Introduzione

giovedì 12 aprile 2018 10.11

Il Java è un linguaggio orientato agli oggetti. Un'applicazione è cioè strutturata come un insieme di componenti detti **classi**. Istanze delle classi sono gli **oggetti**. Una classe non è altro che la descrizione di una categoria concettuale.

Java soddisfa la clausola "Code once, run everywere" poiché i .class generati mediante compilazione possono essere eseguiti su qualsiasi dispositivo dotato di **Java Virtual Machine** (JVM)

In Java, a parte i tipi primitivi quali int, float, double, boolean…, tutte le altre variabili sono sostanzialmente **riferimenti** (puntatori) ad oggetti in memoria.

Stringhe

venerdì 9 marzo 2018 10.58

In C le stringhe sono viste come array di caratteri terminante con il carattere '\0' , sono pertanto modificabili e costituiscono un rischio se usati scorrettamente (accesso alla memoria diretto senza controlli).

In Java le stringhe sono invece viste come oggetti, istanze della classe String, e **non** sono **modificabili**.
Per accedere all'elemento i-esimo infatti occorre utilizzare una notazione diversa da quella del C…

Java:
```
String s = "Nel mezzo del cammin";
char ch = s.charAt(4);
```
Mentre in C# viene mantenuta la stessa notazione…
C#:
```
string s = "Nel mezzo del cammin";
char ch = s[4];
```

La classe String in Java è formata da:

- una parte dinamica: funzioni proprie dell'oggetto
- una parte dinamica: funzioni che non hanno a che vedere con gli oggetti String ma utili per conversioni da altri tipi

  Es:
  ```
  String s = String.valueOf(1.44F);
  ```

Anche nel caso della concatenazione, ottenuta mediante l'operatore +, le stringe di partenza non vengono modificate:
```
String   s1 = "ciao",
         s2 = " mondo!\n";
String s3 = s1 + s2;       //"ciao mondo!\n"
```

In Java due stringhe si possono confrontare:

- con l'op. == : per confrontare i riferimenti
- con equals : per confrontare i valori

In C# entrambe gli operatori attuano il confronto tra valori

Ogni classe Java (o C#) dispone di un metodo **toString** (ToString in C#) con cui ottenere una rappresentazione di un'istanza della classe ed è a carico del progettista definirne uno altrimenti:

- In Java il toString predefinito stampa un identifictivo alfanumerico dell'istanza
- *In C# il ToString predefinito stampa il nome della classe a cui appartiene l'istanza*

Per ridefinire il metodo occorrerà scrivere:

- In Java:

  @Override //Notazione non obbligatoria per comunicare al compilatore l'intenzione di fare override
  public String toString()
- *In C#:*

  public override string ToString()

Array

venerdì 9 marzo 2018 11.38

In C gli array erano un'illusione: non erano altro che un puntatore ad un'area di memoria che costiruiva il primo elemento dell'array e pertanto automaticamente l'array:

- veniva passato per indirizzo
- non poteva essere copiato
- non poteva essere restituito da una funzione
- non si può sapere quanti elementi contenga se passato come parametro di una funzione

In Java e C# gli array sono oggetti, istanze di una classe speciale denotata da [ ] e pertanto come per ogni oggetto va prima definito un riferimento e poi creata un'istanza:

```
int[] v = new int[3];          //3 variabili
MyClass[] w = new MyClass[6];  //6 riferimenti a null
```

Si può conoscere la lunghezza dell'array che, una volta creato, rimane fissata:

```
v.length    //è 3
```

```
Per iterare automaticamente tutti gli elementi di un array esistono i costrutti:
```

Java:

```
    for(int x : v){…}
```

C#:

*   foreach(int x in v){…}*

Enumerativi

giovedì 22 marzo 2018 11.09

Il tipo enumerativo è una calsse, il costruttore è privato, gli oggetti sono precostruiti in base a quelli specificati:

```
public enum Direction {
     NORTH, SOUTH, EAST, WEST;
}
Direction dir = Direction.NORTH;
```

Metodi autoaggiunti dal compilatore:

- costruttore privato
- ordinal() : restituisce l'indice del valore nell'elenco specificato
- values() : restituisce un array di tutti i possibili valori
- valueOf(String s) : restituisce l'oggetto enum corrispondente alla stringa data

E' possibile per noi aggiungere altri metodi.

Package

giovedì 22 marzo 2018 11.35

Il **package** risolve il problema del conflitto tra nomi di classi poiché possono esistere due classi con lo stesso nome in packages diversi.

Potremo pensare che le classi appartenenti allo stesso package possano avere visibilità di alcuni elementi tra loro. Per permettere tale visbilità SOLO per le classi dello stesso package va lasciata la visibiità di default davanti alla variabile
Es:

```
private int var; // privato
int var;         // visibilità package
public var;      // pubblico
```

Per identificare i packages univocamente viene utilizzato un nove strutturato partendo dal generale al particolare come ad esempio nel filesystem o nel web
Es:

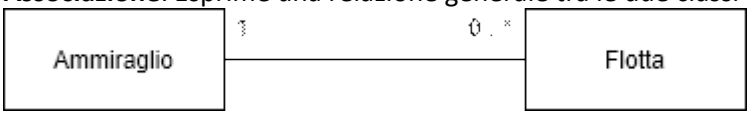
```
import it.unibo.s00001234.utilities.Point
Point p;
p = new Point(x,y);
```

UML

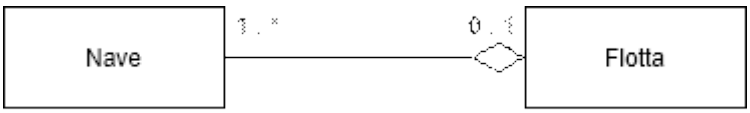
venerdì 23 marzo 2018 09.23

UML (Unified Modelling Language) è un linguaggio grafico per esprimere modelli di un sistema a oggetti

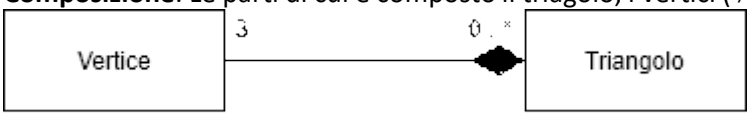
- **Associazione**: Esprime una relazione generale tra le due classi



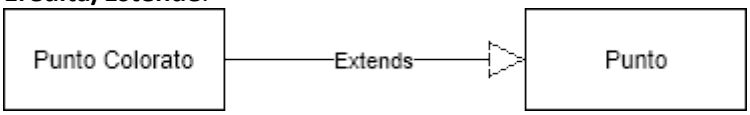
- **Aggregazione**: Le parti di cui è composta la flotta, le navi, esistono indipendentemente se appartengono alla flotta o meno.



