Programmazione II

Kristian Xhani

October 2024

Indice

1 Introduzione

Java nasce negli anni 90 dopo C, però il concetto di **programmazione ad oggetti** esisteva già negli anni 60. I concetti fondamentali della programmazione ad oggetti sono :

- **Ereditarietà:** è un principio nel quale una classe definita come sottoclasse eredita attributi e metodi fa un'altra classe detta superclasse,cosi da poter riutilizzare il codice e crea una sorta di gerarchia tra le classi.
- **Incapsulamento:**è il processo per nascondere i dettagli di un oggetto rendendo accessibili solo alcune parti
- **Polimorfismo:**è la capacità di oggetti di diverse classi di rispondere allo stesso metodo in maniera diversa

NB. questi concetti veranno ripresi nei capitoli futuri

1.1 Esecuzione di un programma in Java

Sappiamo da programmazione I che un file ad esempio pippo.c viene trasformato in eseguibile attraverso il commando **gcc nomefile.c**. Facendo così verrà generato un file eseguibile solamente dal sistema operativo nel quale si è generato il file. Invece con java non è proprio la stessa cosa perchè prendendo in esempio lo stesso file però denominato pippo.java è usando il commando **javac nomefile.java** verrà formato un file con l'estensione **.class** esso rappresenta un **bytecode**. I file .class possono essere eseguiti solo da un interprete Java,esso è un vantaggio perchè si potra eseguire in ogni sistema operativo **solamente** se si conterrà un interprete java. Da qua in poi ogni linguaggio di programmazione utilizzerà questo concetto per poter eseguire i file. NB.il file java deve sempre contenere la prima lettera maiuscola.

```java
public class HelloWorld {
    public static void main(String[] args) {
        // Stampa "Hello, World!" nella console
        System.out.println("Hello, World!");
    }
}
```

Ora generiamo l'eseguibile:

```
>>javac Hello.java  //genererà il file Hello.class
>>java Hello        //esegue il file
>>Hello, World!
```

In questo corso useremo **Eclipse for java developers**.

1.2 I tipi di variabili

I tipi primitivi presenti in Java sono 8:

1. boolean (o True o False)
2. char (singolo carratere)
3. byte (8 bit)
4. short (16 bit)
5. int (32 bit)
6. long (64 bit)
7. float (rappresenta numeri in virgola mobile) (32)
8. double (rappresenta numeri in virgola mobile) (64)

Inoltre esistono i tipi **var** i quali rappresentano un qualsiasi tipo di variabile e vengono identificati proprio da java. Per rappresentare una costante bisogna scrivere **final tipo nome**.

```
1 public class variabili {
2     public static void main(String[] args) {
3         final int contatore = 0; // costante
4         var variabile ;// java assegnera il tipo
5     }
6 }
```

NB. di solito si preferisce non usare il tipo var per la leggibilità del codice.

2 Cosa sono gli Oggetti?

La principale differenza tra Java e C è che Java è un linguaggio di programmazione orientato agli oggetti, mentre C è un linguaggio procedurale. Un **oggetto** è un'**istanza della classe**(un oggetto creato da una detterminata classe),che rappresenta un'entità concreta o astratta. In Java gli oggetti allocano zone di memoria.

2.1 Classi,Metodi e Attributi

Una **classe** è un modello che definisce le caratteristiche e i **comportamenti** di un gruppo di oggetti. In pratica, è un tipo di dato definito dall'utente che rappresenta una categoria o un'entità astratta. Ogni oggetto che appartiene a una classe ha le stesse proprietà (attributi) e può eseguire gli stessi comportamenti (metodi).
Gli **attributi** sono le variabili che memorizzano le **caratteristiche di una classe**. Ogni istanza (oggetto) di una classe ha i propri attributi. Gli attributi possono essere di classe (condivisi da tutte le istanze) o di istanza (specifici di ogni oggetto creato da una classe).
I **metodi** sono funzioni definite all'interno di una classe che descrivono i comportamenti che gli oggetti della classe possono eseguire. I metodi possono manipolare gli attributi e possono essere invocati sugli oggetti della classe.

2.2 Come si crea un oggetto in Java

Un oggetto in Java si dichiara nel seguente modo : new Classe (eventuali parametri).Ad esempio con la Classe Scanner (questa classe serve per prendere in input un dato):

```
1 import java.Util.Scaner // libreria da incorporare
2 public class CreareObj {
3     public static void main(String[] args) {
4         Scanner Keyboard = new Scanner (System.In);
5     }
6 }
```

3 Scanner e String

Lo String in Java non bisogna pensarla come un array o una concatenazione di char ma come un oggetto al quale viene allocato una cella di memoria. Per quanto riguarda String la sua libreria risulta già incorporata in Java quindi non serve definirla(java.lang.String). Facciamo un esempio nel quale cerchiamo di stampare il valore che l'utente ci invia:

```
1 include java.Util.Scanner
2 public class Papagallo {
```

```
3       public static void main(String[] args) {
4           Scanner Keyboard = new Scanner (System.In);
5           String line = Keyboard.nextLine(); // usata per
                prendere le stringhe
6
7           System.out.println(line);
8           Keyboard.close(); // questa e una chiamata a metodo
9       }
10  }
```

Le diverse Funzionalita di **Scanner** che vedremmo inq uesto corso sono:

- **Scanner(source)** (costruttore, che crea uno Scanner legato alla sorgente indicata)
- **void close()** (chiude lo Scanner: dopo non può più essere usato)
- **double nextDouble()**
- **float nextFloat()**
- **int nextInt()**
- **String nextLine()**
- **long nextLong()**

Invece i diversi metodi di **String** sono:

- **String(String other)** (costruttore di copia: crea un clone)
- **char charAt(int index)** (esso prende in input l'indice e in ouput ti da in corrispondenza dell'indice il char)
- **int compareTo(String other)** (ritorna negativo, zero, oppure positivo)
- **int compareToIgnoreCase(String other)** (ritorna negativo, zero, oppure positivo)
- **String concat(String other)** (implicitamente usato per la concatenazione con +)
- **boolean endsWith(String end)** (Viene utilizzato per verificare se una stringa termina con un particolare suffisso specificato. In altre parole, controlla se la parte finale della stringa corrente corrisponde esattamente alla stringa passata come argomento.)

```
1   public class Main {
2       public static void main(String[] args) {
3           String str = "ciao mondo";
4
5           System.out.println(str.endsWith("mondo"));  //
                true
```

```
6           System.out.println(str.endsWith("ciao"));    //
                false
7           System.out.println(str.endsWith("do"));      //
                true
8       }
9   }
```

- **boolean equals(Object other)** (controlla se due oggetti hanno la stessa informazione/contenuto)
- **boolean equalsIgnoreCase(String other)**(controllo se due stringhe sono uguali ignorando la differenza tra maisucole e minuscole)
- **static String format(String format, Object... args)** (della classe String in Java viene utilizzato per creare una nuova stringa formattata. Questo metodo funziona in modo simile a quello che trovi in altri linguaggi, come printf in C. In pratica, consente di inserire dei segnaposto all'interno della stringa e sostituirli con i valori forniti come argomenti.)

```
1   public class Main {
2       public static void main(String[] args) {
3           String nome = "Mario";
4           int eta = 30;
5
6           // Formattazione della stringa con String.format
                ()
7           String risultato = String.format("Ciao, mi
                chiamo %s e ho %d anni.", nome, eta);
8           System.out.println(risultato);
9       }
10  }
```

- **int indexOf(int character)** (l'indice della prima occorrenza di un carattere specificato all'interno della stringa. Se il carattere non viene trovato nella stringa, il metodo restituisce -1.)
- **int indexOf(String what)** (restituisce l'indice della prima occorrenza della sottostringa specificata all'interno della stringa su cui viene chiamato il metodo. Se la sottostringa non viene trovata, restituisce -1.)

```
1   public class Main {
2       public static void main(String[] args) {
3           String str = "Benvenuto nel mondo di Java";
4
5           // Trova la prima occorrenza della sottostringa
                "mondo"
6           int index = str.indexOf("mondo");
7           System.out.println("Indice della sottostringa '
                mondo': " + index);  // 14
```

```
8
9           // Trova la prima occorrenza della sottostringa
                "Java"
10          index = str.indexOf("Java");
11          System.out.println("Indice della sottostringa '
                Java': " + index);  // 21
12
13          // Se la sottostringa non e presente
14          index = str.indexOf("Python");
15          System.out.println("Indice della sottostringa '
                Python': " + index);  // -1
16      }
17  }
```

- **boolean isEmpty()** (utilizzato per verificare se una stringa è vuota, ovvero se non contiene caratteri. Una stringa è considerata vuota se la sua lunghezza è zero ("").)
- **int length()** (restituisce la lunghezza della stringa, ovvero il numero di caratteri che essa contiene. Questo include anche spazi, simboli e lettere. La lunghezza è contata a partire da 1, quindi se la stringa è vuota, il metodo restituisce 0.)
- **boolean startsWith(String what)** (utilizzato per verificare se una stringa inizia con una sottostringa specificata. Questo metodo restituisce true se la stringa inizia con la sottostringa specificata, altrimenti restituisce false.)
- **String substring(int start)** (da start incluso)
- **String substring(int start, int end)** (da start incluso ad end escluso)
- **String toLowerCase()** (viene utilizzato per convertire tutti i caratteri di una stringa in minuscolo. Questo metodo è utile quando si desidera uniformare il caso dei caratteri, ad esempio per confronti o per formattazione.)
- **String toUpperCase()** (contrario toLowerCase())
- **String trim()** (rimuovere gli spazi bianchi all'inizio e alla fine di una stringa. Questo è utile per pulire le stringhe di input, in particolare quando si lavora con dati forniti dagli utenti, in cui possono esserci spazi indesiderati.)
- **static String valueOf(int i)** (esegue una conversione esplicita di tipo; esiste per tutti i tipi primitivi, non solo per int; implicitamente usato per la concatenazione con +)

Facciamo un esempio con String per ragionarci su prendendo la classe di prima Papagallo:

```
1  include java.Util.Scanner
2  public class Papagallo {
3      public static void main(String[] args) {
4          Scanner Keyboard = new Scanner (System.In);
5          do{
6              String line = Keyboard.nextLine();
7              System.out.println(line);
8              Keyboard.close();
9          }while(line != "fine")
10     }
11 }
```

Possiamo notare che in riga 6 viene dichiarata una nuova variabile,c'è una regola fondamentale ovvero che:

> **Importante**
>
> La durata di vita di una variabile e da dove la dichiaro fino alla prima graffa di chiusura.

Allora facciamo cosi:

```
1  include java.Util.Scanner
2  public class Papagallo {
3      public static void main(String[] args) {
4          Scanner Keyboard = new Scanner (System.In);
5          String line;
6          do{
7              line = Keyboard.nextLine();
8              System.out.println(line);
9              Keyboard.close();
10         }while(line != "fine")
11     }
12 }
```

Però non funziona lo stesso perchè "fine" è un oggetto String allocato in una memoria,in Java, l'operatore == confronta i riferimenti (o gli indirizzi di memoria) degli oggetti, non il loro contenuto. Quando hai a che fare con oggetti di tipo String, usare == non confronta il contenuto delle stringhe, ma verifica se entrambi i riferimenti puntano alla stessa posizione di memoria.(stessa cosa con !=). L'unico modo allora è usare uno dei metodi presente su String overo **.equals()** in questo modo:

```
1  include java.Util.Scanner
2  public class Papagallo {
3      public static void main(String[] args) {
4          Scanner Keyboard = new Scanner (System.In);
5          String line;
6          do{
```

```
            line = Keyboard.nextLine();
            System.out.println(line);
            Keyboard.close();
        }while(!line.equals("fine"))
    }
}
```

4 Regole di programmazione

4.1 Concatenazione

Per usare la concatenazione si utilizza il simbolo +.

```
1 public class concatenamento {
2     public static void main(String[] args) {
3         int contatore = 0;
4         System.out.Println("Contatore : " + contatore);
5     }
6 }
```

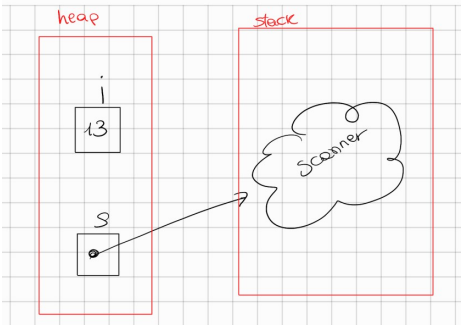
NB.non posso concatenare **variabile + variabile** e mandare in output.

4.2 Operazioni aritmetiche

Le operazioni aritmetiche sono **C style**(anche l'assegnamento e l'incrementamento). Esistono le operazioni **aritmetiche ibride** tra tipi nel quale prevale il tipo più grande.

4.3 Tipi primitivi e Tipi riferimento

Nel linguaggio Java esistono due tipi di valori i valori primitivi per esempio **int i = 13** e **Scanner n = new Scanner(System.in)**. La differenza e che il tipo primitivo dentro la variabile ci sta un valore invece dentro il tipo riferimento il valore viene riferito a partire della variabile. Possiamo immaginarli come due contenitori dove nel primo ci sta proprio il valore invece nel secondo contiene un riferimento all'oggetto.



Come possiamo vedere dall'immagine le variabili primitive stanno nella memoria heap della ram invece l'oggetto puntato sta nella memoria stack. Proprio per questo è molto sbagliato fare i.lenght().
NON BISOGNA MAI CHIAMARE METODI SUI TIPI PRIMITIVI Invece per buona informatica e giusto definire la classe con la prima lettera maiuscola anche durante la fase di coding quando si chiama un'oggetto definirlo con lettera maiuscola.

