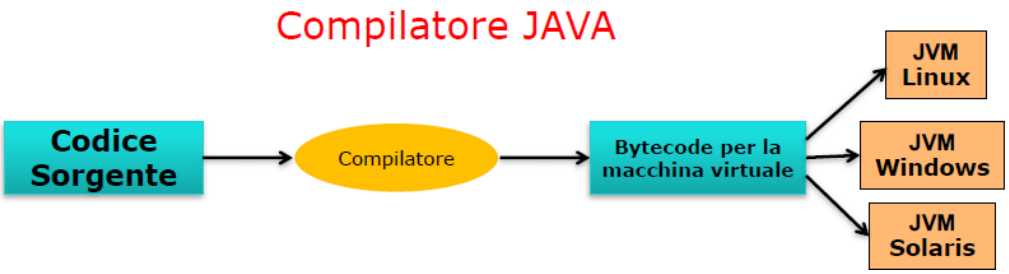
PER ALTRI APPUNTI CONSULTARE IL SITO:

https://luigi-v.github.io/Appunti_Universita/

Per altri appunti consultare il sito: *https://luigi-v.github.io/Appunti_Universita/*

By *Vollono Luigi*

Orale POO (Linguaggio interpretato)



Un file di ***bytecode*** è costituito da una sequenza di caratteri in formato **Unicode**, contiene tutte le informazioni che descrivono le classi ed è dato "in pasto" alla ***JVM*** che ne legge i contenuti.

I ***linguaggi ad oggetti*** permettono al programmatore di rappresentare e manipolare non solo dati numerici o stringhe ma anche dati più complessi e aderenti alla realtà (conti bancari, schede personali,…).

Modelli in Java

- Elementi del modello: ***Oggetti***
- La categorie di oggetti vengono chiamate ***Classi***
- Una ***classe***: Determina il comportamento degli oggetti appartenenti ed è definita da una sezione di codice
- Un ***oggetto***: Appartiene ad una ***classe***, e costituisce una ***istanza*** di tale classe.

Fuoco: gli oggetti e le classi che ne definiscono il comportamento.

Filosofia: In un programma in esecuzione sono gli oggetti che eseguono le operazioni desiderate.

Una ***classe*** è a tutti gli effetti un tipo di dato (come gli interi e le stringhe e ogni altro tipo già definito).

Nomi e referenze

Gli oggetti NON hanno nome l In Java gli oggetti sono identificati da riferimenti.

Un riferimento (**reference**) è una frase che si riferisce ad un oggetto.

```
System.out.println ("Benvenuti al corso")
```

Riferimento — Messaggio

Un ***oggetto*** è un'***entità*** di un programma dotata di tre proprietà caratteristiche

- ***Stato***: Sono informazioni conservate nell'oggetto. Condiziona il comportamento dell'oggetto nel futuro e può variare nel tempo.
- ***Comportamento***: Definito dalle operazioni (***metodi***) che possono essere eseguite dall'oggetto. I metodi possono modificare lo stato dell'oggetto.
- ***Identità***

Quando viene creato l'**oggetto** box con **Rectangle box = new Rectangle(5, 10, 20, 30);** viene allocato uno spazio di memoria in cui sono conservati

```
Rectangle box = new Rectangle(5, 10, 20, 30);
Rectangle r = new Rectangle(5, 10, 20, 30);
```

r e ***box*** si riferiscono a due oggetti che sono indistinguibili rispetto allo stato (stesso stato) e al comportamento, ma hanno identità differenti.

Variabili

Le ***variabili di istanza*** (***campi***) memorizzano lo stato di un oggetto. Ciascun oggetto di una certa classe ha la propria copia delle variabili di istanza. Solitamente possono essere lette e modificate solo dai metodi della stessa classe (***incapsulamento dei dati***).

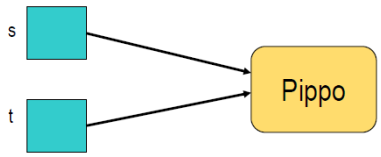
Variabile: un identificatore a cui si può attribuire un valore.

Variabile di riferimento: Una variabile il cui valore è un riferimento ad un oggetto.

Il tipo del valore deve combaciare con il tipo con cui si è dichiarata una variabile (**type matching**).