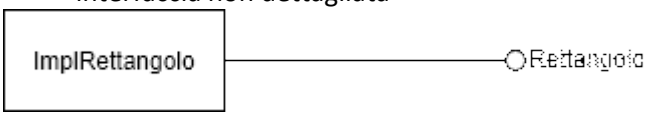
- **Composizione**: Le parti di cui è composto il triagolo, i vertici (≠ punti) esistono solo se esiste il trianolo.



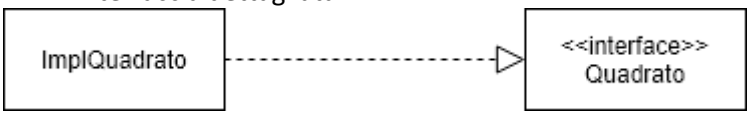
- **Eredita/Estende**:



- **Implementa:**
  - interfaccia non dettagliata



  - interfaccia dettagliata



Legenda:

+	public
-	private
#	protected
:	type
italics	abstract

Factory

venerdì 23 marzo 2018 10.24

La **factory**, o fabbrica, è un **design pattern** che consiste nel lasciare i costruttori privati e inserire un metodo statico definito come **getNew**ClasseIstanza() che applica determinate procedure per la corretta creazione di una classe o, se non possibile, restituire un puntatore a null

Gestione del tempo

venerdì 23 marzo 2018 10.37

Da Java 8 la gestione del tempo avviene da parte delle librerie del package java.time e tutti gli oggetti sono immutabili.

Enumerativi:

DayOfWeek

MONDAY, TUESDAY…

```
DayOfWeek d = DayOfWeek.of(1); --> MONDAY
d.getvalue();                  --> 1
```

Month

JANUARY, FEBRUARY…

```
Month m = Month.of(10);        --> OCTOBER
m.getValue();                  --> 10
```

LocalDate / LocalTime / LocalDateTime

```
LocalDate xmas2017 = LocalDate.of(2017, 12, 25);
LocalTime noon= LocalTime.of(12, 0, 0);
LocalDateTime xmas2017noon = LocalDateTime.of(xmas2017, noon);
                           LocalDateTime.of(2017, 12, 25, 12, 0, 0);
```

Aritmetica

```
LocalDateTime date = LocalDateTime.now();
```

date.plusYears(1);	date.minusYears(1);	date.withYear(2000);
date.plusMonths(1);	date.minusMonths(1);	date.withMonth(1);
date.plusWeeks(1);	date.minusWeeks(1);	
date.plusDays(1);	date.minusDays(1);	date.withDayOfMonth(1);date.withDayOfYear(1);
date.plusHours(1);	date.minusHours(1);	date.withHours(12);
date.plusMinutes(1);	date.minusMinutes(1);	date.withMinutes(0);
date.plusSeconds(1);	date.minusSeconds(1);	date.withSeconds(0);

```
LocalDateTime date1 = LocalDateTime.now();
LocalDateTime date2 = date1.plusDays(1);
```

date1.isBefore(date2);	date1.isEqual(date2);	date1.isAfter(date2);

Duration

```
Duration d = Duration.between(TemporalAccessor start, TemporalAccessor end);
             Duration.ofDays(1); Duration.ofHours(1); Duration.ofMinutes(1);
             Duration.ofSeconds(1);
long convertedVal = d.toDays(1); d.toHours(1); d.toMinutes(1); d.toSeconds(1);
```

Formattatori e supporto all'internazionalizzazione

venerdì 23 marzo 2018 11.23

Un' ulteriore problematica è quella di tenere conto dei separatori per la parte decimale e le migliaia, il formato delle date e delle ore, i nomi dei giorni, dei mesi, simboli di valuta ecc..

In java esiste il concetto di cultura locale in java.util.Locale

```
float x = 3004,98;
```

```
NumberFormat formatNum = NumberFormat.getNumberInstance(Locale.ITALY);
String strNum = formatNum.format(x); // in Italia 3.004,98
Number num = formatNum.parse(strNum);
float x = num.floatValue();

NumberFormat formatPerc = NumberFormat.getPercentInstance();
System.out.println(format.format(x));

NumberFormat formatCurr = NumberFormat.getCurrencyInstance();
System.out.println(formatCurr.format(x); // in Italia (con l'euro) € 3.004,98
```

```
LocalDateTime date LocalDateTime.now();
```

```
DateTimeFormatter formatDateTime = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM,
                                                                         FormatStyle.SHORT);
System.out.println(formatDateTime.format(date));
```

19/03/15	SHORT
19-mar-2015	MEDIUM
19 marzo 2015	FULL
giovedì 19 marzo 2015	LONG

18.51	SHORT
18.51.16	MEDIUM

Ereditarietà - progett. incrementale

giovedì 5 aprile 2018 10.26

L'idea è quella di creare un oggetto composto che incapsuli il componenete esistente con tutte le sue funzionalità e lo integri con i nuovi bisogni (design pattern: **adapter**).

```
public class Counter2 extends Counter { ... }
```

Per dare accesso agli attributi della classe ereditata alle classi che ereditano esiste il livello di visibilità **protected**.
Di solito tale qualifica viene applicata ai metodi set in modo che comunque si possano definire dei filtri di accesso ai parametri:

Es.

```
public class Counter{
    private int val;
    ...

    protected void setVal(int val){ ... }
    public int getVal(){ ... }
    ...
}
```

Per fare riferimento alla classe ereditata esiste la parola chiave **super**, valida anche per riferire il costruttore con super(...) .