4.4 NULL

null in programmazione rappresenta un valore speciale utilizzato per indicare che una variabile di tipo riferimento non punta a nessun oggetto o dato valido. È una sorta di "segnaposto" che dice: "Questa variabile esiste, ma non contiene attualmente nessun valore o oggetto valido."

```
1
2
3  public class Main {
4      public static void main(String[] args) {
5          String s;
6          s = null;
7          int l = s.lenght();
8      }
9  }
```

Se si compila il programma esso darà un errore in riga 7 ovvero **Null pointer exeption** ovvero stiamo provare un metodo su un puntatore a null. Il vantaggio e che possiamo assegnarlo a ogni oggetto lo svantaggio e che ti darà sempre errore in fase di esecuzione. Lo svantaggio dell'inizializzazione con null, come hai detto, è che devi gestire esplicitamente questi casi per evitare errori di runtime, il che può essere fonte di problemi se non ci si fa attenzione. Ad esempio si puo risolvere in questo modo:

```
1  public class Main {
2      public static void main(String[] args) {
3          String s = null;
4
5          if (s != null) {
6              int l = s.length();
7              System.out.println("La lunghezza della stringa e
                   : " + l);
8          } else {
9              System.out.println("La stringa e null");
10         }
11     }
12 }
```

5 Classi Random e Math

5.1 Random

La libreria designata per usare un oggetto random è **java.util.Random**. Per creare un oggetto random bisogna eseguire il seguente commando : **Random r = new Random();**
Vediamo ora altri metodi di questa funzione:

- **boolean nextBoolean()** (genera un numero booleano randomico)
- **double nextDouble()**
- **float nextFloat()**
- **int nextInt()**
- **int nextInt(int max)** (restituisce un numero casuale tra 0 e max escluso)
- **long nextLong()**

Vediamo ora un'applicazione pratica:

```
1  // generare un intero randomico e mandarlo a video
2  import java.Util.Scanner
3  import java.Util.Random
4
5  public class Main {
6      public static void main(String[] args) {
7          Random r = new Random();
8          int a = r.nextInt();
9          System.out.print(a);
10     }
11 }
```

5.2 Math

La libreria designata per un oggetto Math è **java.until.Math**
Vediamo ora altri metodi di questa funzione:

- **static double E** (costante della classe)
- **static double PI** (costante della classe)
- **static int abs(int i)** (esiste anche per altri tipi numerici)
- **static double cos(double d)**
- **static double log(double d)** (in base e)
- **static double log10(double d)** (in base 10)

- **static int max(int a, int b)** (esiste anche per altri tipi numerici)
- **static int min(int a, int b)** (esiste anche per altri tipi numerici)
- **static double sin(double d)**
- **static double sqrt(double d)**
- **static double tan(double d)**
- **static double toDegrees(double radiants)**
- **static double toRadiants(double degrees)**

6 Creazione di una Classe

6.1 Creazione di una classe e utilizzo

Creiamo un file Date.java e ci scriviamo il seguente codice :

```
public class Date {
    // Attributi
    int day;
    int month;
    int year;
}
```

Invece nel file MainDate ci scrivo :

```
public class MainDate {
    public static void main(String[] args){
        Date d1;
        Date d2;
        d1 = new Date();
        d2 = new Date();
        System.out.println(d1.day); // mando in output day
    }
}
```

In output otterò 0 senza scrivere nulla,però qua non ho inizializzato nulla perchè? Perche gli attributi non inizializzati tengono il valore di default 0 poi varia a seconda del tipo per bool è false,per int è o,per float/double è 0.0,per gli oggetti è NULL. posso anche scriverci facendo cosi:

```
public class MainDate {
    public static void main(String[] args){
        Date d1;
        Date d2;
        d1 = new Date();
        d2 = new Date();
        d1.day = 11;
        d1.month = 10;
        d1.year = 2021;
        System.out.println(d1.day);
    }
}
```

6.2 Attributi

Se mando in esecuzione effettivamente mi stampa il giorno però questo modo di scrittura ricorda molto il C ma quindi **non è a oggetti**. Pensiamo al termine di **incapsulazione** che ci dice che serve per "nascondere" i dettagli di un oggetto,noi invece stiamo violando una regola della programmazione ad oggetti

Ma com'è possibile che MainDate acceda direttamente agli attributi?
Bisognerebbe dichiarare gli oggetti private in questo modo :

```
public class Date {
    // Attributi
    private int day;
    private int month;
    private int year;
}
```

Di default gli attributi sono **public**.

6.3 Costruttore

Sorge un'altro problema ovvero e ora come li richiamiamo nel main??
Si utilizza quello che viene detto **costruttore** ovvero un qualcosa che serve per creare oggetti della classe definita. Esso si chiama riscrivendo il nome della classe,invece all'interno delle parentesi tonde servono gli elementi che devi dichiarare **PER FORZA** per creare quel tipo di oggetto

```
public class Date {
    // Attributi
    private int day;
    private int month;
    private int year;

    // costruttore
    public Date(int d,int m,int y){
        this.day = d;
        this.month = m;
        this.year = y;
    }
}
```

NB. this non serve per forza chiamarlo però se non ci fosse si potrebbe creare ambiguità
E quindi ora MainDate per compilare deve essere cosi:

```
public class MainDate {
    public static void main(String[] args){
        Date d1;
        Date d2;
        d1 = new Date(11,10,2021);
        d2 = new Date(13,1,2022);
    }
}
```

6.4 Metodi

Ora pero vogliamo anche aver la possibilità di stampare,essendo che gli attributi sono private ce bisogno di un nuovo **metodo**. Allora facciamo in questo modo prendendo il file della classe:

```
public class Date {
    // Attributi
    private int day;
    private int month;
    private int year;

    // costruttore
    public Date(int d,int m,int y){
        this.day = d;
        this.month = m;
        this.year = y;
    }

    //metodi
    public String toString(){
        String result = this.day + "/" + this.month + "/" +
            this.year;
        return result;
    }
}
```

6.5 Stampare un Oggetto toString()

Ora Proviamo a chiamare i metodi nel file MainDate in questo modo:

```
public class MainDate {
    public static void main(String[] args){
        Date d1;
        Date d2;
        d1 = new Date(11,10,2021);
        d2 = new Date(13,1,2022);
        String s1;
        s1 = d1.toString();
        System.out.println(s1);
    }
}
```

Facendo così avremmo in output 11/10/2021.
NB nei linguaggi tradizionali vengono posti gli attributi in maniera implicita ovvero passarli direttamente al metodo invece qua in Java si mette in maniera esplicita.
In Java però ce una cosa figa se tu chiami il metodo toString() puoi fare anche una roba del genere:

```java
public class MainDate {
    public static void main(String[] args){
        Date d1;
        Date d2;
        d1 = new Date(11,10,2021);
        d2 = new Date(13,1,2022);
        String s1;
        s1 = d1.toString();
        System.out.println(s1);
        System.out.println(s2);// sottointeso d2.toString()
    }
}
```

Come si puo vedere s2 verrà lo stesso stampata perchè viene sottointeso se presente il metodo toString().

6.6 Comparare un oggetto che ho creato:

Come detto nei capitoli precedenti in Java usare l'operatore == con dei oggetti non ha senso perchè confronterà la loro zona di memoria. Allora creo un metodo che solitamente è chiamato equals(); in questo modo:

```java
public class Date {
    // Attributi
    private int day;
    private int month;
    private int year;

    // costruttore
    public Date(int d,int m,int y){
        this.day = d;
        this.month = m;
        this.year = y;
    }

    //metodi
    public String toString(){
        String result = this.day + "/" + this.month + "/" +
            this.year;
        return result;
    }
    public boolean equals(Date other){
        boolean result = this.day == other.day && this.month
             == other.month && this.year == other.year;
        return result;
    }
}
```

Come possiamo notare dal codice sopra capiamo che other specifica tutti gli attributi di un classe. Ora utilizziamo questo metodo nel main in questo modo:

```
public class MainDate {
    public static void main(String[] args){
        Date d1;
        Date d2;
        d1 = new Date(11,10,2021);
        d2 = new Date(13,1,2022);
        d3 = new Date(13,1,2022);
        String s1;
        s1 = d1.toString();
        System.out.println(s1);
        System.out.println(s2);// sottointeso d2.toString()
        boolean b1 = (d1.equals(d2));
        System.out.println(b1);
        boolean b2 = (d2.equals(d3));
        System.out.println(b2);
    }
}
```

Facendo così notiamo che b1 giustamente è uguale a false invece b2 è uguale a true.

6.7 Compariamo due oggetti

Per lo stesso motivo di prima anche comparare due oggetti normalmente usando l'operatore $>$ non ha senso allora si dovra creare un nuovo metodo in questo modo:

```
public class Date {
    // Attributi
    private int day;
    private int month;
    private int year;

    // costruttore
    public Date(int d,int m,int y){
        this.day = d;
        this.month = m;
        this.year = y;
    }

    //metodi
    public String toString(){
        String result = this.day + "/" + this.month + "/" +
            this.year;
        return result;
    }
    public boolean equals(Date other){
```

```
20          boolean result = this.day == other.day && this.month
                == other.month && this.year == other.year;
21          return result;
22      }
23      //convenzione(visto che uso int):
24      //<0 : this viene prima di other
25      //>0 : this viene dopo other
26      //== 0 : this e other coincidono cronologicamente
27      public int compareTo(Date other){
28          if(year<other.year){
29              return -1;
30          }else if(year>other.year){
31              return 1;
32          }else if (month<other.month){
33              return -1;
34          }else if(month>other.month){
35              return 1;
36          }else if(day<other.day){
37              return -1
38          }else if(day>other.day){
39              return 1;
40          }else{
41              return 0;
42          }
43      }
44  }
```

Creando un oggetto del genere e richiamando il metodo nel main otterò l'effetto di una comparazione.

7 Meccanismo di Esecuzione dei Programmi nei Computer

Noi ci baseremo sul codice che abbiamo svolto la lezione scorsa:

```
1 public class MainDate {
2     public static void main(String[] args){
3         Date d1 = new Date (12,11,2021);
4         Date d2 = new Date (13,1,2022);
5         boolean b = d1.equals(d2);
6     }
7 }
```

7.1 Concetto di variabili locali

Le **variabili locali** sono variabili definite all'interno di una funzione, di un metodo o di un blocco di codice e sono accessibili solo all'interno di quello specifico contesto in cui sono state dichiarate. Per esempio nel codice sopra le variabili locali del main saranno d1,d2,b e args.

7.2 Stack

Quindi quando il main va in esecuzione ci saranno 4 variabili locali che staranno da qualche parte. Le rappresentiamo in questo modo:

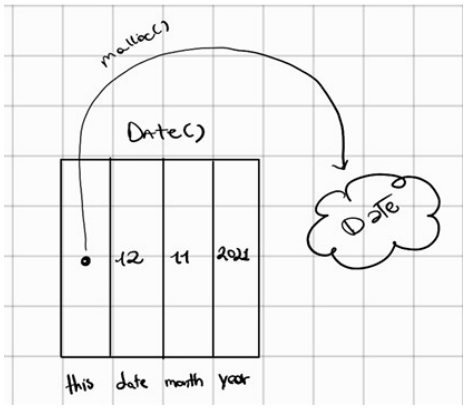
Questa zona di memoria in cui sono allocate le variabili prende nome di **stack**. Lo stack (in italiano, "pila") è una struttura dati fondamentale che segue il principio **LIFO** (Last In, First Out), ovvero l'ultimo elemento inserito è il primo a essere rimosso.

Pensiamo ora al costruttore riportato qua sotto :

```java
public class Date {
    // Attributi
    private int day;
    private int month;
    private int year;

    // costruttore
    public Date(int d,int m,int y){
        this.day = d;
        this.month = m;
        this.year = y;
    }
}
```

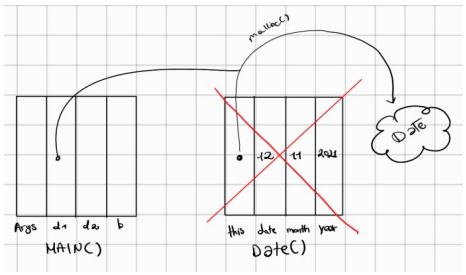
Il costruttore di date avrà le seguenti variabili locali day,month,year e this però a differenza di prima queste variabli sono inizializzate e si possono rappresentare in questo modo:



Questa e una configurazione in cui parte date invece il Main() in questo momento la sua esecuzione viene sospesa e tocca al costruttore di Date().

7.3 Record di attivazione

Il costruttore inizializza le variabili e, una volta completata questa operazione, termina la sua esecuzione. Di conseguenza, la memoria allocata per le sue variabili locali viene liberata, poiché non sono più necessarie dopo la fine del costruttore. Ora sarà rappresentato cosi il nostro stack :



Ogni blocchettino nel Main() prende nome di **record di attivazione** è una struttura di dati utilizzata durante l'esecuzione di un programma per memorizzare tutte le informazioni necessarie per gestire una singola invocazione di una funzione o di un metodo. L'insieme di più record di attivazione è detto **stack di attivazione**. Se andiamo avanti nel codice definiamo d2 nello stesso modo in cui abbiamo definito d1 è cosi via per ogni funzione.

7.4 final

La parola chiave **final** in Java, quando applicata ad un attributo (variabile di istanza), significa che quella variabile non può essere modificata dopo che è stata inizializzata. Questo è utile per creare **variabili immutabili**, come nel caso di una data, in cui giorno, mese e anno non dovrebbero essere modificati dopo la creazione dell'oggetto.

```java
public class Date {
    // Attributi
    private final int day;
    private final int month;
    private final int year;

    // costruttore
    public Date(int d,int m,int y){
        this.day = d;
        this.month = m;
        this.year = y;
    }
}
```

in questo modo gli attributi non possono venir chiamati da nessuna parte tranne nel costruttore inoltre final impone un **vincolo a livello di compilazione**, garantendo che le variabili non siano modificabili dopo essere state assegnate, rendendo il tuo programma più sicuro.