Una variabile di riferimento si riferisce ad un solo oggetto alla volta. Un oggetto può essere referenziato da più variabili simultaneamente.

```
String s, t;
s="Pippo"
t=s;
```



- ***final*** Il loro valore non può essere modificato (costante).
- ***static*** denota una variabile della classe, quindi non ne viene creata una copia per ogni oggetto istanziato ma tutti gli oggetti fanno riferimento alla stessa variabile.

da \ a	boolean	byte	short	char	int	long	float	double
boolean		n	n	n	n	n	n	n
byte 8bit	n		s	c	s	s	s	s
short 16bit	n	c		c	s	s	s	s
char 16 bit	n	c	c		s	s	s	s
int 32bit	n	c	c	c		s	s*	s
long 64bit	n	c	c	c	c		s*	s
float 32bit	n	c	c	c	c	c		s
double 64bit	n	c	c	c	c	c	c	

n non si applica; s viene fatto automaticamente; c mediante casting esplicito

Messaggi e metodi

Il comportamento di un oggetto è attivato dalla ricezione di un messaggio.

Le classi determinano il comportamento degli oggetti definendo quali sono i messaggi leciti.

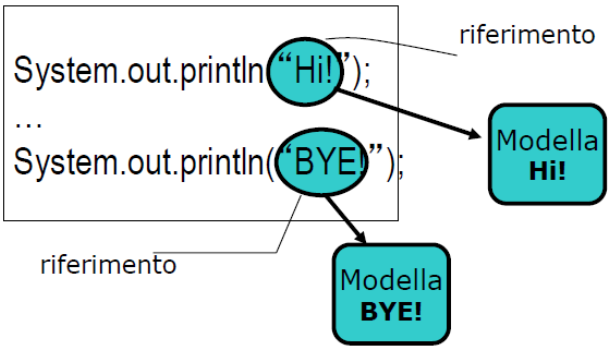
Le classi determinano i messaggi leciti mediante la definizione di metodi:

- Una sezione di codice all'interno di una classe che implementa un particolare comportamento.
- Sono individuati da un nome del metodo.

Overloading: la possibilità di avere una classe che definisca metodi differenti con lo stesso nome.

ES. println("ciao") e println(); //PERCHE IL COMPORTAMENTO E DIVERSO.

String: referenze ed oggetti



"Hi!" e "BYE!" sono due riferimenti a oggetti String che modellano le sequenze di caratteri Hi! e BYE!

Immutabilità: una volta creato un oggetto String NON può cambiare.

Es: l'invio di un messaggio ***toUpperCase*** comporta la creazione di un nuovo oggetto ***String***.

Il metodo costruttore

In Java per creare un oggetto di una classe è necessario eseguire il metodo costruttore. Il costruttore ha sempre lo stesso nome della classe. Esso non deve restituire alcun valore, ma non deve neanche essere dichiarato di tipo **void**.

Solitamente viene dichiarato di tipo **public**. Per compiere le operazioni di inizializzazione vengono spesso usati i parametri.

Classi

Il comportamento di un oggetto è descritto da una classe. Ogni classe ha:

- ***Un'interfaccia pubblica***, insieme di metodi (funzioni) che si possono invocare per manipolare l'oggetto.
- ***Un'implementazione nascosta***, codice e variabili usati per implementare i metodi dell'interfaccia e non accessibili all'esterno della classe.

Coesione:

Una classe deve rappresentare un singolo concetto. Una classe è ***coesa*** se l'interfaccia contiene solo operazioni tipiche del concetto che la classe realizza.

Una classe con bassa coesione fa tante cose insieme, svolgendo molto lavoro "sparso" e non correlato (ha troppe responsabilità). Questo tipo di situazione sarebbe da evitare in quanto queste classi risultano:

- complesse da riutilizzare (***bassa riusabilità***);
- complicate da manutenere (***scarsa manutenibilità***);
- delicate e critiche in quanto soggette a continui cambiamenti (***bassa flessibilità***).

Una forma comune di bassa coesione si ha in quelle classi che presentano un grandissimo numero di ***metodi*** (pubblici o privati).

Accoppiamento:

Una classe A dipende da una classe B se usa esemplari di B (oggetti o metodi di B).