Mediante la clausola **final**, inoltre, è possibile:

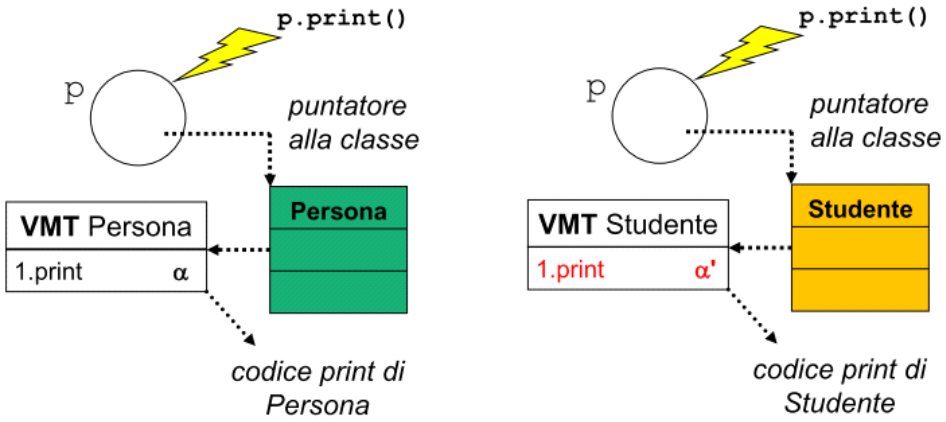
```
public final void inc(){...}   // metodi non possano essere ridefiniti
public final class LastCounter extends Counter{...}  // classi non possono essere estese
```

Polimorfismo

venerdì 6 aprile 2018 08.13

Il **polimorfismo** è una proprietà dei metodi in Java, i quali sono in grado di adattare il loro comportamento allo specifico oggetto su cui opera.

La tecnica su cui si fonda è quella del **late binding** ossia rimandare a runtime la decisione su quale metodo debba essere chiamato. Ad ogni classe viene infatti associata una tabella detta **Virtual Method Table** (VMT) in cui viene associato ad ogni metodo dove trovare il suo codice.
Ogni oggetto contiene un riferimento alla classe di cui è istanza e da lì la JVM riesce a risalire al metodo corretto.



Tipi generici

giovedì 12 aprile 2018 10.28

Per scrivere componenti parametrici rispetto al tipo, Java permette di parametrizzare utilizzando il **tipo generico** mediante la key-word **<T>**:

```
Es:
//Definizione
public static <T> boolean idem(T[] a, T[] b);
```

In questo modo vogliamo intendere che il tipo dei due array deve essere lo stesso (ovviamente anche classi che ereditano dal tipo definito). Cosa che non sarebbe stata possibile con tipo generico Object, che avrebbe permesso di compilare passando qualsiasi combinazione di tipi.

```
//Chiamata
nomeclasse.<Classe>idem(arr1, arr2); //nel caso in cui la classe non fosse specificata si presume Object
```

Entità astratte

giovedì 12 aprile 2018 10.50

Java mette a disposizione degli strumenti per descrivere pure **categorie concettuali** (Es: animali, rocce). Tale sistema viene esaurito dalle **classi astratte** mediante la key-work **abstract**.
Tali entità potranno essere definite ma non istanziate.
Il discorso vale anche per i metodi di tali classi:

Es.

```
public abstract class Animale {
     public abstract String move();
}
```

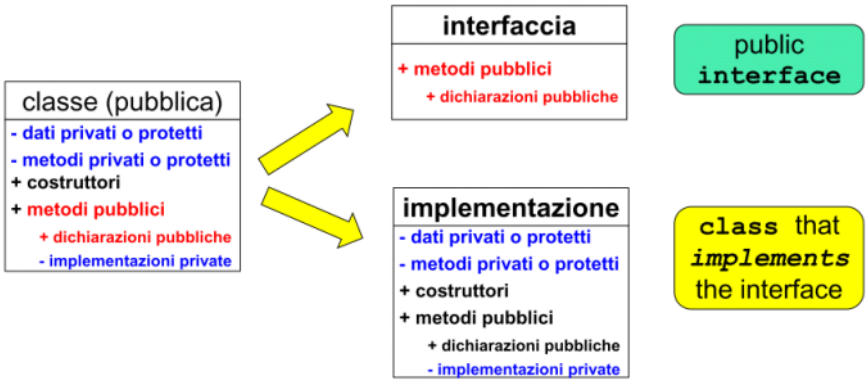
Interfacce

venerdì 13 aprile 2018 09.19

Lavorando con le entità astratte si presentano determinate limitazioni progettuali:

- Le operazioni lasciate "in bianco" possono essere implementate solo in una sottocalsse
- L'ereditarietà multipla non permette di esprimere il caso di intersezione di due insiemi della realtà e l'ereditarietà multipla fra classi genera caos

Il costrutto **interface** mette a disposizione gli strumenti per affrontare tali aspetti nella maniera corretta.



Il concetto è quello di poter estrapolare e separare le proprietà degli oggetti dall'implementazione vera e propria. Le interfacce non contengono infatti attributi se costanti (final)

Es:

```
public interface Rettangolo {
     public final int ampiezzaAngoli = 90;
     public double altezza();
     public double base();
}

class RettangoloImpl implements Rettangolo {
     private double lato1, lato2;
     public RettangoloImpl(double altezza, double base) {
          lato1 = base;
          lato2 = altezza;
     }
     public double base(){return lato1;}
     public double altezza() {return lato2;}
     ...
}
```

Da Java 8 le interfacce possono contenere metodi statici già definiti. L'efficienza massima si ha sfruttando questa feature per poter definire delle factory all'interno delle interfacce:

```
public interface Rettangolo{
     public static Rettangolo of(...){
          ...
          return new ImplRettangolo(...);
     }
}

Rettangolo r = Rettangolo.of(...);
```

Con lo svantaggio di perdere la factory generale che costruisce ogni astrazione.

