda 5 (peso 10%)
Scrivere il codice del metodo `List <Forma> formeOrdinatePerLuminosita()` che restituisce in una lista tutte le forme presenti nel disegno. La lista deve essere ordinata in ordine crescente di luminosità del colore. Non è possibile modificare le classi `Forma` e `Colore`, mentre è possibile, se necessario, introdurre nuove classi.

Domanda 6 (peso 15%)
Scrivere una classe di test per la classe `Disegno`. In particolare, scrivere almeno 4 test case per testare la correttezza del metodo `colore2forme()` ed almeno 4 per `formeOrdinatePerLuminosita()`. In particolare la correttezza dei due metodi va verificata nel caso di: *(i)* disegno vuoto, *(ii)* disegno composto da una forma singola, *(iii)* disegno composto da due forme di colore diverso, *(iv)* disegno di due forme dello stesso colore.

Domanda 7 (peso 10%)
Data la seguente porzione di codice, disegnare un diagramma che rappresenti lo stato degli oggetti referenziati dalle variabili `elencoPunti`, `p0`, `p1`, `p2` nel punto marcato con il commento `// ****`

```
public static void main(String[] args) {
    List<Punto> elencoPunti = new LinkedList<Punto>();
    Punto p0 = new Punto(0, 0);
    Punto p1 = new Punto(1, 1);
    Punto p2 = new Punto(2, 2);
    elencoPunti.add(p0);
    elencoPunti.add(p1);
    elencoPunti.add(p2);
    p1 = p2;
    p2 = new Punto(4, 4);
    elencoPunti.add(p1);
    elencoPunti.add(p2);
    // ****
}
```

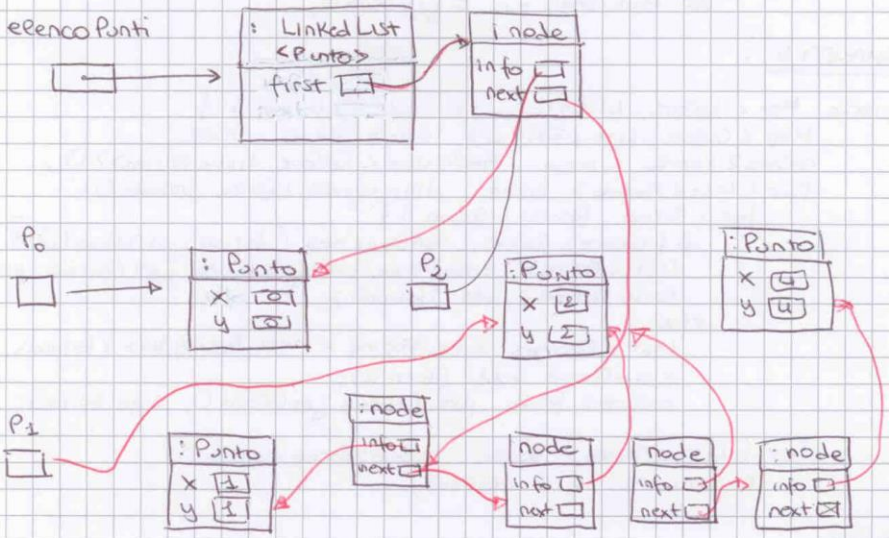


```
import java.util.*;
public class Disegno {
    private Map<Integer, List<Rettangolo>> livello2rettangoli;
    private Map<Integer, List<Cerchio>> livello2cerchi;
    public Disegno() {
        this.livello2rettangoli = new TreeMap<Integer, List<Rettangolo>>();
        this.livello2cerchi = new TreeMap<Integer, List<Cerchio>>();
    }
    public void aggiungiRettangolo(Rettangolo rettangolo, int livello){
        //DOMANDA 1: scrivere il codice di questo metodo
        //DOMANDA 2: eliminare questo metodo e sostituirlo con il metodo
        //          public void aggiungiForma(Forma forma, int livello)
    }
    public void aggiungiCerchio(Cerchio cerchio, int livello){
        // codice omesso
        //DOMANDA 2: eliminare questo metodo e sostituirlo con il metodo
        //          public void aggiungiForma(Forma forma, int livello)
    }
    public Set<Colore> coloriPresentiNelDisegno() {
        Set<Colore> coloriNelDisegno = null;
        //DOMANDA 3: scrivere il codice di questo metodo
        //          (è possibile modificare Colore; non e' possibile introdurre nuove classi)
        return coloriNelDisegno;
    }
    public Map<Colore, List<Forma>> colore2forme() {
        Map<Colore, List<Forma>> colore2forme = null;
        //DOMANDA 4: scrivere il codice di questo metodo
        //          (non è possibile introdurre altre classi; è possibile modificare Forma)
        return colore2forme;
    }
    public List<Forma> formeOrdinateParLuminosita() {
        List<Forma> forme = null;
        //DOMANDA 5: scrivere il codice di questo metodo
        //          (non è possibile modificare Forma e Colore, e possibile introdurre nuove classi)
        return forme;
    }
}

public class Rettangolo {
    private Colore colore;
    private Punto vertice;
    private int larghezza;
    private int altezza;
    public Rettangolo(Colore colore, Punto vertice, int altezza, int larghezza){
        this.colore = colore;
        this.vertice = vertice;
        this.larghezza = larghezza;
        this.altezza = altezza;
    }
    public Colore getColore(){
        return this.colore;
    }
    public Punto getVertice(){
        return this.vertice;
    }
    public int getAltezza(){
        return this.altezza;
    }
    public int getLarghezza(){
        return this.larghezza;
    }
    public double superficie(){
        return (this.altezza * this.larghezza);
    }
}

public class Cerchio {
    private Colore colore;
    private Punto centro;
    private int raggio;
    public Cerchio(Colore colore, Punto centro, int raggio){
        this.colore = colore;
        this.centro = centro;
        this.raggio = raggio;
    }
    public Colore getColore(){
        return this.colore;
    }
    public Punto getCentro(){
        return this.centro;
    }
    public int getRaggio(){
        return this.raggio;
    }
    public double superficie(){
        return Math.PI * this.raggio * this.raggio;
    }
}

public class Punto {
    private int x;
    private int y;
    public Punto(int x, int y){
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
}

public class Colore {
    private int red;
    private int green;
    private int blue;
    public Colore(int red, int green, int blue) {
        this.red = red;
        this.blue = blue;
        this.green = green;
    }
    public int getRed() {
        return this.red;
    }
    public int getGreen() {
        return this.green;
    }
    public int getBlue() {
        return this.blue;
    }
    public int luminosita() {
        return (this.red + this.green + this.blue) /
```