È possibile avere molte classi che dipendono tra di loro (accoppiamento elevato).Problemi dell'accoppiamento elevato:

- Se una classe viene modificata tutte le classi che dipendono da essa potrebbero necessitare di una modifica
- Se si vuole usare una classe in un altro programma bisognerebbe usare anche tutte le classi da cui quella classe dipende

Array e ArrayList

Le principali differenze sono:

1. ***Array*** è una ***struttura di dati a lunghezza fissa***, mentre ***ArrayList*** è una classe ***Collection*** a ***lunghezza variabile***. Non è possibile modificare la lunghezza dell'array una volta creato in Java ma ArrayList si ridimensiona automaticamente quando si riempie in base alla capacità e al fattore di carico. Poiché ***ArrayList*** è supportato internamente da ***Array*** in Java, qualsiasi operazione di ridimensionamento in ***ArrayList*** *rallenterà le prestazioni in quanto comporta la creazione di un nuovo array e la copia del contenuto dal vecchio array al nuovo array*.

2. Non è possibile utilizzare ***Generics*** insieme a ***Array***, poiché l'istanza Array conosce il tipo che può contenere e genererà *ArrayStoreException* , se si tenta di memorizzare un tipo che non è convertibile in tipo di Array. ArrayList ti consente di utilizzare ***Generics*** per garantire la sicurezza dei tipi.

3. Tutti i tipi di array forniscono variabili ***length*** che indicano la lunghezza di ***Array,*** mentre ***ArrayList*** fornisce il metodo ***size()*** per calcolare la dimensione di ***ArrayList*** in Java.

4. Non è possibile archiviare le ***primitive*** in ***ArrayList*** , può contenere solo oggetti. Mentre ***Array*** può contenere sia ***primitive*** che ***Oggetti*** in Java. Per i dati primitivi si utilizzano ***classi wrapper*** (***involucro***). Sebbene ***Autoboxing*** possa dare l'impressione di memorizzare ***primitive*** in ***ArrayList*** , in realtà *converte automaticamente le primitive in Object.*

```
ArrayList < Intero > integerList = new ArrayList < Intero > () ;

integerList. aggiungi ( 1 ) ; // qui non memorizziamo la primitiva in ArrayList, invece l'autoboxing

convertirà la primitiva int in oggetto intero
```

5. Java fornisce il metodo ***add ()*** per inserire elementi in ***ArrayList***, mentre puoi semplicemente usare l'operatore di assegnazione per memorizzare elementi in ***Array.***

```
Object [] objArray = new Object [ 10 ] ;

objArray [ 1 ] = new Object () ;
```

6. È possibile creare ***un'istanza di ArrayList senza specificare le dimensioni***, mentre Java creerà ***Array*** con dimensioni predefinite ma è obbligatorio fornire dimensioni di ***Array*** durante la creazione diretta o indiretta inizializzando Array durante la creazione.

In termini di prestazioni, ***Array*** e ***ArrayList*** forniscono prestazioni simili in termini di tempo costante per l'aggiunta o l'ottenimento di elementi se si conosce l'indice. Sebbene il ridimensionamento automatico di ***ArrayList*** possa rallentare un po' l'inserimento.

Interfacce

Serve per il riutilizzo del codice. Nelle classi il meccanismo può essere lo stesso in tutti i casi, cambiano solo i dettagli. Tutte le classi in questione potrebbero accordarsi su un unico metodo "*getMeasure*" che dia per ogni oggetto il valore da considerare. Il comportamento di getMeasure varia a seconda di ciò che rappresenta realmente l'oggetto. Non è possibile scrivere un'implementazione unica di getMeasure che vada bene per tutti gli oggetti.

Un'interfaccia dichiara una collezione di metodi elencando le loro firme (con tipo del valore restituito) ma non fornisce alcuna implementazione dei metodi.

Es.: l'interfaccia che dichiara il metodo getMeasure è

```
public interface Measurable{
      double getMeasure();
}
```

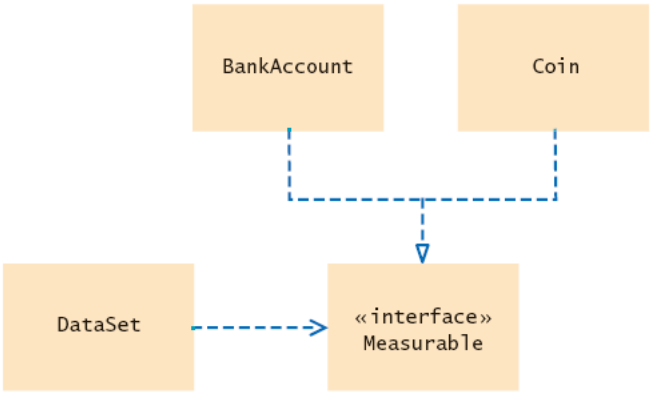
Questa dichiarazione dichiara semplicemente un ***contratto.***

Il contratto ***Measurable*** dice che per essere rispettato da una certa classe c'è bisogno che essa fornisca un metodo di nome getMeasure, senza parametri, che restituisca un double.