Interfacce standard

venerdì 13 aprile 2018 16.52

Esistono molte interfacce standard, destinate ad esprimere:

- **Abilità** (che hanno come suffisso "-able"):
  Es: Comparable, Printable …
  tra cui anche le **interfacce marker** che non dichiarano alcun metodo ma che fungono da tag per identificare determinate caratteristiche di oggetti e bloccare eventuali incongruenze a compile-time.
- **Concetti astratti**:
  Es: azione, ascoltatore di eventi, servizio di stampa …

Comparable è un'interfaccia tipizzata cioè è possibile specificare il tipo su cui andare a lavorare con le stesse modalità dei tipi generici.

Es:

```
public class Counter implements Comparable<Counter>{
    public int compareTo(Counter that){
        ...
    }
}
```

Ovviamente, come per i tipi generici, se non viene specificato nulla si presume Object.

Tipo Optional

giovedì 19 aprile 2018 15.35

Tipicamente quando delle funzioni hanno risultati impossibili o non disponibili il pattern consigliato per il tipo di dato da restituire è quello del tipo **Optional<T>**.

Il tipo optional ci permette di incapsulare un possibile valore T in modo da evitare che l'utente lavori con un dato null ; tale dato sarà invece nascosto da questo tipo e per essere usato dovrà essere estratto verificando però che contenga il valore.
Così facendo chi vorrà utilizzare tale risultato non avrà direttamente a che fare con un oggetto "pericoloso" ma con un dato strutturato.

Optional prevede i metodi:

```
Optional.of(...)        per incapsulare un valore
Optional.empty(...)     per esprimere il "nessun valore"
isPresent()             per accertare la presenza del valore
get()                   per estrarre il valore
```

Lambda Expression

venerdì 20 aprile 2018 07.29

La funzione acquista l'aspetto di una classe. Una particolare funzione non è altro che un'istanza di tale classe e per questo motivo potrebbe essere assegnata ad una variabile.
Questa è l'idea di fondo dell'introduzione di **Lambda Expression**.

Dal punto di vista pratico una lambda expression è realizzata mediante un'interfaccia.
Un'interfaccia che dichiara un solo metodo è detta interfaccia funzionale e può essere etichettata come @FunctionalInterface.

```
([Type] arg1, [Type] arg2, ...) -> { corpo della funzione }
```

In alcuni casi non è nemmeno necessario specificare il tipo degli argomenti poiché il compilatore riesce a capire autonomamente il tipo mediante un meccanismo detto Type Inference.

A volte le espressioni non sono altro che il risultato di un metodo dell'oggetto del dominio, in questo caso si possono usare tali scorciatoie:

```
nomeclasse::nomemetodo
istanza::nomemetodo
```

L'utilità si ha utilizzando delle apposite factory di LambdaExpression che in alcuni prcessi comuni ci permettono di risparmiare parecchio tempo. Nell'esempio si ordina un array di persone in base al cognome, in subordine in base al nome e in subordine in base all'eta:

```
Arrays.sort(persone,
    Comparator.comparing(Persona::getCognome)
              .thenComparing(Persona::getNome)
              .thenComparing(Persona::getEta)
);
```

Eccezioni

venerdì 20 aprile 2018 11.33

In certe situazioni critiche è prevedibile che la chiamata di un certo metodo possa causare errori:

- apertura di un file
- connessione di rete
- conversione da una stringa a numero

Le **eccezioni** sono l'approccio adottato per bloccare la creazione di oggetti nulli o peggio non validi.
La clausola *throws* nela firma di un metodo indica un allarme: potrebbero esserci operazioni rischiose che potrebbero generare valori non validi.
Tale clausa comporta l'introduzione di un ambiente controllato: il costrutto *try/catch*.
Per lanciare un'eccezione all'interno di un metodo la clausola è *throw*.

Le eccezioni che derivano da Exception è obbligatorio bloccarle invece quelle che derivano da RuntimeException è facoltativo gestirle.

Gestione I/O binario

giovedì 26 aprile 2018 14.18

```
java.io
java.nio      (Java 4)
java.nio.file (Java 7)
```

Cronologia versioni

Con Java 7 e il nuovo package java.nio.file viene introdotta l'astrazione **Path**, un'interfaccia la cui costruzione avviene tramitei metodi della fatctory **Paths** e permette di:

- recuperare informazioni su file o directory
- operare su percorsi

```
Paths.get(String str)
```

Per operare su file da Java 7 si utilizzano i metodi statici della factory **Files** introdotta sempre in java.nio.file.

```
Files.exists(Path p)
Files.move(Path src, Path dst)
Files.isDirectory(Path p)
Files.list(Path p)
Files.readAllBytes(Path p)
Files.write(Path p, byte[] bytes, OpenOption... opt)
```

Il package java.io definisce invece i concetti base e l'architettura per gestire l'I/O.
All'interno si trova la classe astratta **InputStream** che definisce il concetto generale di "canale di input" operante a byte e la classe astratta **OutputStream** che definisce il concetto generale di "canale di output" operante a byte.
Da qui derivano tutte le classi specifiche per le varie tipologie di sorgenti di dati.