Programmazione Orientata agli Oggetti — ESAME FEBBRAIO 2008

Sezioni XVI

DOMANDA 4 :



DOMANDA 1

```
public void aggiungiRettangolo (Rettangolo rettangolo, int livello){
    List<Rettangolo> listaRettangoli = this.livelli2rettangoli.get(livello);
    if(listaRettangoli == null){
      listaRettangoli = new ArrayList<Rettangolo>();
      this.livelli2rettangoli.put(livello, listaRettangoli);}
      listaRettangoli.add(rettangolo);
```

DOMANDA 2

implements Comparable<Forma>

```
public abstract class Forma {
    private Colore colore;
    public Forma (Colore colore){
        this.colore = colore;}
    abstract public double superficie();
    public Colore getColore(){
        return this.colore;}
```

public int compareTo(Forma o) {

visto che ho messo HashSet devo mettere nella classe colore hashCode e Equals

DOMANDA 3

```
public Set<Colore> coloriPresentiNelDisegno(){
    Set<Colore> coloriNelDisegno = null;
    coloriNelDisegno = new HashSet<Colore>();
    for(List<Forma> forme : this.livelli2forme.values())
        for(Forma forma : forme)
            coloriNelDisegno.add(forma.getColore());
    return coloriNelDisegno;
```

```
public int hashCode(){
    return this.red + this.green + this.blue; }

public boolean equals(Object o) {
    Colore c = (colore) o;
    return this.red == c.getRed() && this.green == c.getGreen()
        && this.blue == c.getBlue();
```