7.5 enum

In java enum è un tipo di dato che rappresenta un insieme fisso di costanti, come un gruppo di valori predefiniti che non cambiano. Gli enum sono utili quando hai

bisogno di rappresentare un numero limitato e ben definito di valori che possono essere assegnati a una variabile.Per esempio nella nostra classe data abbiamo bisogno di scrivere la data in maniera "Americana" ovvero scrivere prima il il mese rispetto al giorno e una data in maniera "Italiana".Allora creiamo un nuovo file denominato Language.java e riportiamo il seguente codice:

```
public enum Language{
    ITALIAN,
    AMERICAN
}
```

Ora questo enum può essere specificato con un tipo nella classe in questo modo :

```
public class Date {
    // Attributi
    private final int day;
    private final int month;
    private final int year;
    private Language language;

    // costruttore
    public Date(int d,int m,int y){
        this.day = d;
        this.month = m;
        this.year = y;
        this.language = Language.ITALIAN;
    }
}
```

Abbiamo anche inizializzato la classe in italiano.

7.5.1 Metodi d'uso frequente delle classi E definite tramite enum

- **static E[] values()** (ritorna l'array di tutti gli elementi dell'enumerazione)
- **static E valueOf(String name)** (ritorna l'elemento dell'enumerazione che ha il nome indicato)
- **int compareTo(E other)** (determina chi viene prima nell'enumerazione)
- **int ordinal()** (ritorna il numero d'ordine di un elemento dell'enumerazione)

7.6 Metodi set

Le funzioni che iniziano con set sono chiamate **setter** o **metodi mutatori**, e sono utilizzate per modificare il valore di un attributo privato di una classe. In un contesto di programmazione orientata agli oggetti, gli attributi delle classi sono

spesso definiti come privati per proteggere i dati e garantire l'incapsulamento. Nel nostro caso faremmo così :

```
public class Date {
    // Attributi
    private final int day;
    private final int month;
    private final int year;
    private Language language;

    // costruttore
    public Date(int d,int m,int y){
        this.day = d;
        this.month = m;
        this.year = y;
        this.language = Language.ITALIAN;
    }
    // metodi
    public void setItalian(){
        this.language = Language.ITALIAN;
    }

    public void setAmerican(){
        this.language = Language.AMERICAN;
    }

}
```

7.7 Static

Il modificatore static in Java è utilizzato per definire membri di classe (attributi o metodi) che appartengono alla classe stessa piuttosto che alle singole istanze (oggetti) della classe. Questo significa che un membro static è condiviso da tutte le istanze della classe e può essere usato senza dover creare un'istanza della classe.

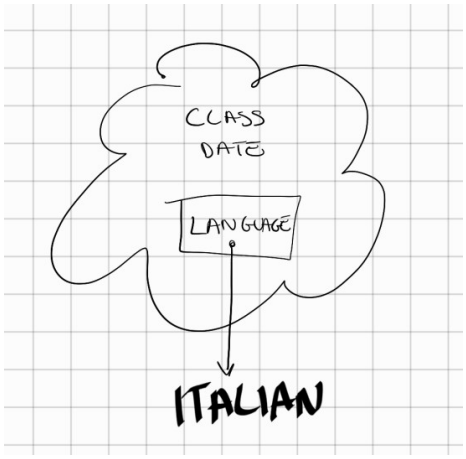
```
public class Date {
    // Attributi
    private final int day;
    private final int month;
    private final int year;
    private static Language language;

    // costruttore
    public Date(int d,int m,int y){
        this.day = d;
        this.month = m;
        this.year = y;
        this.language = Language.ITALIAN;
```

```
    }
    // metodi
    public void setItalian(){
        this.language = Language.ITALIAN;
    }

    public void setAmerican(){
        this.language = Language.AMERICAN;
    }
}
```

Cosa sta succedendo ?



Facendo se io cambiassi la lingua a un solo oggetto la lingua verrà cambiata a tutti gli oggetti.Però scritta in questo modo non va bene si dovrebbe scrivere così:

```
public class Date {
    // Attributi
    private final int day;
    private final int month;
    private final int year;
    private static Language language;

    // costruttore
    public Date(int d,int m,int y){
        this.day = d;
        this.month = m;
        this.year = y;
        this.language = Language.ITALIAN;
    }
    // metodi
    public static void setItalian(){
        Date.language = Language.ITALIAN;
```

```
    }

    public static void setAmerican(){
        Date.language = Language.AMERICAN;
    }
}
```

Invece nel main lo richiamiamo in questo modo:

```
public class MainDate {
    public static void main(String[] args){
        Date d1 = new Date (12,11,2021);
        Date d2 = new Date (13,1,2022);
        Date.setLanguge(Language.American);
    }
}
```

I campi statici si utilizzano poche volte.

7.8 tipo Array

Vogliamo far in modo che venga stampato il mese a parole allora in questo caso introduciamo il tipo array e scriviamo la classe in questo modo :

```
public class Date {
    // Attributi
    private final int day;
    private final int month;
    private final int year;
    private static Language language;

    private static String[] months = {
        "gennaio","febbraio","marzo","aprile","maggio","
           giugno","luglio","agosto","settembre","ottobre","
           novembre","dicembre"
    }

    private static String[] americanmonth = {
        "january","February","March","April","May","June","
           July","August","September","October","November","
           December"
    }
    // costruttore
    public Date(int d,int m,int y){
        this.day = d;
        this.month = m;
        this.year = y;
        this.language = Language.ITALIAN;
    }
    // metodi
```

```
23      public String toString(){
24          if(language == Language.ITALIAN){
25              return day + " " + months[month - 1]+" "+year;
26          }else{
27              return americanMonths[month - 1]+" "+day+","+
                    year;
28          }
29      }
30  }
```

Facendo così posso stampare le date in entrambe le maniere

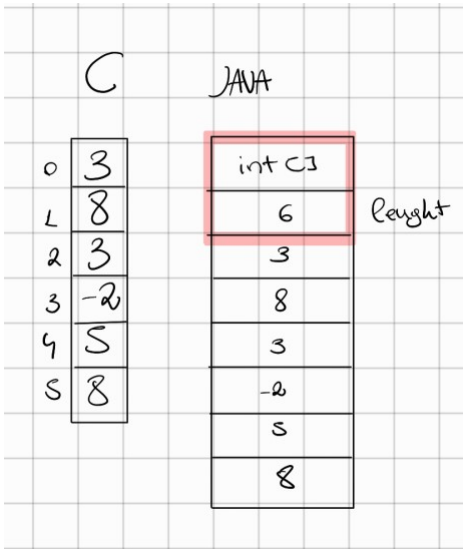
8 Array e Matrici in Java

Come abbiamo visto in programmazioneI gli array sono strutture di dati che permettono di memorizzare una sequenza di elementi dello stesso tipo, organizzati in posizioni contigue di memoria. Ogni elemento in un array è accessibile tramite un indice numerico. Per definire un array in Java partiamo da questo esempio creando la classe MainArray in questo modo:

```
1  public class MainArray{
2
3      public static void main(String[] args){
4          // CREAZIONE PER ENUMERAZIONE DEGLI ELEMENTI
5          int[] arr1 = {
6              3,8,3,-2,5,8
7          };
8
9          for(int pos = 0 ; pos < 6;pos++){
10             System.out.println(Arr1[pos]);
11         }
12     }
13 }
```

8.1 Creazione array per enumerazione

A differenza del linguaggio C in JAVA quando dichiariamo un array nella zona di memoria dedicata si formano due celle in piu dove una ne indica la lunghezza (lenght) e una ne indica il tipo in questo caso (int[]).Vediamolo graficamente:



Aggiorniamo il nostro codice in questo modo :

```java
public class MainArray{

    public static void main(String[] args){
        // CREAZIONE PER ENUMERAZIONE DEGLI ELEMENTI
        int[] arr1 = {
            3,8,3,-2,5,8
        };

        for(int pos = 0 ; pos < arr1.length;pos++){
            System.out.println(Arr1[pos]);
        }
    }
}
```

8.2 Creazione array manualmente

Vediamo il seguente codice :

```java
public class MainArray{

    public static void main(String[] args){
        int [] arr1 = new int[6];
        arr[0] = 3;
        arr[1] = 8;
        arr[2] = 3;
        arr[3] = -2;
        arr[4] = 5;
        arr[5] = 8;
        for(int pos = 0 ; pos < arr1.length;pos++){
            System.out.println(Arr1[pos]);
        }
    }
}
```

Ottengo lo stesso risultato del primo array,sto creando un puntatore a un array di 6 interi.//

8.3 Ciclo foreach

In Java è stato introdotto il seguente metodo di stampa per gli array :

```java
public class MainArray{

    public static void main(String[] args){
        int [] arr1 = new int[6];
        arr[0] = 3;
        arr[1] = 8;
        arr[2] = 3;
        arr[3] = -2;
```

```
        arr[4] = 5;
        arr[5] = 8;
        for(int x : arr1){
            System.out.println(x + " ")
        }
    }
}
```

NB. Anche se stai vedendo Java, ricorda che in PHP i due punti possono rappresentare "appartenenza" in un ciclo foreach. Nel nostro caso, la variabile x è la chiave o l'indice, mentre arr1 è l'array. Tuttavia, in Java, come in PHP, ha senso usare un ciclo foreach solo per leggere.

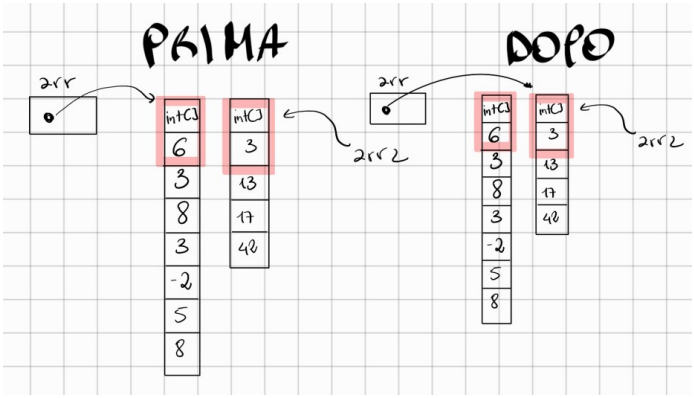
8.4 Assegnamento tra Array

In C l'assegnamento tra array non funziona invece in Java si.Ad esempio:

```
public class MainArray{

    public static void main(String[] args){
        int [] arr1 = new int[6];
        arr[0] = 3;
        arr[1] = 8;
        arr[2] = 3;
        arr[3] = -2;
        arr[4] = 5;
        arr[5] = 8;
        int [] arr2 = new int[3];
        arr2[0] = 13;
        arr2[1] = 17;
        arr2[2] = 42;
        for(int x : arr1){
            System.out.println(x + " ")
        }
    }
}
```

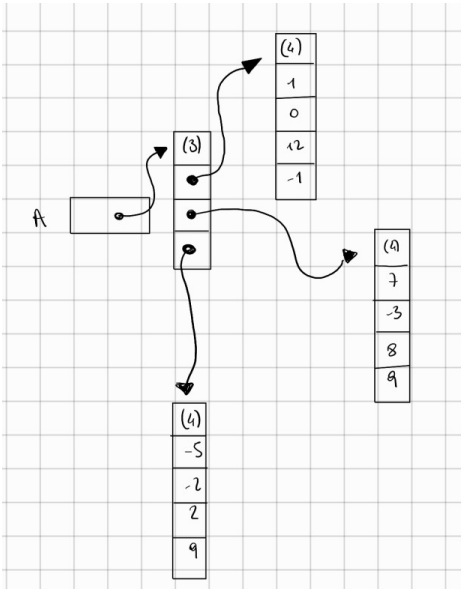
Per capire cosa succede lo rappresentiamo meglio con un disegno:



In pratica durante l'assegnamento viene cambiato l'oggetto puntato però per fare un assegnamento del genere c'è bisogno che i due tipi siano identici.

8.5 Matrici

una matrice è un array bidimensionale, usato per memorizzare dati in forma di righe e colonne. Puoi immaginarlo come una tabella, dove ogni elemento è accessibile tramite due indici: uno per la riga e uno per la colonna.



In C sappiamo che non esiste una propria matrice ma è un unico e grande array e con le formulette adatte ti riesci a ricavare la cellad della matrice invece in Java si rappresenta in una serie di puntatori.In pratica se creiamo un array A = new int[3][4] stiamo dicendo che A ha un puntatore a un array di dimensione

3 (i) i quali ogni cella puntera altri array di dimensione 4(j). L'unico vantaggio di questa modalità è la sua **flessibilità**. Per esempio:

```
public class MainArray{

    public static void main(String[] args){
        int [][] arr = {
            {1,5,-2,5},
            {4,8,8,0},
            {4,7,11,-1}
        };
        arr[2][1] = 99; // SI PUO FARE
        arr[1] = new int[4]; // nella riga uno sto creando
            un nuovo array di interi inizializzati di default
             a zero solo per quella riga li
        arr[1] = new int[5]; // in riga 1 viene inserito un
            array da 5 --> chiamati jagged array

        for(int i = 0;i<arr.lenght;i++){
            for(int j = 0 ;j<arr[i].lenght; j++){
                Sysrem.out.printf("%3d",arr[i][j]);
            }
        }

    }
}
```

In Java ci possono essere array multidimensionali di **qualsiasi dimensione**.