Alcuni stream di nostro iteresse sono:

Input

`DataInputStream`

- `readInteger(), readDouble(), readBoolean(), ...`

`ObjectInputStream`

- `readObject(), readInteger(), readDouble(), ...`

Output

`DataOutputStream`

- `writeInteger(int d), writeDouble(double d), ...`

`ObjectOutputStream`

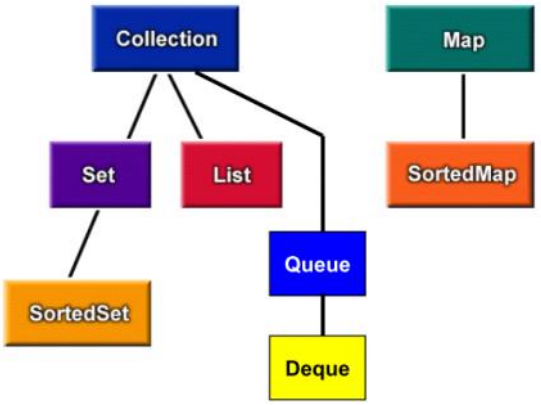
- `writeObject(Object obj), writeInteger(int d), writeDouble(double d), ...`

Per permettere che un oggetto possa essere serializzato/deserializzato dai metodi precedentemente elencati esso deve implementare l'interfaccia vuota **Serializable** in java.io; altrimenti, per motivi di sicurezza e riservatezza dei dati, l'operazione viene bloccata.

Collezioni

giovedì 3 maggio 2018 12.28

`java.util`



L'interfaccia *Collection<T>* introduce l'idea di **collezione di elementi** e deve almeno implementare i metrodi base:

- add
- remove
- contains
- isEmpty
- size
- toArray
- equals

L'interfaccia *Set<T>* è un insieme di elementi **privo di duplicati**.
L'interfaccia *SortedSet<T>* è un insieme di elementi privo di duplicati e **ordinato**.
L'interfaccia *List<T>* è una **sequenza** di elementi.
L'interfaccia *Queue<T>* è una coda di elementi (si aggiunge in coda e si toglie in testa).
L'interfaccia *Deque<T>* è una doppia coda di elementi (si aggiunge/toglie da entrambe le estremità).

L'interfaccia *Map<T>* introduce l'idea di **tabella di element**i associati ad una **chiave identificativa univoca**.
L'interfaccia Sorted*Map<T>* è una mappa di elementi univocamente identificati e **ordinata**.

Iteratore

Per poter definire il modo in cui deve essere effettuato lo scorrimento su una collezione è opportuno definire un iteratore implementando l'interfaccia *Iterator<T>* e riscrivendo i metodi

- next
- hasNext
- remove //opzionale

Il costrutto for each diventa così

```
for(Iterator<Type> i = coll.iterator(); i.hasNext()){
    Type x = i.next();
}
```

La classe deve però implementare l'interfaccia *Iterable*.

Gestione I/O di testo - Lettura

venerdì 4 maggio 2018 09.17

Le classi astratte delegate allo stream di caratteri sono **Reader** e **Writer**. Esse convertono l'UNICODE nella codifica della piattaforma locale tenendo conto della cultura locale.

Reader	Writer
FileReader	FileWriter
BufferedReader	PrintWriter

Lettura da file

```
FileReader fr = new FileReader(filename); throws FileNotFoundException
int x = fr.read();              throws IOException
int[] buf = new int[length];    throws IOException
fr.read(buf, offset, length); throws IOException
```

Il metodo read() restituisce la codifica del carattere
o -1 se non è stato letto nessun carattere / EOF

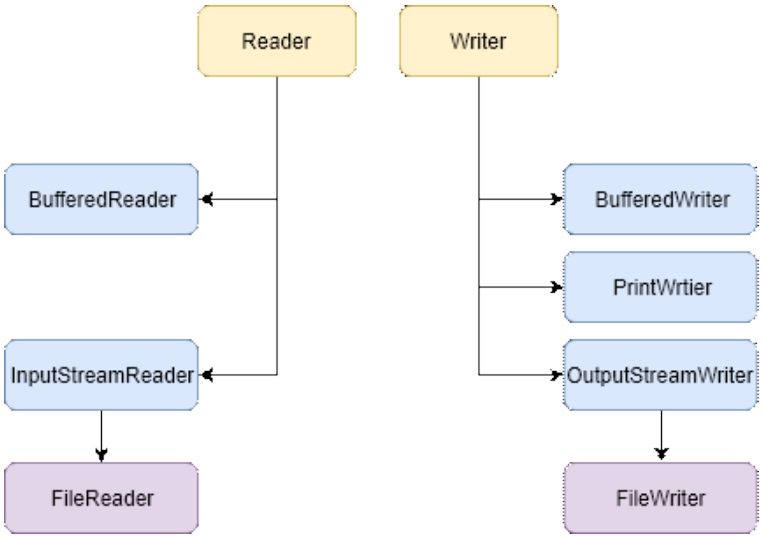
Input bufferizzato

```
BufferedReader br = new BufferedReader(new FileReader(filename)); //più in generale basta un Reader
String line = br.readLine(); throws IOException
```

Il metodo readLine() restituisce la stringa letta
o null se non è stato letto nessun carattere / EOF

Output bufferizzato

```
PrintWriter pw = new PrintWriter(new FileWriter(filename));
pw.print(...);
pw.println(...);

BufferedWriter bw = new BufferedWriter(new FileWriter(filename)); //più in generale basta un Writer
bw.write(...);  throws IOException
bw.flush();
```



Esempio di stream di testo da console

La classe Console è una classe singleton, ne può essere istanziata solo una alla volta.
Si usa tramite il Reader e Writer associati.

```
Console console = System.console();
if(console != null){
    String username = console.readLine("Usr: ");
    char[] password = console.readPassword("Psw: ");
}
```

Gestione I/O di testo - Tokenizzazione

venerdì 4 maggio 2018 10.01

Si preferisce all'implementazione mediante split l'utilizzo di classi apposite come **StringTokenizer** e **Scanner**.