DOMANDA 4 :

```
public Map<Colore, List<Forma>> colore2Forme() {
    Map<Colore, List<Forma>> colore2forma = null;
    colore2forma = new HashMap<Colore, List<Forma>>();
    for (List<Forma> forme : this.elenco2forme.values())
        for (Forma forma : forme) {
            if (colore2forme.containsKey(forma.getColore())) {
                List<Forma> listaForme = colore2forme.get(forma.getColore());
                listaForme.add(forma); }
            else
                List<Forma> listaForme = new ArrayList<Forma>();
                listaForme.add(forma);
                colore2forme.put(forma.getColore(), listaForme); } }

    for (List<Forma> forme : colore2forme.values())
        Collections.sort(forme);
```

poi

```
* public int compareTo(Forma f) {
    Double s1 = new Double(this.superficie());
    Double s2 = new Double(f.superficie());
    return s1.compareTo(s2); }
```

Programmazione Orientata agli Oggetti 25/05/12

Lezione XVII

ESERCIZIO 2 : ESAME 2011 GIUGNO

Source → indenta / format

(e meglio così perché se dovessi cambiare la struttura dati non cambierebbe nulla)

```
public Set<Posizione> posizioniAnimaliVecchi() {
    Animale vecchio = Collections.max(this.getAnimali(), new ComparatoreAnimaleEta());
if(vecchio!=null) int etaMax = vecchio.getAnni();
    Set<Posizione> posizioni = new HashSet<Posizione>();
    for (Animale a : this.getAnimali()) {
        if (etaMax == a.getAnni())
            posizioni.add(a.getPosizione()); }
    return posizioni; }

public class ComparatoreAnimaliPerEta implements Comparator<Animale>

public int compare(Animale a1, Animale a2) {
    return a1.getAnni() - a2.getAnni();
```

ESERCIZIO 3b

```
public Map<Integer, Set<Animale>> anno2Animali() {
    Map<Integer, Set<Animale>> anno2animali = new HashMap
    = new HashMap<Integer, Set<Animale>>();
    return anno2animali;
```

(!) attenzione a questo test: devo stare attento perché la popola territorio lo genera indipendentemente (casualmente) quindi non lo posso impostare, per questo

ESERCIZIO 3 :

Nella classe Animale ci deve essere Hash code ed Equals

```
public int HashCode() {
    return this.getPosizione().hashCode(); }

public boolean equals(Object o) {
    Animale a = (Animale) o;
    return a.getPosizione().equals(this.getPosizione());
scritto meglio
    return this.getPosizione().equals(a.getPosizione());
```

ESERCIZIO 4

```
public void setUp() throws Exception {

public class TerritorioTest {
    private Territorio territorioVuoto;
    public void setUp() throws Exception
        this.territorioVuoto = new Territorio(200);
        for( Animale a : this.territorioVuoto.getAnimali()){
            this.territorioVuoto.rimuoviAnimale(a); }
```

→

```
pubblic void testAnno2Animali_TerritorioVuoto(){
    assertEquals(this.territorioVuoto.getAnimali().size(), 0);
    AssertTrue(this.territorioVuoto.getAnimali().isEmpty());
    Map<Integer, Set<Animale>> anno2animali;
    anno2animali = this.territorioVuoto.anno2animali();
    assertTrue(anno2animali.isEmpty()); }

pubblic void testAnno2animali_2animaliStessoEta(){
    assertTrue(this.territorioVuoto.getAnimali().isEmpty());
    Map<Integer, Set<Animale>> anno2animali;
    Animale animale1 = new Erbivoro();
    Animale animale2 = new Erbivoro();
    Posizione posizione1 = new Posizione(1,1);
    Posizione posizione2 = new Posizione(2,2);
    territorioVuoto.setAnimale(animale1, posizione1);
    territorioVuoto.setAnimale(animale2, posizione2);
    assertEquals(animale1.getAnni(), animale2.getAnni());
    anno2animali = this.territorioVuoto.anno2animali();
    assertEquals(anno2animali.size(), 1);
    assertTrue(anno2animali.containsKey(animale1.getAnni()));
    assertEquals(anno2animali.get(animale1.getAnni()).size(), 2);
    Set<Animale> animali = anno2animali.get(animale1.getAnni());
    assertTrue(animali.contains(animale1));
    assertTrue(animali.contains(animale2));

(nel SetUp devo metterle)

List<Animale> animali = new ArrayList<Animale>();
animali.addAll(this.territorioVuoto

pubeic Map<Integer, Set<Animale>> anno2animali(){
    Map<Integer, Set<Animale>> anno2animali = new HashMap<Integer,
    ,Set<Animale>>()

    for(Animale animale : this.getAnimali(){
        if(anno2animali.containsKey(animale.getAnni())){
            anno2animali.get(animale.getAnni()).add(animale);}
        else{
            Set<Animale> animaliConLoStessoAnno = new HashSet<Animale>();
            animaliConLoStessoAnno.add(animale);
anno2animali.put(animale.getAnno(), animaliConLoStessoAnno);}

    return anno2animali;
```