Un'interfaccia ***non è una classe***: non si possono creare oggetti di tipo Measurable!

Differenze tra classi e interfacce

1. Tutti i metodi di un'interfaccia sono ***astratti***, cioè non hanno un'implementazione
2. Tutti i metodi di un'interfaccia sono automaticamente ***public*** (non serve lo specificatore d'accesso)
3. Un'interfaccia non ha variabili di istanza, può definire solo costanti
4. Esistono variabili del tipo di un'interfaccia ma non esistono istanze di un'interfaccia
5. Una variabile del tipo di un'interfaccia può contenere istanze delle classi che implementano l'interfaccia



Una o più classi possono realizzare (***implementare***) un'interfaccia se forniscono l'implementazione di tutti i metodi dichiarati nell'interfaccia.

BankAccount b = new BankAccount(100);

Measurable x = b;

BankAccount account = (BankAccount) x;

Non è possibile convertire dal tipo di una classe al tipo di un interfaccia che NON è implementata da quella classe.

È possibile effettuare il casting di un oggetto ad un certo tipo solo se l'oggetto in origine era di quel tipo.

BankAccount b = new BankAccount(100);

Measurable x = b;

Coin c = (Coin) x; */* errore che provoca un'eccezione: il tipo originale dell'oggetto a cui si riferisce x non è Coin ma BankAccount */*

L'operatore ***instanceof*** permette di verificare se un oggetto appartiene ad un determinato tipo

Polimorfismo

Measurable x;

x = new ... (BankAccount OR Coin)

double i = x.getMeasure();

Quale metodo getMeasure viene invocato? Le classi BankAccount e Coin forniscono due diverse implementazioni di getMeasure.

JVM utilizza il metodo getMeasure() della classe a cui si riferisce l'oggetto.

Il metodo ***getMeasure()*** viene detto ***polimorfico*** (multiforme)

Polimorfismo vs Overloading

Entrambi invocano metodi distinti con lo stesso nome, ma:

- Con il polimorfismo avviene in fase di esecuzione
- Con l'overloading scelta del metodo appropriato avviene in fase di compilazione, esaminando il tipo dei parametri

Interfacce di smistamento

Permettono ad una classe di richiamare un metodo prestabilito per ottenere maggiori informazioni:

- Con:

```
public interface Measurable{
        double getMeasure();
}
```

  misurazione demandata all'oggetto stesso
- Con:

```
public interface Measurer{
        double measure(Object anObject);
    // restituisce la misura dell'oggetto anObject
}
```

  misurazione implementata in una classe dedicata (**Interfaccia di smistamento**).

Ereditarietà

Meccanismo di estensione del comportamento (metodi e campi).

Se abbiamo una classe esistente potrebbe non essere desiderabile modificarla perché può implicare una complessità aggiuntiva.

Tutti i metodi e le variabili d'istanza della classe padre sono ereditati automaticamente

La classe ***Object*** è la superclasse di tutte le classi.

In Java le classi sono raggruppate in gerarchie di ereditarietà.

Metodi di una sottoclasse

- ***Sovrascrivere*** metodi della superclasse: la sottoclasse ridefinisce un metodo con la stessa firma del metodo della superclasse e vale il metodo della sottoclasse.
- ***Ereditare*** metodi dalla superclasse: la sottoclasse non ridefinisce nessun metodo della superclasse
- Definire ***nuovi*** metodi: la sottoclasse definisce un metodo che non esiste nella superclasse

Variabili di istanza di sottoclassi

- ***Ereditare*** variabili istanza: Le sottoclassi ereditano tutte le variabili di istanza della superclasse
- Definire ***nuove*** variabili istanza: Esistono solo negli oggetti della sottoclasse, Possono avere lo stesso nome di quelle nella superclasse, ma non sono sovrascritte, Quelle della sottoclasse mettono in ombra quelle della superclasse.