StringTokenizer

StringTokenizer avvolge una stringa già letta e offre metodi per estrarne le parti:

```
StringTokenizer stk = new StringTokenizer(str);
int nTokens = stk.countTokens();
Boolean hasToken = stk.hasMoreTokens();
String token = stk.nextToken();
```

Il costruttore di default usa come separatori tra i token i caratteri: spazio, tab, a capo

E' possibile specificare i caratteri separatori:

```
StringTokenizer stk = new StringTockenizer(str, ":-");
String token = stk.nextToken(":;");
```

Qui si intende sia l'uno che l'altro carattere presi separatamente sono separatori

Es limite:

```
String str = "prova ---1      4  6";
StringTokenizer stk = new StringTokenizer(str, "-");
System.out.println(stk.countTokens());      // 2
System.out.println(stk.nextToken().trim()); // prova
System.out.println(stk.nextToken());        // 1      4  6
```

I token nulli, come in questo caso in cui ci sono separatori concatenati, vengono ignorati dallo Stringtokenizer

Scanner

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
String word = sc.next();

Scanner sc = new Scanner(new File("numeri.txt"));
while (sc.hasNextLong()) {
     long x = sc.nextLong();
}
```

JavaFX - Grafica

giovedì 10 maggio 2018 12.20

Si può operarare:

- A livello di singoli pixel
- A livello di controlli grafici evoluti

`javafx.*`

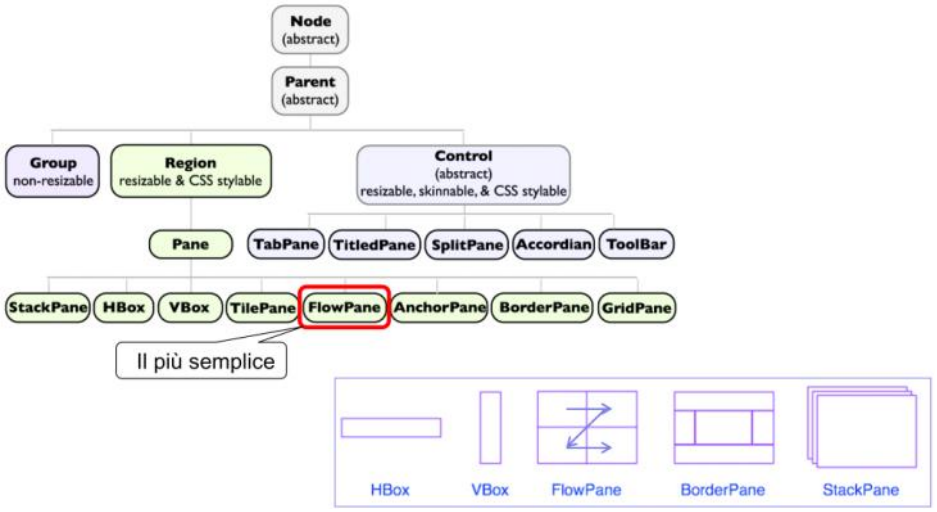
La finestra grafica in JavaFX inplementa la classe base Application. A differenza di Swing, viene messo a disposizione un palcoscenico nel quale l'utente andà a predisporre le varie scene.

Es:

```
public class EsJavaFX00 extends Application{
	public void start(Stage stage){
		stage.setTitle("Esempio 0");
		Pane root = new Pane();
		Scene scene = new Scene(root, 300, 50, Color.YELLOW);
		stage.show();
	}

	// Può anche essere omesso
	public static void main(String[] args){
		launch(args);
	}
}
```

Es:

```
public class EsJavaFX01 extends Application {
	public void start(Stage stage){
		stage.setTitle("Esempio 1");

		FlowPane panel = new FlowPane();
		panel.setPrefSize(200,130);

		Canvas canvas = new Canvas(150, 130);
		panel.getChildren().add(canvas);

		GraphicsContext g = canvas.getGraphicsContext2D();
		g.setFont(Font.font("Serif", FontWeight.BOLD, 20));
		g.setFill(Color.RED);
		g.fillRect(20,20, 100,80);
		g.setFill(Color.BLUE);
		g.strokeRect(30,30, 80,60);
		g.setFill(Color.BLACK);
		g.fillText("ciao",50,60);

		Scene scene = new Scene(panel);

		stage.setScene(scene);
		stage.show();
	}
}
```



JavaFX - Gestione di eventi

venerdì 11 maggio 2018 09.18

La classe che deve gestire l'evento deve implementare l'interfaccia *EventHandler<T>* dove T è il tipo di evento di interesse. Tale classe deve perciò definire il metodo handle in cui vi è il corpo della funzione da svolgere in caso l'evento venga recepito.

```
Es:
public class EsJavaBtnClk extends Application {
	private Label l;
	public void start(Stage stage){
		stage.setTitle("Esempio Click bottone");
		FlowPane panel = new FlowPane();
		Button b = new Button("Click");
		b.setTooltip(new Tooltip("Premere qui"));
		b.setOnAction(new ButtonClicListener(l));
		panel.getChildren().addAll(l,b);
		Scene scene = new Scene(panel,Color.WHITE);
		stage.setScene(scene); stage.show();
	}
}

public class ButtonClickListener implements EventHandler<ActionEvent> {
	private Label l;
	public ButtonClicListener(Label label){
		this.l = label;
	}
	public void handle(ActionEvent event){
		l.setText("Bottone cliccato");
	}
}
```

Oppure invece di creare una nuova classe possiamo ottenere lo stesso risultato usando una lambda expression:

```
Es:
Button b.setOnAction( event -> l.setText("Bottone cliccato"));
```

Nel caso di bottoni differenti che svogono azioni simili un'altra possibile soluzione offerta da Java è quella di passare un metodo in forma di **method reference**.

```
Es:
public class EsJavaRoB extends Application {
	private Button b1, b2; private FlowPane panel;
	public void start(Stage stage){
		stage.setTitle("Esempio rosso/blu");
		panel = new FlowPane();
		b1 = new Button("Rosso");
		b2 = new Button("Blu");
		b1.setOnAction(this::myHandle);
		b2.setOnAction(this::myHandle);
		panel.getChildren().addAll(b1,b2);
		Scene scene = new Scene(panel,Color.WHITE);
		stage.setScene(scene);
		stage.show();
	}
	private void myHandle(ActionEvent event){
		if (event.getSource() == b1) panel.setStyle("-fx-background: red;");
		else panel.setStyle("-fx-background: blue;");
	}
}
```

JavaFX - Proprietà osservabili

venerdì 11 maggio 2018 10.06

Il concetto di proprietà osservabile è espresso dall'interfaccia *ObservableValue<T>* la quale definisce i metodi:

- addListener
- getValue
- removeListener

L'ascoltatore deve implementare l'interfaccia *ChangeListener<T>* e deve definire il metodo
`void changed(ObservableValue<? extends T> obs, T oldValue, T newValue);`

In particolari casi si può arrivare addirittura a non doverlo implementare per niente.