8.6 Esempio

Creaiamo inizialmente un enum delle stagioni in questo modo(prima creare il file Seasion.java):

```
public enum Season{
    SPRING,
    SUMMER,
    AUTUMN,
    WINTER
}
```

Ora creaiamo il nostro metodo dentro la classe di Date:

```
public class Date {
    // Attributi
    private int day;
    private int month;
    private int year;

    // costruttore
    public Date(int d,int m,int y){
```

```
        this.day = d;
        this.month = m;
        this.year = y;
    }

    //metodi
    Season getSeason(){
        Date startOfSpring = new Date(21,3,year);
        Date startOfSummer = new Date(21,6,year);
        Date startOfAutumn = new Date(22,9,year);
        Date startOfWinter = new Date(21,12,year);

        if (this.compareTo(startOfSpring)>=0 && this.
            compareTo(startOfSummer)<0){
            return Season.SPRING;
        }else if(this.compareTo(startOfSummer)>=0 && this.
            compareTo(startOfAutumn)<0){
            return Season.SUMMER;
        }else if(this.compareTo(startOfAutumn)>=0 && this.
            compareTo(startOfWinter)<0)){
            return Season.AUTUMN;
        }else{
            return Season.WINTER;
        }


    }
}
```

Creaiamo un nuovo main :

```
import java.util.Random;
public class MainDate {
    public static void main(String[] args){
        Date[] dates = new Date[100];
        Random random = new Random();
        for(int pos = 0 ; pos<dates.lenght;pos++){
            dates[pos] = new Date(random.nextInt(31)+1,
                random.nextInt(11)+1,random.nextInt(40)+1990)
                ;
        }
        for(Date date : dates){
            System.out.println(date + " "+date.getSeason());
        }

        //contare quante ce ne stanno per ogni stagione
        int [] counters = new int [4];
        for(Date date : dates){
            counters[date.getSeason().ordinal()]++;
        }
```

```
18
19          for(int key:counters){
20              System.out.prinln(key);
21          }
22      }
23  }
```

8.7 Metodi di uso frequente della classe java.util.Arrays

- **static int binarySearch(int[] arr, int key)** (ritorna la posizione di key dentro arr, oppure un numero negativo se arr non contiene key. Assume che l'array arr sia ordinato. Questo metodo esiste anche per gli altri tipi primitivi numerici e per i tipi riferimento, nel qual caso chiama compareTo() per decidere l'ordine)

- **static boolean equals(int[] arr1, int[] arr2)** (controlla che arr1 e arr2 abbiano stessa lunghezza e contengano gli stessi elementi nello stesso ordine. Questo metodo esiste anche per gli altri tipi primitivi nonché per array di tipi riferimento, nel qual caso chiama equals() fra tutte le coppie di oggetti da confrontare)

- **static void fill(int[] arr, int val)** (assegna val a tutti gli elementi di arr. Questo metodo esiste anche per tutti gli altri tipi primitivi e per array di tipi riferimento)

- **static void sort(int[] arr)** (ordina arr in senso crescente, in tempo O(n log n). Questo metodo esiste anche per tutti gli altri tipi primitivi numerici e per i tipi riferimento, nel qual caso chiama compareTo() per decidere l'ordine)

- **static String toString(int[] arr)** (ritorna una stringa che riporta gli elementi di arr, nel loro ordine. Questo metodo esiste anche per gli altri tipi primitivi e per array di tipi riferimento, nel qual caso chiama toString() sugli elementi dell'array e concatena il risultato.

9 Costruttore più in specifico

Come abbiamo visto nei capitoli seguenti il costruttore ha lo stesso nome della classe,non ha tipi di ritorno e nelle parentesi graffe vanno gli attributi che servono per inizializzare l'oggetto. Per richiamare il costruttore si fa nel main in questo modo : **new nomeOggetto(campi richiesti)** .

Importante

Se la nostra classe non ha un costruttore il compilatore aggiunge un **costruttore di default**.Il costruttore di default è il seguente:

```
1 public class C {
2     public C(){}
3 }
```

Come si può notare esso non fa nulla.

9.1 Overloading del Costruttore

L'overloading del costruttore si verifica quando una classe presenta più costruttori, ciascuno con una firma diversa, consentendo così di creare oggetti in modi differenti a seconda dei parametri forniti.
Per esempio:

```
1  public class Date(){
2      \\Attributi
3      private final int day;
4      private final int month;
5      private final int year;
6      private static Language language = Language.ITALIAN;
7
8      \\costruttori
9      public Date(int day,int month,int year){
10         this.day = day;
11         this.month = month;
12         this.year = year;
13     }
14     \\year implicitamente 2021
15     public Date(int day,int month){
16         this.day = day;
17         this.month = month;
18         this.year = 2021;
19     }
20
21 }
```

Il compilatore decide automaticamente quale costruttore utilizzare in base agli argomenti forniti. Se nel metodo main si forniscono due argomenti, il costruttore predefinito verrà invocato, e l'anno verrà inizializzato automaticamente al valore predefinito di 2021. D'altra parte, se vengono forniti tre argomenti, verrà utilizzato il primo costruttore, che accetta un nome, un mese e un giorno, consentendo di specificare maggiori dettagli sull'oggetto creato.

9.2 Concatenazione dei Costruttori

Quando facciamo un overloading di costruttori potrebbe sembrare che il codice sia tutto uguale per esempio:

```
public class Date(){
    \\Attributi
    private final int day;
    private final int month;
    private final int year;
    private static Language language = Language.ITALIAN;

    \\costruttori
    public Date(int day,int month,int year){
        this.day = day;
        this.month = month;
        this.year = year;
    }
    \\year implicitamente 2021
    public Date(int day,int month){
        this.day = day;
        this.month = month;
        this.year = 2021;
    }
    public Date(int day){
        this.day = day;
        this.month = 10;
        this.year = 2021;
    }
    public Date(){
        Random random = new Random();
        this.day = random.nextInt(31) + 1;
        this.month = random.nextInt(12) + 1;
        this.year = random.nextInt(40) + 1990
    }

}
```

Possiamo ottimizzare questo codice in questo modo :

```
public class Date(){
    \\Attributi
    private final int day;
```

```
    private final int month;
    private final int year;
    private static Language language = Language.ITALIAN;

    \\costruttori
    public Date(int day,int month,int year){
        this.day = day;
        this.month = month;
        this.year = year;
    }
    \\year implicitamente 2021
    public Date(int day,int month){
        this(day,month,2021);
        \\chiama il costruttore di questo oggetto
    }
    public Date(int day){
        this(day,10,2021);
    }
    public Date(){
        Random random = new Random();
        this.day = random.nextInt(31) + 1;
        this.month = random.nextInt(12) + 1;
        this.year = random.nextInt(40) + 1990
    }

}
```

Quando scrivo **this(day,10,2021)** sto dicendo di chiamare il costruttore di questo oggetto in questo modo. Inoltre se premo F3 su **this** mi porta sul primo costruttore.

Vincolo

this(bla,bla,bla) DEVE ESSERE LA PRIMA ISTRUZIONE DEL COSTRUTTORE.
Infatti nel terzo caso **NON** possiamo usare la notazione this perchè verrebbe utilizzata come seconda istruzione.

L'unico modo per ottimizzare al massimo sarebbe mettere il Random fuori in questo modo :

```
public class Date(){
    \\Attributi
    private final int day;
    private final int month;
    private final int year;
    private static Language language = Language.ITALIAN;

    \\costruttori
    public Date(int day,int month,int year){
```

```
10          this.day = day;
11          this.month = month;
12          this.year = year;
13      }
14      \\year implicitamente 2021
15      public Date(int day,int month){
16          this(day,month,2021);
17          \\chiama il costruttore di questo oggetto
18      }
19      public Date(int day){
20          this(day,10);
21      }
22      private final static Random random = new Random();
23      public Date(){
24          this(random.nextInt(31)+1,random.nextInt(12)+1,
                random.nextInt(40)+1990);
25      }
26
27  }
```

9.3 Inserimento dei vincoli del costruttore

Possiamo notare che se inizializziamo un oggetto Date possiamo anche inserire mesi negativi per risolvere questo problema dobbiamo inserire dei **vincoli**.

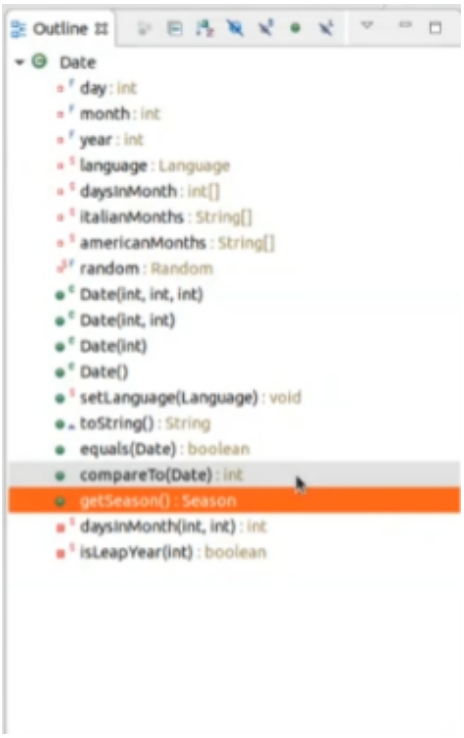
```
1   public class Date(){
2       \\Attributi
3       private final int day;
4       private final int month;
5       private final int year;
6       private static Language language = Language.ITALIAN;
7
8       \\costruttori
9       public Date(int day,int month,int year){
10          this.day = day;
11          this.month = month;
12          this.year = year;
13
14          if(month<1||month>12||day <1 || day > daysInMonth(
                month,year)|| year < 1600){
15              System.out.ptinln("Data inesistente");
16          }
17
18      }
19
20      private static int daysinMonth[]={
21      31,28,31,30,31,30,31,31,30,31,30,31
22      };
23      private static int daysInMonth(int month,int year){
```

```
24      if(month == 2 && isLeapYear(year)){
25          return 29;
26      }else{
27          return daysInMonth[month-1];
28      }
29      }
30      private static boolean isLeapYear(int year){
31          return year % 4 ==0 && (year % 100 != 0 || year %
                400 == 0)
32      }
33  }
```

Visto che ho i costruttori concatenati basta che si inserisca il vincolo solo sul primo costruttore.
Dal momento che isLeapYear() e daysInMonth() non usano this e assomigliano molto a delle funzioni piuttosto che a dei metodi sarebbe meglio metterci static per renderle un attimo piu efficienti.

9.4 Interfaccia pubblica della classe

Sono tutte le cose pubbliche definite dalla classe.Su Eclipse se andiamo nella sezione Outline vediamo questo :



I pallini verdi raffigurano tutti i punti public invece i pallini rossi tutti i punti private. In linea di massima nel public si mette il meno possibile.

9.5 Stato dell'oggetto

Lo stato dell'oggetto è definito dall'insieme dei campi(attributi) che fanno parte dell'oggetto (non sono static).

9.6 Oggetti mutabili e immutabili

Se degli attributi hanno **final** vuol dire che sono inizializzabili solo dal costruttore,dal momento che si usa il final lo stato viene chiamato **immutabile** ovvero che non ce modo modificare questi oggetti. Quindi esistono anche oggetti **mutabili** ovvero coloro che possono modificare lo stato dell'oggetto.
Esempio immutabile :

```
public class Date(){
    \\Attributi
    private final int day;
    private final int month;
    private final int year;
}
```

Esempio mutabile :

```
public class MutableDate(){
    \\Attributi
    private  int day;
    private  int month;
    private  int year;

    \\metodi
    public void increase(){
        day++;
        if(day>daysInMonth(mnth,year)){
            day = 1;
            month++;
            if(month == 13){
                month = 1;
                year ++;
            }
        }
    }
}
```

9.7 Aliasing/Side effect

Succede quando lo stesso oggetto può essere raggiunto da percorsi diversi,se nel percorso modifico l'oggetto lo modifico anche per l'altro percorso. Per esempio se facciamo questo in un nuovo Main:

```
public class MainAlias{
    public static void main(String[] args){
        MutableDate today = new MutableDate(19,10,2021);
        MutableDate tommorow = today;
        tommorow.increase();

        System.out.ptinln("today is " + today);
        System.out.println("tommorow is " + tommorow);
    }
}
```

Esso stamperà in out la stessa data visto che hanno un percorso collegato.

9.8 Commenti

Di solito quando si programma a oggetti bisogna commentare solo la parte public.

9.8.1 Commenti JavaDoc

I commenti JavaDoc si possono mettere dei tag che specificano l'informazione semantica del commento per esempio :

```
public class Date(){
    /**
    *Costruisce una data del calendario,
    *verificando che sia legale.
    *@param day giorno della data
    *@param month mese della data
    *@param year anno della data
    */
    private final int day;
    private final int month;
    private final int year;
    /**
    * Per i metodi ci sono i seguenti tag
    *   @param
    *   @return
    */
}
```

Il vantaggio di scrivere con questo standard e che ci sono dei tool che eseguendo automaticamente questi commenti. Puoi anche Generare il JavaDoc se utilizzi il menu in alto,verrà generata una cartella con diversi file.
Questo JavaDoc crea il tuo sito internet con la documentazione del tuo progetto.

9.9 Cosa succede se metto due file in package diversi?

Facendo cosi potremmo notare che si formerà un errore di compilazione, possiamo notare che non ci trova proprio la classe. L'unico modo per usare due file in due diversi package è usare **import nomepackage.nomeclasse;**

10 Ereditarietà

Come abbiamo visto nelle lezioni precedenti l'ereditarietà è una caratteristica fondamentale della programmazione ad oggetti.L'Ereditarietà 'e un principio nel quale una classe definita come sottoclasse eredita attributi e metodi fa un'altra classe detta superclasse,cosi da poter riutilizzare il codice e crea una sorta di gerarchia tra le classi.

10.1 Creazione del figlio

Partiamo da una classe Note.java :

```
1  public class Note {
2      private final int semitone;
3
4      public Note(int semitone){
5          this.semitone = semitone;
6      }
7
8      public String toString(){
9          System.out.println("nota di semitono" + semitone);
10     }
11
12 }
```

Creiamo ora una nuova classe ItalianNote:

```
1  public class ItalianNote extends Note{
2
3  }
```

Usare l'operatore **extends** significa che stiamo ereditando da Note.L'ereditarietà permette di riutilizzare il codice di una classe base all'interno di una classe derivata, senza dover riscrivere manualmente il codice (si ottengono attributi e metodi senza ereditare il costruttore) quindi :

```
1  public class ItalianNote extends Note{
2      public ItalianNote(int semitone){
3          this.semitone = semitone
4      }
5  }
```

Ma perchè non funziona?

10.2 Funzione super

Come possiamo vedere nella nostra classe Note, abbiamo utilizzato l'attributo **private**, quindi non possiamo accedervi direttamente a causa del principio dell'incapsulamento. Allora l'unico modo è quello di delegare il costruttore in questo modo:

```java
public class ItalianNote extends Note{
    public ItalianNote(int semitone){
        super(semitone);
    }
}
```

super fa una delega al costruttore della superclasse(la classe che sta sopra).