Programmazione Orientata agli Oggetti 29/05/12

Lezione XVIII

ESAME 2011 GIUGNO

@Test

```
public void testAnno2Animali_3animali2eta() {
    Map<Integer, Set<Animale>> anno2Animali;
    Animale animale1 = new Erbivoro();
    Animale animale2 = new Erbivoro();
    Animale animale3 = new Erbivoro();

    Posizione posizione1 = new Posizione(0,0);
    Posizione posizione2 = new Posizione(1,1);
    Posizione posizione3 = new Posizione(2,2);

    animale3.incrementaAnni();
    this.territorioVuoto.setAnimale(animale1, posizione1);
    this.territorioVuoto.setAnimale(animale2, posizione2);
    this.territorioVuoto.setAnimale(animale3, posizione3);

    assertEquals(animale1.getAnni(), animale2.getAnni());
    assertFalse(animale1.getAnni() == animale3.getAnni());
    anno2Animali = this.territorioVuoto.anno2Animali();
    assertEquals(anno2Animali.size(), 2);
    assertTrue(anno2Animali.containsKey(animale1.getAnni()));
    assertTrue(anno2Animali.containsKey(animale3.getAnni()));
    assertTrue(anno2Animali.get(animale1.getAnni()).contains(animale2));
    assertTrue(anno2Animali.get(animale3.getAnni()).contains(animale3));
    assertEquals(anno2Animali.get(animale1.getAnni()).size(), 2);
    assertTrue(anno2Animali.get(animale3.getAnni()).contains(animale3));
    assertEquals(anno2Animali.get(animale3.getAnni()).size(), 1); }
```

ESERCIZIO nella classe territorio scrivere il metodo <tt> Map<Integer, Integer> anno2numeroAnimali() </tt>: restituisce una mappa che ha per chiave un anno, e per il valore il numero di animali con tale anno.

```
public Map<Integer, Integer> anno2numeroAnimali() {
    Map<Integer, Integer> anno2numeroAnimali = new HashMap<Integer
    , Integer>();
    int cont = 0;
    for (Animale animale : this.getAnimali()) {
        if(anno2numeroAnimali.containsKey(animale.getAnni())){
    temp = anno2numeroAnimali.get(animale.getAnni());
    anno2numeroAnimali.put(animale.getAnni(), tempAnimaliStessoAnno+1
        else {
            anno2numeroAnimali.put(animale.getAnni(), 1); } }
    return anno2numeroAnimali;
```

GESTIONE DELLE ECCEZIONI
GESTIONE DEI FILE:

5/06/12

1/2

Gli errori che si incontrano generalmente sono dovuti ad alcune situazioni anomale causate dall'ambiente:
- URL o nome del file errato
- Hard Disk pieno
- Interruzione di rete

} o più genericamente: Mancanza di permessi appropriati per una risorsa

- DEFENSE PROGRAMMING: un programma OO può essere visto come un'interazione tra oggetti "client" e oggetti "server", dove gli oggetti SERVER offrono servizi invocati dagli oggetti client

→ LE ECCEZIONI: sono uno strumento attraverso il quale un oggetto Server, in caso di anomalia interrompe il flusso di esecuzione e comunica attraverso un oggetto
↳ il linguaggio supporta le eccezioni con costrutti per imporre al programmatore del client di scrivere codice per la gestione dell'eccezione

(!) Se si dovesse verificare una condizione anomala, un metodo può interrompere il flusso dell'esecuzione e lanciare una eccezione al client
L'eccezione è un oggetto e come tale può contenere informazioni sulla natura dell'errore. → Quando il client riceve un'eccezione a seconda del tipo di eccezione può ~~non~~ ignorarla oppure essere costretto a gestirla.