Per invocare il costruttore della superclasse dal costruttore di una sottoclasse uso la parola chiave ***super*** seguita dai parametri del costruttore. Se non viene richiamato invoca il costruttore predefinito della superclasse, cioè quello senza parametri.

Si può salvare un riferimento ad una sottoclasse in una variabile di riferimento ad una superclasse.

Polimorfismo

```
public void transfer(BankAccount other, double amount)
{
    withdraw(amount);
    other.deposit(amount);
}
```

Gli si può passare qualsiasi tipo di ***BankAccount***.

E' lecito passare un riferimento di tipo CheckingAccount a un metodo che si aspetta un riferimento di tipo BankAccount.

BankAccount momsAccount = . . .;

CheckingAccount harrysChecking = . . .;

momsAccount.transfer(harrysChecking, 1000);

l compilatore copia il riferimento all'oggetto harrisChecking di tipo sottoclasse nel riferimento di superclasse other.

Il metodo transfer invoca il metodo deposit.Quale metodo?

Dipende dal tipo reale dell'oggetto (***late binding***).

Fattorizzazione

L'ereditarietà può essere anche usata per spostare un comportamento comune a due o più classi in una singola superclasse.

Ereditarietà vs. Interfacce

- Un'interfaccia non è una classe: non ha uno stato, né comportamento, è un elenco di metodi da implementare
- Una sottoclasse è una classe, ha uno stato e un comportamento che sono ereditati dalla superclasse

Metodi Astratti

Si può considerare il problema di una figura. Ha sempre un'area ma non può essere mai calcolata a priori. Dipende dalla figura presa in considerazione.

```
public abstract class Figura {
   private String colore;
   public Figura(String col) {
      colore = col;
   }
  String getColore() {
      return colore;
   }
  public abstract double getArea();
}

class Rettangolo extends Figura {
   private double altezza, base;
   Rettangolo (String col, double alt, double bas)
      {
      super(col);
      altezza = alt;
      base = bas;
   }
   double getArea() {
      return altezza * base;
   }
}
```

In figura è presente un metodo getArea() impossibile da concretizzare ignorando il tipo di figura.

Si realizza il metodo dichiarandolo **abstract**.

Tutte le sottoclassi devono fornire un'implementazione del metodo per non essere a loro volta astratte.

Una ***classe astratta*** è un ibrido tra classe ed interfaccia ha alcuni metodi normalmente implementati ed altri astratti.

Programmazione generica

Creazione di strutture dati e algoritmi che possono essere utilizzati con tipi di dati diversi. Si può realizzare con:

- ereditarietà (variabili di tipo Object)
- variabili di tipo

Le variabili di tipo di una classe generica:

- sono dichiarate tra parentesi angolari dopo il nome della classe
- di solito sono indicate con una lettera maiuscola
- sono utilizzate per dichiarare le variabili, i parametri dei metodi e il valore di restituzione nel codice della classe

```
public class Pair<S,T>{
 public Pair(S primoEl, T secondoEl){
   primo = primoEl;
   secondo = secondoEl;
 }
 public S getFirst() { return primo; }
 public T getSecond() { return secondo; }
 private S primo;
 private T secondo;
}
```

Pair<Double,Integer> p = new Pair<Double,Integer>(3.0,3);

L'effetto ottenuto è come se le variabili di tipo venissero assegnate con i tipi indicati al momento dell'istanziazione

Per utilizzare un metodo generico, non occorre specificare il tipo effettivo da assegnare alle variabili di tipo.

Il tipo del parametro è dedotto dal compilatore dall'uso che ne facciamo.

Può essere necessario limitare variabili di tipo in modo da non permettere a qualsiasi tipo di dato di poter accedere al metodo generico. Ad esempio si potrebbe fare una cosa del genere:

public static <E extends Comparable> E min(E[] a)

In questo modo E può essere assegnata con un qualsiasi tipo che estende Comparable.

Type erasure

Le variabili di tipo non sono tipi di Java. Ad esempio, se T è una variabile di tipo come in Pair, non troveremo mai T.java oppure T.class nel file system. T non fa parte del nome della classe Pair.

Nella compilazione di Pair si genera il file Pair.class dove non vi è traccia dei parametri T e S (type erasure). La classe viene trasformata nella classe "grezza" corrispondente.