```
Es:
StringProperty docente1 = new SimpleStringProperty(null, "docente 1");
docente1.set("ED");
System.out.println("Proprietà 1: " + docente1.getName());
System.out.println("Valore: " + docente1.get() );
StringProperty docente2 = new SimpleStringProperty(null, "docente 2");
docente2.set("GZ");
System.out.println("Proprietà 2: " + docente2.getName());
System.out.println("Valore: " + docente2.get() );

StringProperty docenti = new SimpleStringProperty(null, "docenti");
docenti.bind( Bindings.concat(docente1, " & ", docente2) );
System.out.println("Proprietà 3: " + docenti.getName());
System.out.println("Valore: " + docenti.get() );

docente2.set("Gabriele");

System.out.println("Proprietà 3: " + docenti.getName());
System.out.println("Valore: " + docenti.get() );
```

Risultato:

```
---------------------------------
Proprietà 1: docente 1
ED
Proprietà 2: docente 2
GZ
Proprietà 3: docenti
ED & GZ
Proprietà 3: docenti
ED & Gabriele
---------------------------------
```

Un possibile caso d'uso è il contatore di caratteri metre si sta scrivento su un'area di testo

```
Es:
...
TextArea area = new TextArea();
Label label = new Label("Caratteri: 0");
area.textProperty().addListener(
      (obs, oldV, newV) -> label.setText( "Caratteri: " + area.getText().length())
);
...
```

JavaFX - Componenti pt.1

venerdì 11 maggio 2018 11.00

TextArea

```
TextArea textArea = new TextArea();
textArea.setEditable(false);
textArea.setText("Testo");
```

ToggleButton

```
ToggleButton toggleBtn = toggleButton("Toggle");
toggleBtn.setOnAction(
    event -> System.out.println("ToggleBtn cliccato")
);
```

CheckBox - ToggleGroup

```
ToggleGroup tg = new ToggleGroup();
CheckBox checkBox1 = new CheckBox("Pere");   checkBox1.setToggleGroup(tg);
CheckBox checkBox2 = new CheckBox("Mele");   checkBox2.setToggleGroup(tg);
CheckBox checkBox3 = new CheckBox("Arance"); checkBox3.setToggleGroup(tg);
tg.selectedToggleProprierty().addListener(
    (changed, oldval, newval) -> System.out.println(((CheckBox)newval).getText());
);
```

RadioButton - ToggleGroup

```
ToggleGroup tg = new ToggleGroup();
RadioButton radioBtn1 = new RadioButton("Pere");   radioBtn1.setToggleGroup(tg);
RadioButton radioBtn2 = new RadioButton("Mele");   radioBtn2.setToggleGroup(tg);
RadioButton radioBtn3 = new RadioButton("Arance"); radioBtn3.setToggleGroup(tg);
tg.selectedToggleProprierty().addListener(
    (changed, oldval, newval) -> System.out.println(((RadioButton)newval).getText());
);
```

ListView

```
ListView listView = new ListView<>();
listView.setItems(FXCollections.observableArrayList("Rosso", "Verde"));
listView.getItems().add("Blu");
listView.getItems().add("Giallo"); //Non lancia eccezioni!
```

ComboBox

```
ComboBox comboBox = new ComboBox<>();
comboBox.setValue(FXCollections.observableArrayList("Rosso", "Verde"));
comboBox.setOnAction(
    event -> System.out.println(comboBox.getValue())
);
comboBox.getItems().add("Blu");
```

DatePicker

```
DatePicker datePicker = new DatePicker();
datePicker.setValue(LocalDateTime.now());
datePicker.getValue();
```

ColorPicker

```
ColorePicker colorPicker = new ColorPicker();
colorPicker.getValue();
```

Slider

```
Slider slider = new Slider(0, 10, 0);
slider.setShowTickMarks(true);
slider.setShowTickLabels(true);
slider.setMajorTickUnit(1);
slider.setMinorTickCount(1);
slider.setBlockIncrement(1);
slider.setSnapToTicks(true);
slider.setOrientation( Orientation.VERTICAL);
```

Spinner

```
Spinner<Integer> spinner = new Spinner(0, 10, 0, 1);
```

JavaFX - Componenti pt.2

giovedì 17 maggio 2018 12.32

ProgressBar

```
ProgressBar bar = new ProgressBar(0.25);              //barra
ProgressIndicator pid = new ProgressIndicator(0.25); //cerchio
bar.progressProperty().addListener(
    (changed, oldval, newval) -> System.out.println(newval);
);
bar.setProgress(0.456);
```

FileChooser

```
FileChooser chooser = new FileChooser();
chooser.setTitle("Apri file");
File f = chooser.showOpenDialog(stage);
File f = chooser.showSaveDialog(stage);
*check if f is null
```

BarChart - StackedBarChart - ScatterChart - AreaChart - LineChart

```
CategoryAxis asseOrizz = new CategoryAxis();
asseOrizz.setLabel("Tipi di frutta");
NumberAxis asseVert = new NumberAxis();
asseVert.setLabel("Vendite");
BarChart<String,Number> chart = new BarChart<>(asseOrizz,asseVert);
chart.setTitle("Andamento vendite frutta");
XYChart.Series<String,Number> modena = new XYChart.Series<>(); modena.setName("Modena");
modena.getData().add( new XYChart.Data<>("Mele", 30));
modena.getData().add( new XYChart.Data<>("Pere", 15));
chart.getData().add(modena);
```

PieChart