Standard Library

ERROR: non è proprio errori, non gestibile come casi eccezionali dovuti a ~~MyUnc~~ fattori esterni

CHECKED EXEPTION: Sotto classe di Exeption; il client è costretto a gestire questo tipo di eccezione; In genere si usa quando è possibile un ripristino

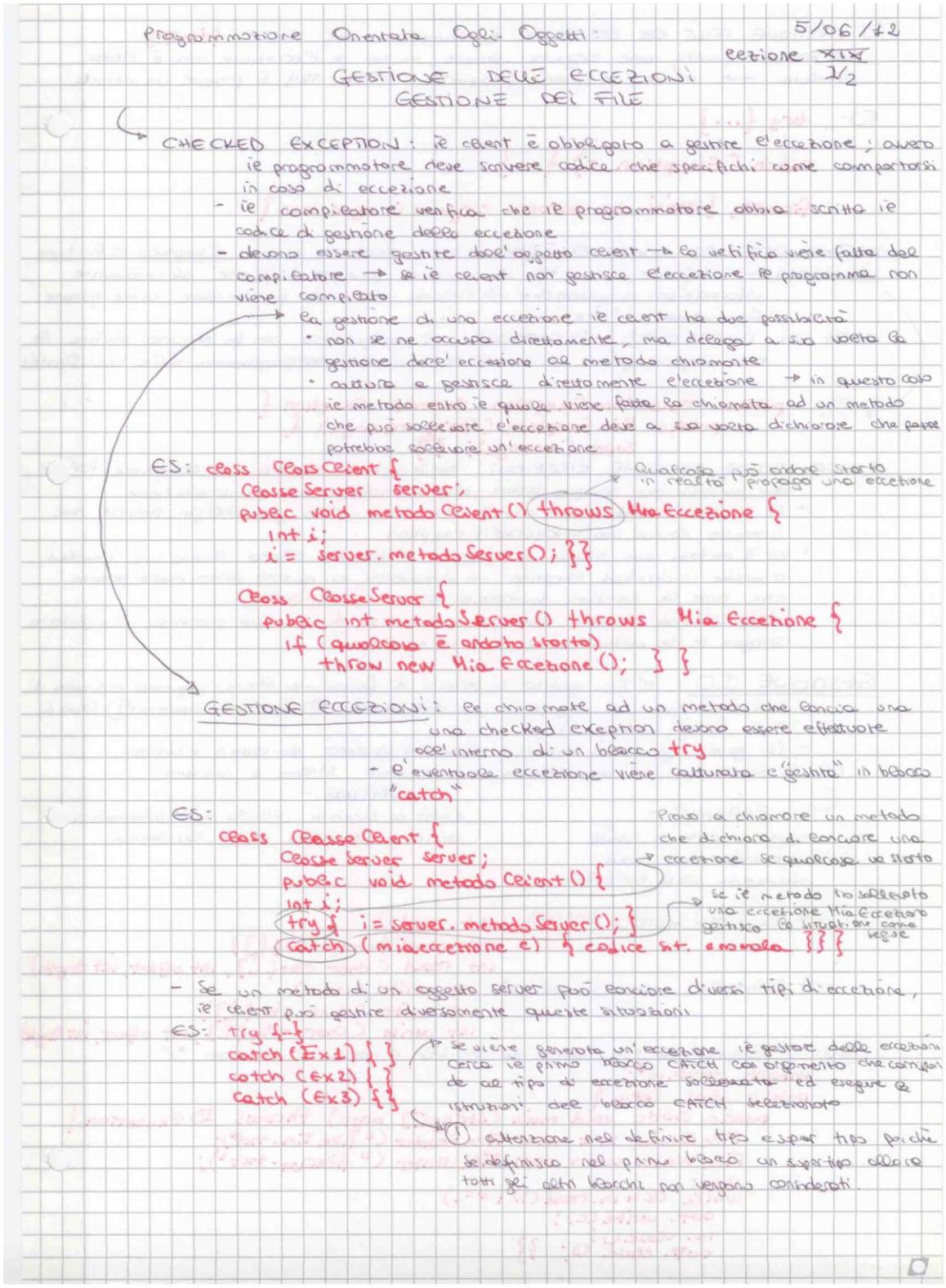
UNCHECKED EXEPTION: Sotto classe di RUNTIMEEXEPTION: il client non è costretto a gestire l'eccezione, se non viene gestita il programma abortisce.