Consente alle applicazioni Java che usano tipi generici di essere compatibili con le librerie e le applicazioni create prima dell'avvento dei tipi generici.

Per effetto della type erasure non vi è modo di risalire a runtime al tipo dell'oggetto che effettivamente viene usato in una classe generica.

Collezioni di dati

- Collection è l'interfaccia che descrive genericamente le funzionalità di una collezione:
  indica i metodi che devono essere presenti in una classe che descrive una generica collezione di oggetti.

- Set è l'interfaccia (estensione di Collection) che descrive le funzionalità di un insieme:
  Non ammette elementi duplicati e non ha nozione di posizione e pone vincoli sui duplicati (non possono esserci).

- List è l'interfaccia (estensione di Collection) che descrive le funzionalità di una lista:
  Ammette duplicati, ha una nozione di posizione e pone vincoli su metodi Collection.

Iteratori

È un oggetto di supporto usato per accedere agli elementi di una collezione, uno alla volta e in sequenza.

Un iteratore è sempre associato ad un oggetto collezione, per funzionare ha bisogno di conoscere la classe implementata dalla collezione.

Debugging

Un programma che esegue il programma sotto collaudo e analizza il suo comportamento a run-time.

Un debugger permette di arrestare e far ripartire l'esecuzione del programma (usando i ***breakpoint***), visualizzare il contenuto delle variabili e eseguire il programma un passo alla volta (single step).

L'esecuzione è sospesa ogni volta che viene raggiunto un **breakpoint**. Quando l'esecuzione si arresta si può, ispezionare le variabili.

Ci sono due varianti del comando ***singlestep*** (esecuzione di una singola linea di codice):

- ***Step Over***: salta le chiamate a metodi
- ***Step Into***: esegue le chiamate a metodi

Eccezioni

Una condizione di errore in un programma può avere molte cause:

- Errori di programmazione: Divisione per zero, cast non permesso, accesso oltre i limiti di un array, …
- Errori di sistema: Disco rotto, connessione remota chiusa, memoria non disponibile, …
- Errori di utilizzo: Input non corretti, tentativo di lavorare su file inesistente, …

In JAVA la gestione degli errori può essere fatta usando il meccanismo delle ***eccezioni***, che sono oggetti che possono essere creati e lanciati (***throw***) in determinate condizioni, e che possono essere catturati (***catch***) dal codice scritto appositamente per la loro gestione:

1. Le eccezioni non devono poter essere trascurate
2. Le eccezioni devono poter essere gestite da uno gestore competente, non semplicemente dal chiamante del metodo che fallisce

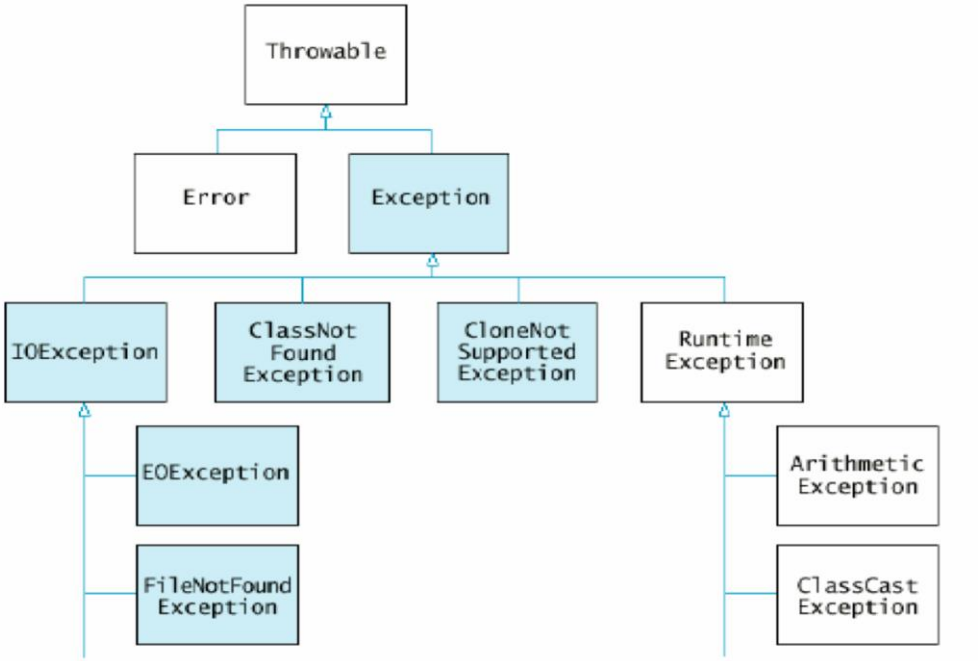
Informalmente, un'eccezione è un'anomalia, di cui sia possibile effettuare il recupero, occorsa durante l'esecuzione di un programma.