```
ObservableList<PieChart.Data> dati = FXCollections.observableArrayList(
    new PieChart.Data("Mele", 30),
    new PieChart.Data("Pere", 15),
    new PieChart.Data("Arance", 50)
);
PieChart chart = new PieChart();
chart.setTitle("Andamento vendite frutta");
chart.setLabelsVisible(true);
chart.setData(dati);
dati.add(new PieCart.Data("Limoni", 10)); //Si modifica in automatico il chart
```

Alert

```
Alert alert = new Alert(AlertType.INFORMATION);
alert.setTitle("Titolo");
alert.setHeaderText("Header");
alert.setContentText("Body");
alert.show();
```

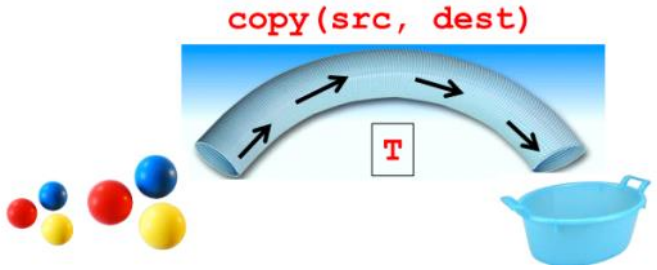
Metodi generici

```
componente.setPrefSize(width, height);
componente.setPrefHeight(height);
componente.setPrefWidth(width);
```

Tipi parametrici varianti (Wildcard)

venerdì 18 maggio 2018 09.25

Può capitare di aver bisogno di definire, ad esempio in una funzione, argomenti generici ma legati una qualche gerarchia. Il caso più comune è quello della copia di una lista in un'altra:



In questo caso gli oggetti da copiare possono essere grandi quanto T o meno, il contenitore può invece essere grande quanto T o anche più.
Java fornisce le wildcard, parole Jolly che ci permettono di definire il concetto sopra indicato.

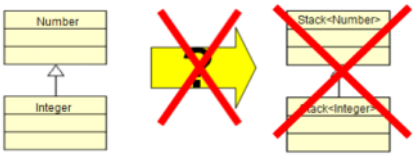
```
<E extends T>    <? extends T>    <E super T>    <? super T>    <?>
```

Una possibile implementazione della funzione copy tra liste potrebbe essere:

```
public static void copy(Collection<? extends T> src, Collection<? super T> dest){
    for(T elem : src) dest.add(elem);
}
```

```
MyStack<Integer> stack1 = new MyStack<>();
MyStack<Number>  stack0 = new MyStack<>();
stack0 = stack1; // STRONCATO!
```

```
TestEs3.java:10: incompatible types
found    : MyStack<java.lang.Integer>
required: MyStack<java.lang.Number>
                stack0 = stack1; // STRONCATO!
                         ^
```



Strutture ad albero

venerdì 18 maggio 2018 10.18

Un **grafo** orientato è una struttura G = <A,R> dove

- A è un insieme non vuoto di nodi
- R è un insieme di archi orientati tali che se <ai,aj> ∈ R, allora c'è un arco orientato da ai verso aj

Un **albero** è un grafo orientato aciclico tale che

- esiste un nodo (radice) con grado d'ingresso 0
- ogni altro nodo ha grado d'ingresso 1

In Java esiste una struttura per definire i nodi di un albero detta **TreeItem<T>**
Es:

```
TreeItem<String> root = new TreeItem<>("Esseri viventi");

TreeItem<String> animali = new TreeItem<>("Animali");
TreeItem<String> vegetali = new TreeItem<>("Vegetali");
root.getChildren().add(animali);
root.getChildren().add(vegetali);
```

TreeSet<T> e **TreeMap<T>** sono inceve alberi bilanciati.
Es:

```
TreeSet<Integer> t1 = new TreeSet<>();
t1.add(8); t1.add(-1); t1.add(121); t1.add(4);
int cont = 0;
for(Integer n : t1){
     cont++;
     System.out.println(cont + ") " + n);
}
//Nel foreach l'iteratore è implicito
Iterator<Integer> iterator = t1.iterator();
while (iterator.hasNext()) {
      Integer n = iterator.next();
      cont++;
      System.out.println(cont + ") " + n);
}
```

Esame

giovedì 24 maggio 2018 13.05

Settaggio locale italiano vecchio Eclipse:
Run -> Run Configurations -> Arguments -> VM arguments -> -Djava.locale.provides=COMPAT

Lavorare con stream di dati:

```
List list = elements.stream()
         .filter(el -> el.getVal().equals(val))
         .collect(Collectors.toList());
```

equivalente:

```
List<Element> list = ArrayList<>();
for(Element el : elements)
    if(el.getVal().equals(val))
         list.add(el);
```

Stream di operazioni

venerdì 25 maggio 2018 11.24

Uno **stream di operazioni** è un'astrazione per specificare e concaternare operazioni.
Non è una collection. Nasce l'idea di abbandonare la programmazione imperativa e di delegare le operazioni al minimo componente indispensabile.

Il modello di pensiero è significativo se si pensa che l'esecuzione di operazioni in parallelo sfruttando tutti i core potrebbe essere direttamente demandata al compilatore.

```
List<String> elencoParole = …;
Stream<String> st = elencoParole.stream();
long numParoleLunghe = st.filter( p->p.length()>4 ).count();
List<String> paroleMaiuscole= st.map( p->p.toUpperCase() ).collect(*);

String[] array = { "ciao", "ragazzi" };
Stream<String> st = Stream.of(array);



Stream<Double> s = Stream.generate(Math::random);
Stream<Integer> pari = Stream.iterate(2, n->n+2);

//importante per gli stream potenzialmente infiniti, limita i primi 10 elementi
Stream<Double> st1 = Stream.generate(Math::random).limit(10);
//Non considera i primi 5 elementi
Stream<Double> st2 = st1.skip(5);
//Concatena ue stream
Stream<Double> st3 = st1.concat(st2);



*
Collectors.toCollection(TreeSet::new)
          .toList()
          .toMap(key,value)
          .toSet()
```