Se qualcosa va storto lanciamo un eccezione → (costruzione) `new ExceptionType("");`
- I metodi che lanciano una checked exeption → `throw eccezione`

devono dichiararlo; sulla base di questa dichiarazione il compilatore verifica che i client che usano questo metodo hanno previsto codice per la gestione dell'eccezione

ES:
```
class ClasseClient {
  ClasseServer server;
  public void metodoClient() {
    int i;
    i= server.metodoServer();}}

class ClasseServer {
  public int metodoServer() throws MioEccezione {
    ...
    if(qualcosa è andata storto)
      throw new MioEccezione();}}
```

EFFETTO di un'ECCEZIONE: il metodo che lancia un eccezione finisce prematuramente. Non viene ritornato nessun valore
↳ il controllo non ritorna al punto di chiamata del client, ma il client può catturare e gestire l'eccezione



Programmazione Orientata Agli Oggetti 5/06/12
Lezione XIX 1/2

GESTIONE DELLE ECCEZIONI
GESTIONE DEI FILE

CHECKED EXCEPTION: il client è obbligato a gestire l'eccezione; ovvero il programmatore deve scrivere codice che specifichi come comportarsi in caso di eccezione
- il compilatore verifica che il programmatore abbia scritto il codice di gestione delle eccezione
- devono essere gestite dall'oggetto client → la verifica viene fatta dal compilatore → se il client non gestisce l'eccezione il programma non viene compilato

La gestione di una eccezione il client ha due possibilità
- non se ne occupa direttamente, ma dirotta a sua volta la gestione dell'eccezione al metodo chiamante
- cattura e gestisce direttamente l'eccezione → in questo caso il metodo entro il quale viene fatta la chiamata ad un metodo che può sollevare l'eccezione deve a sua volta dichiarare che esso potrebbe sollevare un'eccezione

ES:
```
class ClasseClient {
  ClasseServer server;
  public void metodoClient() throws MiaEccezione {
    int i;
    i = server.metodoServer(); }}

class ClasseServer {
  public int metodoServer() throws MiaEccezione {
    if (qualcosa è andato storto)
      throw new MiaEccezione(); } }
```
Qualcosa può andare storto in realtà propago una eccezione

GESTIONE ECCEZIONI: le chiamate ad un metodo che lancia una una checked exception devono essere effettuate all'interno di un blocco try
- l'eventuale eccezione viene catturata e "gestita" in blocco "catch"