Formalmente, un'eccezione è una violazione di vincoli semantici o di risorsa.

Una ***eccezione*** è un evento che interrompe la normale esecuzione del programma.

Se si verifica un'eccezione il metodo trasferisce il controllo ad un ***gestore delle eccezioni***.

La ***superclasse*** di tutti gli errori è la classe ***Throwable*** nel package java.lang. Esiste una gerarchia.



Error: Errori fatali, dovuti a condizioni accidentali.

Exception: Tutti gli errori che non rientrano in Error, i programmi possono gestire o no questi errori a seconda dei casi

Categorie di Eccezioni:

- **Eccezioni non controllate**: (Tutte le sottoclassi di **RunTimeException**)
  Dovute a circostanze che il programmatore **può evitare**, correggendo il programma.
  Esempio:
  NullPointerException: uso di un riferimento null.
  IndexOutofBoundException: accesso ad elementi esterni ai limiti di un array.
  Non è obbligatorio scrivere un codice per gestire questo tipo di eccezione.
  Il programmatore può prevenire queste anomalie, correggendo il codice.

- **Eccezioni controllate**: (Tutte le sottoclassi di **IOException**)
  Dovute a circostanze esterne che il programmatore **non può evitare,** il compilatore vuole sapere cosa fare nel caso si verifichi l'eccezione.
  Esempio:
  EOFException: terminazione inaspettata del flusso di dati in ingresso.
  Può essere provocata da eventi esterni: errore del disco, interruzione del collegamento di rete.
  Il gestore dell'eccezione si occupa del problema.

Ogni eccezione deve essere gestita, altrimenti causa l'arresto del programma.

Per installare un gestore si usa l'enunciato **try**, seguito da tante clausole **catch** quante sono le eccezioni da gestire.

A volte vogliamo eseguire altre istruzioni prima dell'arresto. La clausola ***finally*** viene usata per indicare un'istruzione che va eseguita sempre.

Il package java.io e i flussi

In Java input e output sono definiti in termini di flussi (stream): Sequenze ordinate di dati

Due tipi di flussi:

- Di dati binari (byte stream)
- Di caratteri (character stream)

Ciascun tipo di flusso è gestito da apposite classi

Flussi di Oggetti consentono di operare su interi oggetti:

- Per scrivere un oggetto non dobbiamo prima decomporlo
- Per leggere un oggetto non dobbiamo leggere i dati separatamente e poi ricomporre l'oggetto

Flussi in ***scrittura***: Classe ObjectOutputStream

Flussi in ***lettura***: Classe ObjectInputStream

La memorizzazione di oggetti in un flusso viene detta ***serializzazione.***

Grafica

Un ***Frame*** è una finestra dotata di barra di titolo e cornice. La Java Virtual Machine esegue ogni frame su un ***thread*** separato. Un ***thread*** è un flusso di esecuzione, processo, visione dinamica di un programma sequenziale.

Per modificare figure bisogna modificare ***JComponent***. In sintesi:

1. Costruire un frame
2. Costruire una componente:
3. Aggiungi la componente al frame
4. Rendi il frame visibile

Gestione degli eventi

Un ***Evento*** è una conseguenza di un azione dell'utente. Ci sono 2 tipologie:

Ricevitore dell'evento (***listener***):

- Riceve una notifica quando un evento accade
- I suoi metodi descrivono le azioni da eseguire quando si verificano gli eventi
- Un programma sceglie gli eventi da trattare installando un ricevitore per ciascuno di essi

Sorgente dell'evento (***source***):

- La componente (dell'interfaccia utente) che ha generato l'evento
- Quando capita un evento, la sorgente notifica tutti i ricevitori dell'evento