10.3 superclasse e sottoclasse

Una **superclasse** (o classe base) è una classe che fornisce attributi e metodi che possono essere ereditati da altre classi. In sostanza, rappresenta la classe generica da cui altre classi possono derivare. Una **sottoclasse** (o classe derivata) è una classe che eredita attributi e metodi da una superclasse, ma può anche aggiungere o sovrascrivere funzionalità per specializzarsi o comportarsi in modo diverso.
In questo caso la sottoclasse è ItalianNote e la superclasse è Note.

10.4 Funzione Get e uso di protected

Get serve per accedere indirettamente a un attributo private

```java
public class Note {
    private final int semitone;

    public Note(int semitone){
        this.semitone = semitone;
    }

    public String toString(){
        System.out.println("nota di semitono" + semitone);
    }

    protected int getSemitone(){
        return semitone;
    }

}
```

Se utilizzo **protected** sto dicendo che posso usare quei metodi anche nelle sue sottoclassi(si fa cosi perchè non va bene creare molti metodi public). Inoltre in Java posso rimpiazzare i metodi delle funzioni in questo modo:

```java
public class ItalianNote extends Note{

    public ItalianNote(int semitone){
        super(semitone);
    }

```

```
7       private final static String[] notes = {"DO","DO#","RE","
            RE#","MI","MI#","FA","FA#","SOL","SOL#","LA","LA#","
            SI","SI#"};
8
9       public String toString(){
10          return notes[semitone];
11      }
12  }
```

10.5 Relazione di sottotipo o sostituzione di Liskov

se un tipo B è un sottotipo di A, significa che ogni oggetto di tipo B può essere utilizzato al posto di un oggetto di tipo A senza rompere il codice (principio di sostituzione di **Liskov**). Questo implica che:

Il sottotipo B eredita tutte le caratteristiche (attributi e metodi) del supertipo A. Il sottotipo B può aggiungere nuove funzionalità o comportarsi in modo più specifico, ma senza violare l'interfaccia del supertipo. Esempio classico: Se "Veicolo" è un supertipo e "Auto" è un sottotipo, un'auto è sempre un veicolo, quindi possiamo trattarla come tale. Quindi nel main posso scrivere così:

```
1   public class Main{
2       public static void main(String[] args){
3           Note n1;
4           n1 = new Note(3);
5           Note n1;
6           n2 = new Note(3);
7           ItalianNote n1;
8           n1 = new ItalianNote(8);
9           //posso fare:
10          Note n4;
11          n4 = new ItalianNote(7);
12      }
13  }
```

10.6 Tipo statico e Tipo Dinamico

Il **tipo statico** di una variabile è il tipo che viene determinato durante la fase di compilazione. In altre parole, il tipo della variabile viene dichiarato esplicitamente nel codice e non cambia durante l'esecuzione Il **tipo dinamico** di una variabile è il tipo che viene determinato durante l'esecuzione del programma, cioè il tipo effettivo dell'oggetto a cui la variabile fa riferimento al runtime.

10.7 Legame ritardato

Il legame ritardato (o late binding), noto anche come dynamic dispatch, è un concetto della programmazione orientata agli oggetti in cui la decisione su quale metodo chiamare avviene durante l'esecuzione del programma, e non durante la

compilazione. Quando un oggetto di una sottoclasse viene trattato come un'istanza della sua superclasse, il metodo che verrà eseguito (quello della superclasse o della sottoclasse) viene determinato al runtime in base al tipo effettivo dell'oggetto.

10.8 Class Tag

La class tag è un termine usato per descrivere una situazione in cui una classe in una gerarchia di ereditarietà esiste principalmente come un identificatore o segnaposto, piuttosto che per definire una nuova funzionalità o comportamento. In altre parole, è una classe che viene utilizzata per "etichettare" oggetti, in modo da poterli trattare in modo differente in base alla loro appartenenza a questa classe, ma senza aggiungere nuove funzionalità o attributi. L'aspetto negativo è che avviene uno spreco di memoria quindi ogni oggetto spreca memoria col suo class tag.

10.9 Java.lang.object

In Java tutte le superclassi di default sono estese da object essa è l'unica superclasse che non è estesa da nulla. Inoltre otteniamo anche i suoi metodi tipo :

- equals() : controlla se due oggetti sono uguali ma molto raramente è utile conviene crearsi il proprio.
- ToString() : ritorna nome classe + indirizzo di memoria

10.10 Chiamata a metodi ereditario

Importante

Una classe può chiamare dentro se stessa i metodi della funzione della sua superclasse.

sfruttando la funzione **super()** in questo modo:

```
1  public class ItalianNoteWithDuration extends ItalianNote{
2      private final Duration duration;
3
4      public ItalianNoteWithDuration(int semitone,Duration
           duration){
5          super(semitone);
6          this.duration = duration;
7      }
8
9      public String toString(){
10         return super.toString() + " " + duration;
11     }
```

```
}
```

10.11 Operatori di Casting tra Oggetti

In Java, il casting è il processo di conversione di una variabile da un tipo a un altro, ed è fondamentale per gestire variabili di tipi diversi (si può attuare anche con delle variabili).

```
public class Main{
    public static void main(String[] args){
        ItalianNote n2 = new ItalianNoteWithDuration(8,
            Duration.MINIMA);

        System.out.println("n3 durata:" + ((
            ItalianNoteWithDuration) n3).getDuration();
    }
}
```

10.12 istanceof()

istanceof() serve per controllare il tipo dinamico di una variabile.

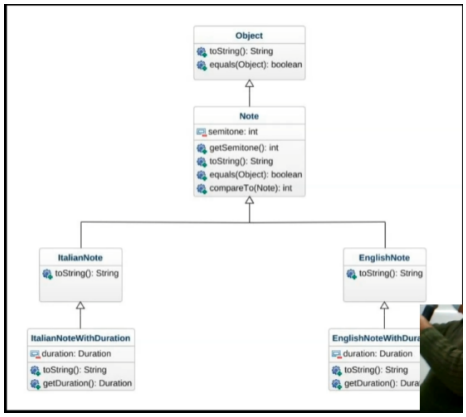
```
public class Note{
    .
    .
    .
    .
    .
    .
    public boolean equals(Object other){
        if(other e Note ){
            return semitone == ((Note) other).semitone;
        }else{
            return false;
        }
    }
    .
    .
    .
    public boolean equals(Object other){
        if(other istanceof Note){
            return semitone == ((Note) other).semitone;
        }else{
            return false;
        }
    }
}
```

Utile perchè molte librerie usano object quindi bisogna ridefinire con equals di object

11 Astrazione

11.1 UML

UML (Unified Modeling Language) è un linguaggio di modellazione standardizzato utilizzato per specificare, visualizzare, progettare e documentare i sistemi software. Non è un linguaggio di programmazione, ma una notazione che fornisce un insieme di simboli e diagrammi per rappresentare in modo grafico vari aspetti di un sistema.



11.2 Cos'è una Interfaccia

In Java visto che non esiste l'ereditarietà multipla si utilizza quello che viene detto Interfaccia ovvero una classe che è una classe senza codice ma con **SOLAMENTE** le dichiarazioni.

```
1 public interface NoteWithDuration(){
2     public Duration getDuration();
3 }
```

In pratica impongo a ai miei figli il mio metodo. Inoltre bisogna cambiare il codice nel figlio in questo modo:

```
1 public class ItaliannoteWithDuration extends ItalianNote
      implements NoteWithDuration{
2 .
3 .
4 .
5 .
6 .
7 .
8 }
```

11.3 Metodo astratto

Un metodo astratto è un metodo senza la sua implementazione ovvero un metodo non ancora concretizzato. Nelle classi si può creare un miscuglio di metodi concretizzati e metodi astratti usando la clausola **abstract** in qusto modo:

```
public abstract class ItaliannoteWithDuration extends
    ItalianNote implements NoteWithDuration{
.
.
.
public abstract String toString();
.
.
}
```

Inoltre se aggiungiamo il metodo abstract anche la classe sara abstract tipo il COVID(ne basta uno ammalato e tutti si possono ammalare).

Importante

NB. se dichiaro una classe astratta essa non puo piu essere usata nel Main

11.4 Classe Generica

Una classe generica in Java è una classe che può operare con tipi diversi senza dover essere riscritta per ciascun tipo. La genericità permette di scrivere codice più riutilizzabile e sicuro, riducendo il rischio di errori in fase di esecuzione (runtime), come i cast impropri. Con le generics, puoi definire parametri di tipo per classi, interfacce e metodi.

```
public class Pair<F, S> {
    private F first;  // Primo elemento della coppia
    private S second; // Secondo elemento della coppia

    // Costruttore
    public Pair(F first, S second) {
        this.first = first;
        this.second = second;
    }

    // Getter per il primo elemento
    public F getFirst() {
        return first;
    }

    // Getter per il secondo elemento
    public S getSecond() {
        return second;
```

```
19      }
20
21  }
```

Inoltre F e S sono dei tipi generici e sono **SEMPRE** scritti in maiuscolo.

11.5 Come si usano le classi generiche

```
1  public class Main{
2      public static void main(String[] args){
3      //volgio creare una coppia di due stringhe:
4          Pair<String,String> p1 = new Pair<String,String>("
               ciao","ciao");
5      //volgio creare una copia di una classe creata da me:
6      Pair<Figure,Figure> p2 = new Pair <Figure,Figure>(f1,f2)
           ;
7      }
8  }
```

11.6 java.lang.Comparable $< T >$

All'interno di questa classe ce solo un metodo astratto che si chiama **int compareTo(T other)**: ritorna un numero negativo se viene prima this,un numero positvo se viene prima other e 0 se this e other si equivalgono.

```
1  public abstract class Note implements Comparable<Note>{
2  .
3  .
4  .
5  .
6  }
```

11.7 Chiamata a metodo di un tipo generico

Il tipo generico può venir chiamate all'interno del metodo per esempio in questo modo :
Voglio fare un sort di un array:

```
1      public class Utils {
2          public static <T extends Comparable<T>> void sort(T
               [] arr){
3          // bubblesort
4              while(swap(arr));
5          }
6
7          private static <T extends Comparable<T>> boolean
               swap(T[] arr){
```

```
8               boolean done = false;
9
10              for(int pos = 0 ;pos<arr.lenght-1;pos++){
11                  if(arr[pos].compareTo(arr[pos+1]) > 0){
12                      T temp = arr[pos];
13                      arr[pos] = arr[pos + 1];
14                      arr[pos+1] = temp;
15                      done = true;
16                  }
17              }
18              return done;
19          }
20      }
```

Ora il tipo generico è **locale al metodo** (array di tipo generico mele,pere,banane,int...), invece con **extends Comparable¡T¿** lo suo per vincolare T ovvero li sto dicendo di comparare un oggetto uguale a lui. Da un errore se viene passato una array di interi perchè gli ¡T¿ hanno un limite ovvero che possono essere passati solo per riferimento. In java i tipi generici **NON** possono essere passati i tipi primitivi

11.8 Tipi di avvolgimento o Classi Wrapper

Sono tipi di riferimento che identificano i tipi primitivi così da ovviare ai problemi dei tipi generici è sono : Byte,Short,Long,Integer,Float,Double,Character,Boolean. Quindi:

```
1       Integer[]arr4 = {
2           Integer.valueOf(13);
3           Integer.valueOf(14);
4           Integer.valueOf(58);
5       }
6
7       Integere i0 = 78; //Come dire Integer i0.valueof(78);
            boxing automatico
```

11.9 Metodi di uso frequente della classe java.lang.Integer

- static int MAX_VALUE (costante che contiene il massimo int utilizzabile in Java)
- static int MIN_VALUE (costante che contiene il minimo int utilizzabile in Java)
- Integer(int value) (deprecato!)
- Integer(String value) throws java.lang.NumberFormatException
- int intValue() (restituisce il valore int corrispondente)

- int compareTo(Integer other) (infatti Integer implementa Comparable¡Integer¿)
- static int parseInt(String s) throws java.lang.NumberFormatException (traduce la stringa s in int)
- static String toBinaryString(int i) (ritorna la rappresentazione binaria di i)
- static String toHexString(int i) (ritorna la rappresentazione esadecimale di i)
- static Integer valueOf(int i) (ritorna new Integer(i) ma usa una cache per chiamate ripetute)

Esistono altre classi wrapper corrispondenti agli altri tipi primitivi, con costanti, costruttori e metodi simili a quanto riportato sopra: java.lang.Short, java.lang.Long, java.lang.Float, java.lang.Double,java.lang.Byte e java.lang.Boolean.