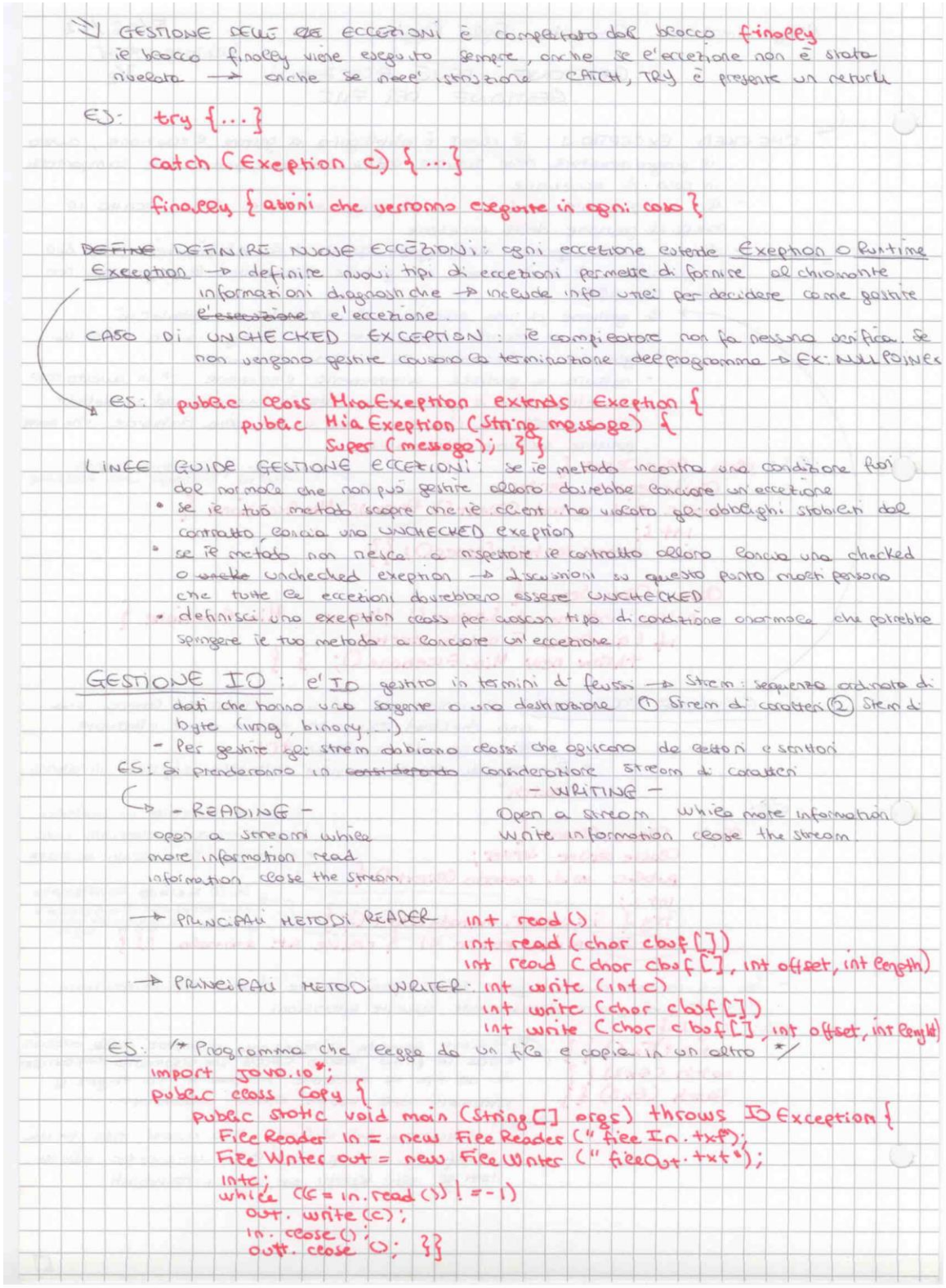
ES:
```
class ClasseClient {
  ClasseServer server;
  public void metodoClient() {
    int i;
    try { i = server.metodoServer(); }
    catch (miaeccezione e) { codice int. anomalo } } }
```
Provo a chiamare un metodo che dichiara di lanciare una eccezione se qualcosa va storto

Se il metodo ha sollevato una eccezione MiaEccezione gestisco la situazione come voglio

- Se un metodo di un oggetto server può lanciare diversi tipi di eccezione, il client può gestire diversamente queste situazioni

ES:
```
try {..}
catch (Ex1) { }
catch (Ex2) { }
catch (Ex3) { }
```
Se viene generata un'eccezione il gestore delle eccezioni cerca il primo blocco CATCH con argomento che combaci con il tipo di eccezione sollevata ed esegue le istruzioni del blocco CATCH selezionato

(!) attenzione nel definire tipo e super tipo poiché se definisco nel primo blocco un super tipo allora tutti gli altri blocchi non vengono considerati.



GESTIONE DELLE ECCEZIONI è completata dal blocco finally
il blocco finally viene eseguito sempre, anche se l'eccezione non è stata rilevata → anche se nessuna istruzione CATCH, TRY è presente un return

ES: try {...}
catch (Exception e) {...}
finally { azioni che verranno eseguite in ogni caso }

DEFINIRE NUOVE ECCEZIONI: ogni eccezione estende Exception o Runtime Exception → definire nuovi tipi di eccezioni permette di fornire al chiamante informazioni aggiuntive → include info utili per decidere come gestire l'eccezione.

CASO DI UNCHECKED EXCEPTION: il compilatore non fa nessuna verifica. Se non vengono gestite causano la terminazione del programma → Ex: NullPointer

ES: public class MiaException extends Exception {
public MiaException (String messaggio) {
super (messaggio); } }

LINEE GUIDE GESTIONE ECCEZIONI: se il metodo incontra una condizione fuori dal normale che non può gestire allora dovrebbe lanciare un'eccezione
- se il tuo metodo scopre che il client ha violato gli obblighi stabiliti dal contratto, lancia una UNCHECKED Exception
- se il metodo non riesce a rispettare il contratto allora lancia una checked o unchecked exception → discussioni su questo punto molti pensano che tutte le eccezioni dovrebbero essere UNCHECKED
- definisci una exception class per ciascun tipo di condizione anormale che potrebbe spingere il tuo metodo a lanciare un'eccezione.