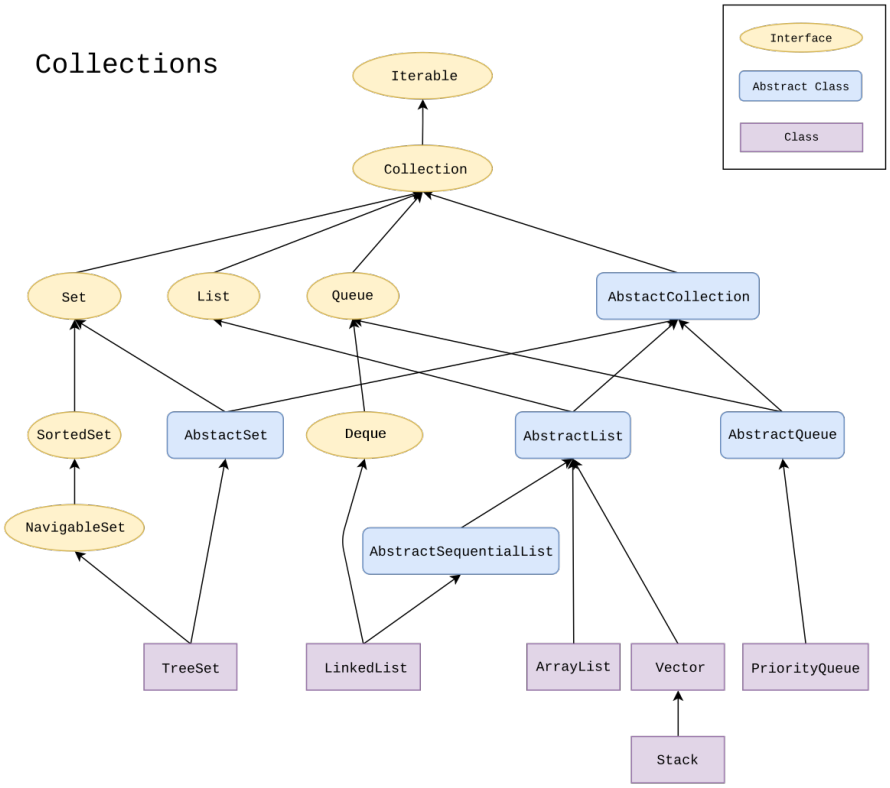
11.10 Metodi di uso frequente della classe java.lang.Character

- static char MAX_VALUE (costante che contiene il massimo char utilizzabile in Java)
- static char MIN_VALUE (costante che contiene il minimo char utilizzabile in Java)
- Character(char value) (deprecato!)
- char charValue() (restituisce il valore char corrispondente)
- int compareTo(Character other) (infatti Character implementa Comparable¡Character¿)
- static boolean isDigit(char c)
- static boolean isLetter(char c)
- static boolean isLetterOrDigit(char c)
- static boolean isLowerCase(char c)
- static boolean isUpperCase(char c)
- static boolean isWhitespace(char c)
- static char toLowerCase(char c)
- static char toUpperCase(char c)
- static Character valueOf(char c) (ritorna new Character(c) ma usa una cache per chiamate ripetute)

12 Collezioni

Abbiamo imparato ad usare strutture ed array nei precedenti capitoli. In java esistono le collezioni che sono dei contenitori più moderni che contengono moltissimi metodi.

12.1 Gerarchia delle collezioni



In giallo abbiamo le interfaccie in blu le classi astratte invece in violetto le vere e proprie classi che andremmo ad usare. Tutto questa sta dentro java.util.Colletions.

- Interfaccia **Collection** è qualsiasi collezione
- Interfaccia **List** una sequenza di valori che a seconda delle neccesità si possono allargare o restringere
- Interfaccia **Queue** sono le code che hanno la carrateristica di ggiungere da un lato e tolgierlo dall'altro

- Interfaccia **Set** indica degli insiemi ma non hanno un ordine inoltre non ci possono essere due elementi uguali.
- Interfaccia **SortedSet** gli elementi in questo caso sono ordinati in ordine crescente
- Interfaccia **Iterable** possibilità di usare il foreach

12.2 Metodi di uso frequente dell'interfaccia java.util.Colletion< E >

- **boolean add(E element)** (ritorna true se l'elemento viene aggiunto)
- **boolean addAll(Collection**< E > **other)** (ritorna true se almeno un elemento viene aggiunto)
- **boolean contains(Object element)**
- **boolean containsAll(Collection**<? > **other)**
- **boolean isEmpty()**
- **boolean remove(Object element)** (ritorna true se l'elemento viene rimosso)
- **boolean removeAll(Collection**<? > **other)** (ritorna true se almeno un elemento viene rimosso)
- **boolean retainAll(Collection**<? > **other)** (ritorna true se almeno un elemento viene rimosso)
- **int size()**

12.3 Metodi di uso frequente dell'interfaccia java.util.List< E >

- **boolean add(E element)** (aggiunge element in fondo alla lista, anche se la lista già lo conteneva; ritorna sempre true)
- **void add(int index, E element)** (piazza l'elemento alla posizione index, che deve essere fra 0 e size() inclusi, spostando di una posizione a destra l'elemento che c'era precedentemente e quelli alla sua destra)
- **E get(int index)** (ritorna l'elemento alla posizione index, che deve essere fra 0 incluso e size() escluso)
- **int indexOf(Object element)** (ritorna la prima posizione in cui occorre element; ritorna -1 se la lista non contiene element)
- **static** < E >**List**< E >**of(E... elements)** (factory method che costruisce una lista immutabile con dentro elements)

- **boolean remove(Object element)** (rimuove la prima occorrenza di element, se presente; ritorna true se l'elemento viene rimosso)
- **E remove(int index)** (rimuove e ritorna l'elemento alla posizione index, che deve essere fra 0 incluso e size() escluso; gli elementi alla sua destra vengono spostati di una posizione a sinistra)
- **E set(int index, E element)** (ritorna l'elemento alla posizione index, che deve essere fra 0 e size() inclusi, e lo sostituisce con element)

12.4 Metodi di uso frequente dell'interfaccia java.util.Queue $< E >$

- **E poll()** (rimuove e ritorna la testa della coda; ritorna null se la coda è vuota)
- **E remove() throws java.util.NoSuchElementException** (rimuove e ritorna la testa della coda, se non è vuota; altrimenti lancia un'eccezione)
- **E peek()** (ritorna la testa della coda, senza rimuoverla; ritorna null se la coda è vuota)
- **E element() throws java.util.NoSuchElementException** (ritorna la testa della coda, senza rimuoverla; se la code è vuota, lancia un'eccezione)
- **boolean offer(E element)** (aggiunge element in fondo alla coda, se c'è spazio. Ritorna true se e solo se l'elemento viene aggiunto)
- **boolean add(E element) throws java.lang.IllegalStateException** (aggiunge element in fondo alla coda, se c'è spazio, altrimenti lancia un'eccezione. Ritorna sempre true)

12.5 java.util.Set$< E >$

- **static** $< E >$ **Set**$< E >$ of(E... elements) (factory method)

12.6 Metodi di uso frequente dell'interfaccia java.util.LinkedList$< E >$

- **LinkedList()**
- **LinkedList(Collection**$<?extends E >$ **parent)** (crea una lista e la riempie con gli elementi di parent)

12.7 Metodi di uso frequente dell'interfaccia java.util.ArrayList< $E >$

- **ArrayList()**
- **ArrayList(Collection**<?$extends E >$ **parent)** (crea una lista e la riempie con gli elementi di parent)

12.8 Metodi di uso frequente dell'interfaccia java.util.PriorityQueue< $E >$

- **PriorityQueue()**
- **PriorityQueue(Collection**<?$extends E >$ **parent)** (crea una coda e la riempie con gli elementi di parent)

12.9 Esempio Pratico:

Creiamo una ArrayList di Stringhe:

```
1  import java.util.ArrayList
2  public class Main{
3      public static void main(String[] args){
4          ArrayList<String> l = new ArrayList<String>();
5          //Aggiungiamo un elemento
6          l.add("ciao");
7          l.add("Hello");
8          l.add("Ciao");
9          l.add(0,"buongiorno");
10         //output: buongionrno ciao Hello Ciao
11         //rimuoviamo un valore viene eliminata solo la prima
               ricorenza
12         l.remove("ciao");
13         //la lunghezza:
14         l.size();
15         for(String s: l){
16             System.out.printl(l);
17         }
18     }
19 }
```

12.10 Differenza tra una LinkedList e un ArrayList

Quando si utilizza una LinkedList al posto di una ArrayList, nel codice non cambia nulla a livello funzionale, ma le differenze principali riguardano l'implementazione e il comportamento delle due strutture.

Con una **ArrayList**, la gestione della memoria è basata su un array sottostante. Questo significa che la dimensione dell'ArrayList inizialmente è limitata,

ma quando l'array si riempie, la struttura procede automaticamente a raddoppiarne la dimensione. Questo processo, sebbene utile per gestire dinamicamente la crescita, può risultare costoso in termini di prestazioni, poiché implica la copia di tutti gli elementi nell'array di dimensione maggiore. Al contrario, una **LinkedList** è composta da nodi, dove ogni elemento è collegato al successivo attraverso un riferimento (puntatore). Non esiste una dimensione predefinita: la lista cresce man mano che vengono aggiunti nuovi elementi, collegandoli semplicemente alla fine o al punto desiderato della catena. La scelta tra le due dipende quindi dal tipo di operazioni più frequenti nel programma.

12.11 varArgs

varArgs permette di inserire una quantita variabile di elementi in questo modo:

```
    .
    .
    .
    public void add(Coin... all){
        coins.add(coin);
    }
    .
    .
    .
```

Nel main ora potro inserire una quantita variabile di Coin in questo modo:

```
    .
    .
    .
    PiggyBank pig = new PiggyBank();
    pig.add(new Coin(200),new Coin(500),new Coin(350));
    .
    .
    .
```

12.12 Le code

In programmazione, una coda (queue) è una struttura dati astratta utilizzata per gestire elementi in modo sequenziale, seguendo il principio **FIFO** (First In, First Out). Questo significa che il primo elemento inserito nella coda sarà il primo ad essere rimosso.Esistono in java in oltre le code con priorità ovvero il più piccolo elemento viene ritornato prima:

```
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Scanner;

public class MainQueue1{
    Queue<String> q = new PriorityQueue<String>;
```

```
    Scanner keyboard = new Scanner(System.in);
    while(true){
        String s = keyboard.nextLine();
        if("fine".equals(s)){
            break;
        }
        q.offer(s);
    }
    String s;
    while((s=q.poll())!=null){
        System.out.println(s);
    }
    keyboard.close();
}
```

L'output sarà ordinato alfabeticamente per stringhe. Ovviamente i numeri devono essere **Comparable** per venire ordinati.

12.13 Set

Sono collezioni di dati non ordinati ma univoci (non ci possono essere due dati simili)

12.13.1 TreeSet

```
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Scanner;

public class MainQueue1{
    Set<String> q = new TreeSet<String>;
    Scanner keyboard = new Scanner(System.in);
    while(true){
        String s = keyboard.nextLine();
        if("fine".equals(s)){
            break;
        }
        q.offer(s);
    }
    String s;
    while((s=q.poll())!=null){
        System.out.println(s);
    }
    keyboard.close();
}
```

Non si può iterare tra Collezioni, allora si crea una copia.

12.14 Mappa

Possiamo identificarlo come un array ma con un indice letterale.

```
1
2
3  public class MainMap{
4      public static void main(String[] args){
5          Map<String,String> m = new HashMap(String,String)();
6          m.put("casa","house");
7          m.put("cane","dog");
8          m.put("casa","home");
9          System.out.println("casa->"+m.get("casa"));
10     }
11 }
```

13 Laboratorio

13.1 Esercitazione 1

```java
/*
Si scriva un programma Java che legge un intero non negativo n da tastiera e stampa una cornice n x n:
@@@@@
@   @
@   @
@   @
@   @
@@@@@

*/
import java.util.Scanner;
public class Cornice {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    int n;

    do {
      n = keyboard.nextInt();
    }while(n<=0);

    keyboard.close();// e bene sempre chiudere un oggetto scanner
    for(int i = 0;i<n;i++) {
      for(int j = 0;j<n;j++) {
        if(i == 0 || i == (n-1)) {
          System.out.print("@");
        }else if(j == 0 || j == (n-1) ) {
          System.out.print("@");
        }else {
          System.out.print(" ");
        }
      }
      System.out.println();
    }
  }

}
```

```java
/*
 Si scriva un programma che legge n >= 1 da tastiera e stampa una piramide
  di altezza n. Per esempio, per n = 4 deve stampare:
   @
  @@
```

```
 @@@@
@@@@@@
 * */
import java.util.Scanner;

public class Piramide {
    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);


        System.out.print("Inserisci l'altezza della piramide
            : ");
        int n = scanner.nextInt();


        if (n >= 1) {
            for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n - i; j++) {
                    System.out.print(" ");
                }
                for (int k = 1; k <= 2 * i - 1; k++) {
                    System.out.print("@");
                }
                System.out.println();
            }
        } else {
            System.out.println("L'altezza deve essere almeno
                1.");
        }

        scanner.close();
    }
}
```

```
/*
 *
 * Si faccia la stessa cosa del punto precedente, ma
     stampando la piramide orizzontalmente:
   @
  @@
 @@
@@@
 @@
  @@
   @
 * */
import java.util.Scanner;

```

```java
public class PiramideOrr {

    public static void main(String[] args) {

        Scanner keyboard = new Scanner(System.in);
        int n;

        do {
            System.out.print("Inserisci un numero positivo
                per l'altezza della piramide: ");
            n = keyboard.nextInt();
        } while (n <= 0);

        for(int i=0;i<(n-1);i++) {
          for(int j=0;j<n;j++) {
            if((n-j)<=i) {
              System.out.print("@");
            }else {
              System.out.print(" ");
            }
          }
          System.out.println();
        }
        for(int i=0;i<n;i++) {
          for(int j=0;j<n;j++) {
            if(i<=j) {
              System.out.print("@");
            }else {
              System.out.print(" ");
            }
          }
          System.out.println();
        }
        // Chiude lo scanner
        keyboard.close();
    }
}
```

13.2 Esercitazione 2

```java
// esercizio sulla parola palindroma:
import java.util.Scanner;

public class Palindromo {

  public static void main(String[] args) {
    boolean palindrome = true;
    String stringa ;
```

```
    Scanner keyboard = new Scanner(System.in);
    stringa = keyboard.nextLine();
    for(int i=0;i<(stringa.length()/2);i++) {
      if(stringa.charAt(i) != stringa.charAt(stringa.length
          ()-i-1)){
        palindrome = false;
      }
    }
    keyboard.close();
    if(palindrome == true) {
      System.out.print("okay");
    }else {
      System.out.print("no okay");
    }
  }

}
```

```
// esercizio sulla conversione in esadecimale con questa
    soluzione puoi fare ogni tipo di conversione:
import java.util.Scanner;

public class Exa {
  public static void main(String[] args) {
    int n;
    Scanner keyboard = new Scanner(System.in);
    do {
      System.out.print("n: ");
      n = keyboard.nextInt(); // se negativo devo
          richiederlo
    }
    while (n < 0);

    // traduco n in esadecimale dentro result
    String result = "";
    String digits = "0123456789abcdef";
    do {
      result = digits.charAt(n % 16) + result;
      n /= 16;
    }
    while (n > 0);
    keyboard.close();
    System.out.println(result);
  }

}
```

```
// esercizio sulla conversione in binario --> un'altro modo
    ma SOLO per il binario
```

```java

import java.util.Scanner;

public class Binario {

  public static void main(String[] args) {
    int num;
    Scanner keyboard = new Scanner(System.in);
    String stringa = "";
    do{
      num = keyboard.nextInt();
    }while(num<0);

    do{

      stringa = num%2+stringa;
      num = num / 2;
    }while(num!=0);

    System.out.println(stringa);


    keyboard.close();
  }

}
```

13.3 Esercitazione 3

Si vuole implementare una carta del gioco del poker, il cui valore è uno fra questi:
2 3 4 5 6 7 8 9 10 J Q K 1

Tale valore può essere visto come un numero fra 0 e 12. Una carta ha anche un seme (in inglese: suit) che può essere:
spades , clubs , diamonds e hearts .

1. Si completi la seguente classe, che rappresenta una carta:

```java
public class Card {

  /**
   * Il valore della carta.
   */
  private final int value;

  /**
   * Il seme della carta.
   */
```

```
11    private final int suit;
12
13    /**
14     * Genera una carta a caso con un valore da min (
          incluso) in su.
15     *
16     * @param min il valore minimo (0-12) della carta che
          puo essere generata
17     */
18    public Card(int min) { ... }
19
20    /**
21     * Genera una carta a caso con un valore da 0 (incluso
          ) in su.
22     */
23    public Card() { ... }
24
25    public int getValue() { ...ritorna il valore della
          carta (0-12) }
26
27    public int getSuit() { ...ritorna il seme della carta
          (0-3) }
28
29    /**
30     * Ritorna una descrizione della carta sotto forma di
          stringa, del tipo 10 oppure J.
31     */
32    public String toString() { ... }
33
34    /**
35     * Determina se questa carta e uguale ad other.
36     *
37     * @param other l'altra carta con cui confrontarsi
38     * @return true se e solo se le due carte sono uguali
39     */
40    public boolean equals(Card other) { ... }
41 }
```

2. Si scriva una classe Main con un metodo iniziale main che crea una carta card1 a caso, quindi crea ripetutamente una carta card2 a caso finché non risulta che card1 è equals con card2. A quel punto termina. Sia card1 che tutte le card2 dovranno venire stampate sul video man mano che vengono generate.

3. Si definiscano due enumerazioni Value e Suit, che rappresentano, rispettivamente, le 13 alternative per il valore delle carte e le quattro alternative per il loro seme. Si modifichi la classe Card in modo da usare queste enumerazioni al posto degli interi come valore e seme delle carte.

4. Si aggiunga un metodo public int compareTo(Card other) alla classe Card, che confronta una carta con un'altra e determina chi viene prima: le carte devono venire ordinate per valore; a parità di valore, devono venire ordinate per seme (picche, fiori, quadri, cuori).

5. Si generino le pagine JavaDoc da Eclipse.

1. File Card.java:

```
1         package Cornice;
2 import java.util.Random;
3 public class Card {
4   //Attributi:
5   private final int value;
6   private final int suit;
7
8   private static String[] listacarte= {"2","3","4","5","
      6","7","8","9","10","J","Q","K","1"} ;
9   private static String[] listasemi= {"\u2660","\u2663",
      "\u2665","\u2666"};
10
11  private final static Random random = new Random();
12  /*
13   * Genera una carta a caso con un valore da min (
       incluso in su)
14   * @param min il valore minimo (0-12) della carta che
       puo essere generata
15   */
16  public Card(int min) {
17    this.value = min + random.nextInt(13-min);
18    this.suit = random.nextInt(4);
19  }
20  /*
21   * Genera una carta a caso con un valore da 0 (incluso
       ) a su
22   */
23  public Card() {
24    this(0);
25  }
26
27  // metodi:
28  public int getValue() {
29    return this.value;
30  }
31
32  public int getSuit() {
33    return this.suit;
34  }
35
36  public String toString() {
```

```
37      return listacarte[this.value] + listasemi[this.suit
            ];
38    }
39    /*
40     * Determina se questa carta e uguale a other
41     * @param other l'altra carta con cui confrontarsi
42     * @return true se e solo se le due carte sono uguali
43     */
44    public boolean equals(Card other) {
45      if(this.value == other.value && this.suit == other.
            suit) {
46        return true;
47      }else {
48        return false;
49      }
50
51    }
52  }
```

2. File MainCard.java:

```
1   package Cornice;
2
3   public class MainCard {
4
5     public static void main(String[] args) {
6       Card card1 = new Card();
7       Card card2;
8       System.out.println("CARTA 1 : "+card1);
9
10      do {
11        card2 = new Card();
12        System.out.println(card2);
13      }while(card1.equals(card2) == false);
14
15
16    }
17
18  }
```

3. Nuova classe :

```
1   package Cornice;
2   import java.util.Random;
3   public class Card {
4     //Attributi:
5     private final Value value;
6     private final Suit suit;
7
```

```
8   // Lista dei valori per generazione casuale:
9     private static final Value[] values = Value.values()
          ;
10    private static final Suit[] suits = Suit.values();
11
12    /*
13  private static String[] listacarte=
        {"2","3","4","5","6","7","8","9","10","J","Q","K
        ","1"} ;
14  private static String[] listasemi= {"\u2660","\u2663
        ","\u2665","\u2666"};*/
15
16  private final static Random random = new Random();
17  /*
18   * Genera una carta a caso con un valore da min (
         incluso in su)
19   * @param min il valore minimo (0-12) della carta che
         puo essere generata
20   */
21  public Card(int min) {
22    this.value = values[min + random.nextInt(13-min)];
23    this.suit = suits[random.nextInt(4)];
24  }
25  /*
26   * Genera una carta a caso con un valore da 0 (incluso
         ) a su
27   */
28  public Card() {
29    this(0);
30  }
31
32  // metodi:
33  public Value getValue() {
34    return this.value;
35  }
36
37  public Suit getSuit() {
38    return this.suit;
39  }
40
41  public String toString() {
42    return value.name() + suit.getSymbol();
43  }
44  /*
45   * Determina se questa carta e uguale a other
46   * @param other l'altra carta con cui confrontarsi
47   * @return true se e solo se le due carte sono uguali
48   */
49  public boolean equals(Card other) {
50    if(this.value == other.value && this.suit == other.
```

```
        suit) {
51        return true;
52      }else {
53        return false;
54      }
55
56    }
57  }
```

4. aggiungiamo la nuova classe :

```
1   package Cornice;
2   import java.util.Random;
3   public class Card {
4     //Attributi:
5     private final Value value;
6     private final Suit suit;
7
8     // Lista dei valori per generazione casuale:
9       private static final Value[] values = Value.values()
            ;
10      private static final Suit[] suits = Suit.values();
11
12      /*
13    private static String[] listacarte=
          {"2","3","4","5","6","7","8","9","10","J","Q","K
          ","1"} ;
14    private static String[] listasemi= {"\u2660","\u2663
          ","\u2665","\u2666"};*/
15
16    private final static Random random = new Random();
17    /*
18     * Genera una carta a caso con un valore da min (
           incluso in su)
19     * @param min il valore minimo (0-12) della carta che
           puo essere generata
20     */
21    public Card(int min) {
22      this.value = values[min + random.nextInt(13-min)];
23      this.suit = suits[random.nextInt(4)];
24    }
25    /*
26     * Genera una carta a caso con un valore da 0 (incluso
           ) a su
27     */
28    public Card() {
29      this(0);
30    }
31
32    // metodi:
```

```
33    public Value getValue() {
34      return this.value;
35    }
36
37    public Suit getSuit() {
38      return this.suit;
39    }
40
41    public String toString() {
42      return value.name() + suit.getSymbol();
43    }
44    /*
45     * Determina se questa carta e uguale a other
46     * @param other l'altra carta con cui confrontarsi
47     * @return true se e solo se le due carte sono uguali
48     */
49    public boolean equals(Card other) {
50      if(this.value == other.value && this.suit == other.
            suit) {
51        return true;
52      }else {
53        return false;
54      }
55
56    }
57
58    public int compareTo(Card other) {
59      int valore = this.value.compareTo(other.value) ;
60      if(valore == 0) {
61        return this.suit.compareTo(other.suit);
62      }else {
63        return valore;
64      }
65    }
66  }
```

13.4 Esercitazione 4

Si crei un package it.univr.figures. Al suo interno, si realizzi il codice descritto sotto, in cui tutti i campi devono essere private per rispettare l'incapsulazione.

- Si scriva un'enumerazione Color che enumera cinque colori di vostra scelta, incluso il verde.
- Si scriva una classe Figure che rappresenta una figura geometrica colorata. Tale classe deve avere un costruttore che riceve un Color. Deve avere un metodo double perimeter() e un metodo double area(); fate ritornare ad entrambi 0. Inoltre deve avere un metodo String toString() che ritorna la stringa "area: A, perimeter: P, color: C", dove A è l'area della figura, P è

il perimetro della figura e C è il colore della figura. Infine, deve avere un metodo accessore protected per il colore.

- Si scriva una sottoclasse Rectangle di Figure che rappresenta un rettangolo, con un costruttore che riceve colore, base e altezza double del rettangolo e in cui i metodi double perimeter() e double area() sono ridefiniti in modo da ritornare perimetro ed area del rettangolo, rispettivamente. Il metodo String toString() deve essere ridefinito in modo da ritornare la stringa "Rectangle of " seguita dalla chiamata al toString() della superclasse.

- Si scriva una sottoclasse Circle di Figure che rappresenta un cerchio, con un costruttore che riceve colore e raggio double del cerchio e in cui i metodi double perimeter() e double area() sono ridefiniti in modo da ritornare perimetro ed area del cerchio, rispettivamente. Il metodo String toString() deve essere ridefinito in modo da ritornare la stringa "Circle of " seguita dalla chiamata al toString() della superclasse.

- Si scriva una sottoclasse Square di Rectangle che rappresenta un quadrato, con un costruttore che riceve colore e lato double del quadrato. Non si ridefiniscano i metodi double perimeter() e double area(). Il metodo String toString() deve invece essere ridefinito in modo da ritornare la stringa "Square, a " seguita dalla chiamata al toString() della superclasse.

- Si scriva una sottoclasse GreenDot di Circle che rappresenta un cerchio di raggio 1 e colore verde, con un costruttore senza argomenti. Non si ridefinisca alcun metodo al suo interno.

- Si scriva una sottoclasse GreenDot di Circle che rappresenta un cerchio di raggio 1 e colore verde, con un costruttore senza argomenti. Non si ridefinisca alcun metodo al suo interno.

Si scriva dentro it.univr una classe MainFigures con un metodo di partenza main che crea e stampa sul video un'istanza di ognuna delle figure geometriche implementate sopra. In tale classe è possibile chiamare il metodo getColor() sulle figure?
RISOLUZIONE:

1. enumeriazione Color:

```
1 package it.univr.figures;
2
3 public enum Color {
4   VERDE,
5   ROSA,
6   BLU,
7   ROSSO,
8   NERO
9 }
```

2. Classe figure :

```
package it.univr.figures;

public class Figure {
  // private final float area;
  // private final float perimetro;
  private final Color colore;

  public Figure(Color colore) {
    this.colore = colore;
  }

  public double perimeter() {
    return 0.0;
  }

  public double area() {
    return 0.0;
  }

  public String toString() {
    return "area :"+area()+"perimeter :"+perimeter()+"
        color : "+colore;
  }

  protected Color getColor() {
    return colore;
  }

}
```

3. Classe Rectangle:

```
package it.univr.figures;

public class Rectangle extends Figure {
  private double base;
  private double altezza;

  public Rectangle(Color colore,double altezza,double
      base) {
    super(colore);
    this.altezza = altezza;
    this.base = base;
  }

  public double perimeter() {
    return base+base+altezza+altezza;
  }
```

```
16
17    public double area() {
18      return base*altezza;
19    }
20
21    public String toString() {
22      return "Rectangle of" + super.toString();
23    }
24
25  }
```

4. Classe Circle :

```
1  package it.univr.figures;
2
3  public class Circle extends Figure {
4    private double raggio;
5
6    public Circle(Color colore,double raggio) {
7      super(colore);
8      this.raggio = raggio;
9    }
10
11   public double perimeter() {
12     return 2*raggio*Math.PI;
13   }
14
15   public double area() {
16     return Math.PI * raggio * raggio;
17   }
18
19   public String toString() {
20     return "Circle of " + super.toString();
21   }
22 }
```

5. Classe Square :

```
1  package it.univr.figures;
2
3  public class Square extends Rectangle{
4    public Square(Color colore,double lato) {
5      super(colore,lato,lato);
6    }
7    public String toString() {
8      return "Square , a" + super.toString();
9    }
10 }
```

6. Classe Greendot:

```
            package it.univr.figures;

public class Square extends Rectangle{
  public Square(Color colore,double lato) {
    super(colore,lato,lato);
  }
  public String toString() {
    return "Square , a" + super.toString();
  }
}
```

7. MainFigures:

```
            package univr.it;
import it.univr.figures.*;

public class MainFigures {

  public static void main(String[] args) {
    Figure f0 = new Figure(Color.ROSSO);
    Figure f1 = new Circle(Color.VERDE,15.0);
    Figure f2 = new Rectangle(Color.ROSA,16.26,12.2);
    Figure f3 = new Square(Color.BLU,12);
    Figure f4 = new GreenDot();


    System.out.println(f0);
    System.out.println(f1);
    System.out.println(f2);
    System.out.println(f3);
    System.out.println(f4);

  }

}
```

13.5 Esercitazione 5

1. Si consideri la seguente interfaccia, che specifica un giocatore di calcio:

```
            public interface SoccerPlayer {
              String toString();  // ritorna il nome del
                   giocatore
              boolean canUseHands();  // determina se il
                   giocatore puo usare le mani
            }
```

Si completi la seguente implementazione di SoccerPlayer:

```
public abstract class AbstractSoccerPlayer
    implements SoccerPlayer {
  ...
protected AbstractSoccerPlayer(String name) {
    ...
}

public final String toString() {
    ...
  }
}
```

E' possibile aggiungere campi o metodi, ma solo private. Si noti che il metodo canUseHands() non è ancora implementato in AbstractSoccerPlayer, per cui tale classe deve essere abstract.