GESTIONE IO: l'IO gestito in termini di flussi → Stream: sequenza ordinata di dati che hanno una sorgente o una destinazione ① Stream di caratteri ② Stream di byte (img, binary...)
- Per gestire gli stream abbiamo classi che offrono le letture e scritture

ES: Si prenderanno in considerazione stream di caratteri

- READING -
open a stream while more information read information close the stream

- WRITING -
Open a stream while more information write information close the stream.

→ PRINCIPALI METODI READER: int read()
int read (char cbuf[])
int read (char cbuf[], int offset, int length)

→ PRINCIPALI METODI WRITER: int write (int c)
int write (char cbuf[])
int write (char cbuf[], int offset, int length)

ES: /* Programma che legge da un file e copia in un altro */
import java.io.*;
public class Copy {
public static void main (String[] args) throws IOException {
FileReader in = new FileReader ("fileIn.txt");
FileWriter out = new FileWriter ("fileOut.txt");
int c;
while ((c = in.read()) != -1)
out.write(c);
in.close();
out.close(); }}

Programmazione orientata agli Oggetti 5/06/12
Lezione XX

RIFLESSIONE

La riflessione è una caratteristica del linguaggio JAVA. Permette di scrivere codice da cui funzionalità principale consiste nell'analizzare codice

↳ A classe e A interfaccia JAVA esiste un oggetto che descrive il contenuto del file .class → è istanza della classe Class del package java.lang

↳ tutte le classi hanno una variabile pubblica e statica chiamata class che memorizza un riferimento all'oggetto di tipo java.lang.class

Ex: Class classeBorsa = it.unioroma3.didadia.giocatore.Borsa.class

(!) si può interrogare l'oggetto class per ottenere le proprietà della classe

```
import java.lang.reflect.Method

pubblic class EsempioRiflessione {
    pubblic static void main (String[] args) {
    Class classeBorsa = it.unioroma3.it.didadia.giocatore.Borsa.class;
    for (Method m: classeBorsa.getMethods())
        System.out.println(m); }}
```

La riflessione si applica nello sviluppo di programmi orientati con funzionalità complesse

- offre inoltre la possibilità di creare oggetti a partire dal nome della classe
- Con la riflessione è possibile creare oggetti invocando il metodo OBJECT NEWISTANCE ()

Ex: Class classeBorsa = it.unioroma3.didadia.giocatore.Borsa.class
Borsa borsa = (Borsa) classeBorsa.newIstonce();

- Attraverso il metodo statico Class forName() possiamo cercare ed eventualmente caricare l'oggetto class di una classe dato il suo nome sotto forma di Stringa.

Ex: Class classeBorsa = Class.forName("it.unioroma3.didadia.Borsa");
Borsa borsa = (Borsa) classeBorsa.newIstonce();

- Si ~~possono~~ può scrivere codice in cui vengono creati dinamicamente oggetti a partire dal nome della loro classe.
Il metodo newIstonce() invoca il costruttore no-arg, per questo motivo nella interfaccia comando abbiamo introdotto il metodo setArgomento(String); sarebbe stato meglio passare l'argomento attraverso un costruttore

ESEMPIO:

```
public class FabbricaDiComandiRiflessiva implements FabbricaDiComandi {
    public Comando costruisciComando (String istruzione) {
        Scanner scannerDiParole = new Scanner (istruzione);
        String nomeComando = null;
        String parametro = null;
        Comando comando = null;

        if (scannerDiParole.hasNext())
            nomeComando = scannerDiParole.next();   // nome del comando
        if (scannerDiParole.hasNext())
            parametro = scannerDiParole.next();  // secondo parola eventuale (parametro)

        try {
            String nomeClasse = "it.uniroma3.diadia.comandi.Comando";
            nomeClasse += Character.toUpperCase(nomeComando.charAt(0));
            nomeClasse += nomeComando.substring(1);
            comando = (Comando) Class.forName(nomeClasse).newInstance();
            comando.setParametro(parametro);
        }
        catch (Exception e) {
            comando = new ComandoNonValido();
            comando.setParametro("Comando Inesistente");
        }
        return comando;
```