2. Si realizzino quattro sottoclassi concrete di AbstractSoccerPlayer, chiamate rispettivamente Forward, Midfield, Defence e GoalKeeper. Il costruttore di tali classi richiede il nome del giocatore come parametro. Solo il GoalKeeper può usare le mani nel gioco del calcio.

3. Si completi la seguente classe, che implementa una formazione del gioco del calcio, cioè l'insieme degli 11 giocatori che formano la squadra durante una partita.

```
public class Formation {
  ...
public Formation(SoccerPlayer[] players) {
    ...
    if (!isValid())
      throw new IllegalArgumentException("
          invalid formation");
}

protected boolean isValid() {
    // ritorna true se e solo se la
       formazione e fatta da 11 giocatori,
       di cui esattamente uno e un portiere
}

protected SoccerPlayer[] getPlayers() {
    // ritorna i giocatori di questa
       formazione
}

public final String toString() {
```

```
18                  // ritorna i nomi dei giocatori della
                        formazione, separati da virgola
19              }
20          }
```

4. Si implementi una sottoclasse concreta di Formation, chiamata Formation433, che, per essere valida, deve essere composta da 4 difensori, 3 centrocampisti e 3 attaccanti, oltre ovviamente a un portiere. L'implementazione di isValid() dovrà quindi cambiare per questa sottoclasse.

5. Si scriva una classe di prova Main con un metodo main() che crea 11 giocatori: 4 difensori (Alex Sandro, Rugani, Chiellini e Dani Alves), 3 centrocampisti (Fabinho, Iniesta, Pjanic), 3 attaccanti (Dybala, Higuain, Bernardeschi) e un portiere (Szczesny). Poi crea una Formation433, passando tali giocatori al costruttore, e la stampa.

Svolgiamo gli esercizi:

1. SoccerPlayer:

```
1   package EsercizioPallone;
2
3   public interface SoccerPlayer {
4       String toString();
5       boolean canUseHands();
6   }
```

Abstract SoccerPlayer:

```
1   package EsercizioPallone;
2
3   public abstract class AbstractSoccerPlayer
        implements SoccerPlayer {
4     public final String name;
5
6     protected AbstractSoccerPlayer(String name) {
7       this.name = name;
8     }
9
10    public final String toString() {
11      return "Nome: "+name;
12    }
13  }
```

2. Forward.java:

```
1   package EsercizioPallone;
2
3   public class Forward extends AbstractSoccerPlayer{
```

```
4
5       public Forward(String name){
6         super(name);
7       }
8
9       public boolean canUseHands() {
10        return false;
11      }
12    }
```

Midfield.java e Defence.java sono uguali a Foward.java invece GoalKeeper.java sara cosi:

```
1         package EsercizioPallone;
2         public class GoalKeeper extends
              AbstractSoccerPlayer{
3           public GoalKeeper (String name) {
4             super(name);
5           }
6           public boolean canUseHands() {
7             return true;
8           }
9         }
```

3. Formation.java:

```
1  package EsercizioPallone;
2
3  public class Formation {
4    private SoccerPlayer[] arrayPlayers;
5
6
7
8    public Formation (SoccerPlayer[] players) {
9      this.arrayPlayers = players;
10     if(!isValid()) {
11       throw new IllegalArgumentException("invalid
             formation");
12     }
13   }
14
15   protected boolean isValid() {
16     // ritorna true solamente sela formazione e fatta da
           11 giocatori e uno    il portiere
17     if (arrayPlayers.length >= 11 && count() == 1 ) {
18       return true;
19     }
20     return false;
21   }
22
```

```
23    private int count () {
24      int count = 0;
25      for(int i=0;i<arrayPlayers.length;i++) {
26        if(arrayPlayers[i] instanceof GoalKeeper) {
27          count ++;
28        }
29      }
30      return count;
31    }
32
33    protected SoccerPlayer[] getPlayers(){
34      return arrayPlayers;
35    }
36
37    public final String toString() {
38      // ritorna i nomi dei giocatori della formazione,
            separati da virgola
39      String result = "";
40      for (SoccerPlayer player: arrayPlayers)
41        if (result.isEmpty())
42          result += player;
43        else
44          result += ", " + player;
45
46      return result;
47    }
48  }
```

4. Formation433.java

```
1   package EsercizioPallone;
2
3   public class Formation433 extends Formation{
4     public Formation433(SoccerPlayer[] players) {
5       super(players);
6     }
7
8     protected boolean isValid() {
9       if(super.isValid() && countdif()==4 && countatt()==3
            && countMid()==3) {
10        return true;
11      }
12      return false;
13    }
14
15    private int countdif() {
16      int count = 0;
17      for(SoccerPlayer player : getPlayers()) {
18        if(player instanceof Defence) {
19          count++;
```

```
      }
    }
    return count;
  }

  private int countatt() {
    int count = 0;
    for(SoccerPlayer player : getPlayers()) {
      if(player instanceof Forward) {
        count++;
      }
    }
    return count;
  }
  private int countMid() {
    int count = 0;
    for(SoccerPlayer player : getPlayers()) {
      if(player instanceof Midfield) {
        count++;
      }
    }
    return count;
  }
}
```

5. Main.java:

```
    package EsercizioPallone;

public class Main {

  public static void main(String[] args) {
        SoccerPlayer[] players = new SoccerPlayer[11];

        players[0] = new Defence("Alex Sandro");
        players[1] = new Defence("Rugani");
        players[2] = new Defence("Chiellini");
        players[3] = new Defence("Dani Alves");

        players[4] = new Midfield("Fabinho");
        players[5] = new Midfield("Iniesta");
        players[6] = new Midfield("Pijanic");

        players[7] = new Forward("Higuain");
        players[8] = new Forward("Dybala");
        players[9] = new Forward("Bernandeshi");

        players[10] = new GoalKeeper("Szczesny");

```

```
23          Formation formazione = new Formation433(players)
                ;
24
25
26          formazione.toString();
27          System.out.println(formazione);
28      }
29
30  }
```

13.6 Esercitazione 6

1. Si crei un progetto Eclipse e si copi al suo interno la seguente interfaccia, che rappresenta un numero non negativo, in una qualsiasi base di numerazione:

```
1           public interface Number extends Comparable<
                Number> {
2             int getValue(); // restituisce il valore di
                  questo numero
3           }
```

2. Si completi la seguente implementazione astratta di un Number, che fornisce le funzionalità comuni a tutti i numeri, cioè il controllo sulla non negatività del valore, l'accesso al valore, la traduzione in stringa e il metodo per il test di uguaglianza:

```
1           public abstract class AbstractNumber implements
                Number {
2     private final int value;
3
4     protected AbstractNumber(int value) {
5       // se value e negativo, esegue throw new
            IllegalArgumentException(); altrimenti
            inizializza il campo value
6       ...
7     }
8
9     // restituisce il valore di questo numero
10    public final int getValue() { ... }
11
12    // restituice la base di numerazione di questo numero
13    protected abstract int getBase();
14
15    // restituisce il carattere che rappresenta la cifra "
          digit" nella base di numerazione
16    // di questo numero. Sara sempre vero che 0 <= digit <
           getBase();
```

```
17    // per esempio, in base sedici si avra getCharForDigit
          (10) == 'A' e
18    // in base otto si avr  getCharForDigit(7) == '7'
19    protected abstract char getCharForDigit(int digit);
20
21    // restituisce una stringa che rappresenta il numero
          nella sua base di numerazione
22    public String toString() { ... }
23
24    public final boolean equals(Object other) {
25      // due numeri sono uguali se e solo se hanno lo
            stesso valore
26      ...
27    }
28
29    public final int compareTo(Number other) {
30      // l'ordinamento fra i Number    quello crescente
            per valore
31      ...
32    }
33  }
```

3. Si scrivano le sottoclassi concrete DecimalNumber, BinaryNumber, OctalNumber ed HexNumber di AbstractNumber, che rappresentano, rispettivamente, un numero in base 10, 2, 8 e 16. Queste classi si instanziano con il loro costruttore, a cui viene passato il valore del numero. Non si ridefinisca, in queste quattro sottoclassi, il metodo toString(): quello ereditato da AbstractNumber dovrà funzionare per tutte queste sottoclassi, traducendo il valore del numero nella giusta base di numerazione.

4. Nella codifica binaria con parità, un numero binario viene esteso con un'ulteriore cifra binaria di controllo, in modo da rendere pari il numero totale di cifre 1. Se quindi il numero binario aveva una quantità pari di 1, si aggiungerà una cifra di controllo 0. Se invece il numero binario aveva una quantità dispari di 1, si aggiungerà una cifra di controllo 1. Questa modifica riduce il rischio di trasmissione di dati corrotti, permettendo di implementare un rudimentale sistema di rilevazione dell'errore. Si implementi una sottoclasse concreta BinaryNumberWithParity di BinaryNumber, ridefinendo solo il metodo toString() in modo da aggiungere in fondo la cifra di controllo opportuna.

5. Nella codifica in base 58, si utilizzano 58 cifre diverse, scelte fra i numeri arabi e le lettere inglesi maiuscole e minuscole. Si evitano i caratteri 0OIl, che potrebbero essere confusi a video, perché graficamente simili. Si implementi una sottoclasse concreta Base58Number di AbstractNumber, in modo da implementare questa numerazione in base 58. Le 58 cifre sono quindi 123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghi-

jkmnopqrstuvwxyz. Non si ridefinisca il metodo toString() ereditato da AbstractNumber.

6. Si scriva una classe di prova MainNumbers con un metodo main() che chiede all'utente di inserire un numero non negativo n, quindi crea il numero n in base 10, poi in base 2, poi in base 2 con parità, poi in base 8, poi in base 16 e infine in base 58, stampando tutti tali numeri. Se per esempio l'utente inserisse il numero 1234567

7. Si scriva una classe di prova MainNumbersSort con un metodo main() che crea un array di Number contenente esattamente sei elementi:

   - 2024 in base 10
   - 113 in base 2
   - 158 in base 2 con parità
   - 827 in base 8
   - 2066 in base 16
   - 8092 in base 58

   Quindi ordina l'array con java.util.Arrays.sort(...) e lo stampa sfruttando java.util.Arrays.toString(...).

1. Number.java:

```
        package Numeri;

public interface Number extends Comparable<Number>{
        int getValue();// restituisce il valore di un numero
}
```

2. AbstractNumber.java:

```
1       package Numeri;
2
3  public abstract class AbstractNumber implements Number {
4    private final int value;
5
6    protected AbstractNumber(int value) {
7      if(value<0) {
8        throw new IllegalArgumentException("Valore
             Negativo");
9      }else {
10       this.value = value;
11     }
12   }
13
14   public final int getValue() {
15     return this.value;
```

```
16    }
17
18    protected abstract int getBase();
19
20    protected abstract char getCharForDigit(int digit);
21
22    public String toString() {
23      int base = getBase();
24      String result = "";
25      int v = value;
26      // divide v ripetutamente per base fino ad arrivare
            a 0
27      // ogni volta prende v % base ed usa getCharForDigit
            ()
28      // per trasformarlo in char, che concatena a result
29      do {
30        result = getCharForDigit(v % base) + result;
31        v /= base;
32      }
33      while (v > 0);
34
35      return result;
36    }
37
38    public final boolean equals(Object other) {
39      return other instanceof Number otherAsNumber &&
            value == otherAsNumber.getValue();
40    }
41    public final int compareTo(Number other) {
42      return value - other.getValue();
43    }
44
45
46  }
```

3. DecimalNumber.java :

```
1           package Numeri;
2
3   public class DecimalNumber extends AbstractNumber{
4     public DecimalNumber(int value) {
5       super(value);
6     }
7     protected int getBase() {
8       return 10;
9     }
10    protected char getCharForDigit(int digit) {
11      String stringa = "0123456789";
12      return stringa.charAt(digit);
13    }
```

```
}
```

Le altre classi si svolgono nello stesso modo solo cambiando la stringa in maniera adeguata.

4. BinaryNumberWithParity.java:

```
package Numeri;

public class BinaryNumberWithParity extends BinaryNumber
    {

  public BinaryNumberWithParity(int value) {
    super(value);
  }

  public String toString() {
    String stringa = super.toString();
    if(countone(stringa)%2==0) {
      return stringa + '0';
    }else {
      return stringa + '1';
    }
  }

  private int countone(String number) {
    int counter = 0;
    for(int i = 0 ; i<number.length();i++) {
      if(number.charAt(i) == '1') {
        counter++;
      }
    }
    return counter;
  }

}
```

5. Questo punto si svolge nello stesso identico modo del punto 3

6. MainNumbers.java:

```
package Numeri;

import java.util.Scanner;

public class Main {

  public static void main(String[] args) {

    Scanner keyboard = new Scanner(System.in);
```

```
    int n;
    do {
      System.out.println("Inserisci un valore:");
      n = keyboard.nextInt() ;
    }while(n<0);

    keyboard.close();

    Number n0 = new DecimalNumber(n);
    Number n1 = new BinaryNumber(n);
    Number n2 = new BinaryNumberWithParity(n);
    Number n3 = new OctalNumber(n);
    Number n4 = new HexNumber(n);
    Number n5 = new Base58Number(n);

    System.out.println("Numeri:");
    System.out.println(n0);
    System.out.println(n1);
    System.out.println(n2);
    System.out.println(n3);
    System.out.println(n4);
    System.out.println(n5);




  }

}
```

7. MainNumberRandom.java:

```
package Numeri;

import java.util.Arrays;
public class MainNumbersSort {
  public static void main(String[] args) {
    Number [] array = new Number[6];

    array[0] = new DecimalNumber(2024);
    array[1] = new BinaryNumber(113);
    array[2] = new BinaryNumberWithParity(158);
    array[3] = new OctalNumber(827);
    array[4] = new HexNumber(2066);
    array[5] = new Base58Number(8092);

    Arrays.sort(array);
    System.out.println(Arrays.toString(array));


```

```
19      }
20  }
```