1. ***Cosa sono le interfacce (+esempio)***
2. ***Cosa sono le interfacce di smistamento?(+ esempio)***
3. ***Differenza Array e ArrayList***
4. ***Programmazione generica***
5. ***Espressioni Lambda***
6. ***Ereditarietà E polimorfismo sia per le interfacce che per I ereditarietà***

Scritto Java

- **boolean equals(Object otherObject):** Verifica se l'oggetto è uguale a un altro.

```
public boolean equals(Object otherObject){
    if (otherObject instanceof Coin){
    Coin other = (Coin)otherObject;
    return name.equals(other.name) && value == other.value;
    }else return false;
}
```

- **Object clone():** Crea una copia dell'oggetto.

```
public Object clone(){
        BankAccount cloned= new BankAccount();
        cloned.balance = balance;
        return cloned;
}
```

- **String toString():** Restituisce una rappresentazione dell'oggetto in forma di stringa.

```
public String toString(){
        return "BankAccount[balance=" + balance + "]";
}
```

Classe Random

La classe ***Random*** della libreria Java permette di generare numeri casuali, cioè produce numeri arbitrari. Contiene i seguenti metodi:

- ***nextInt(n)*** restituisce un numero intero casuale compreso fra 0 (incluso) e n (escluso);
- ***nextDouble()*** restituisce un numero in virgola mobile casuale compreso fra 0 (incluso) e 1 (escluso).

Esempio:

```
import java.util.Random;

public class Dice
{       public static void main(String[] args)
   {    Random generator = new Random();
        // getta il dado dieci volte

        for (int i = 1; i <= 10; i++)
        {  int d = 1 + generator.nextInt(6);
           System.out.print(d + " ");
        }
        System.out.println();
   }
}
```

Ordinamento

È possibile ordinare un ***ArrayList*** utilizzando il metodo **sort ()** della classe **Collections.** Questo metodo accetta un oggetto list come parametro e ne ordina il contenuto in ordine crescente. Ci sono a disposizione 2 interfacce:

1. ***java.lang.Comparable*** di solito si utilizza per definire ***l'ordinamento "naturale" di un oggetto*** e consta di un metodo con firma:

int compareTo(T o), che paragona l'oggetto corrente (this) con l'oggetto fornito in input al metodo (o).

Esempio:

```
public class Persona implements Comparable<Persona> {
    public Persona(String nome, String cognome, int eta){
        this.nome=nome;
        this.cognome=cognome;
        this.eta=eta;
    }
    private String nome, cognome;
    private int eta;
    @Override
    public int compareTo(Persona o) {
        return this.cognome.compareTo(o.cognome);
}   }
```

2. L'interfaccia ***java.util.Comparator*** si utilizza invece quando si vogliono definire ***ordinamenti alternativi*** dell'oggetto. Si ***crea*** dunque una ***classe a parte*** che implementa l'interfaccia con il seguente metodo:

public int compare(T a,T b);

Esempio:

```
import java.util.Comparator;
public class PersonaEtaComparator implements Comparator<Persona> {
    @Override
    public int compare(Persona p1, Persona p2) {
        int retVal=0;
        if(p1.getEta()>p2.getEta()){
            retVal=1;
        }else if(p1.getEta()<p2.getEta()){
            retVal=-1;
        }
        return retVal;
}   }
```

Metodi String

- Il metodo ***replace*** esegue un'operazione di ricerca e sostituzione in una stringa:

  river.*replace*("issipp", "our"); // costruisce una nuova stringa ("Missouri")

- Ritorna il carattere in posizione i-esima:

  char c=s.charAt(0); // c ==”p”

- Ritorna l□indice della prima occorrenza del carattere indicato

  int i=s.indexOf(“o”); // i == 3

- Il ***contains()***controlla se una stringa contiene una sequenza di caratteri. Restituisce ***true*** se i caratteri esistono e in ***false*** caso contrario.

```
String myStr = "Hello";
  System.out.println(myStr.contains("Hel"));  // true
  System.out.println(myStr.contains("Hi"));   // false
```

Gestione delle data

Bisogna ricordare che il numero dei mesi, a differenza di quello che si fa di solito, parte da 0.

GregorianCalendar data1 = new GregorianCalendar(2008, 11, 18);

Per ottenere informazioni sull'ANNO, MESE, GIORNO e ORA si utilizza il metodo get() passandogli come parametro le costanti statiche definite nella classe (YEAR, MONTH, DAY_OF_MONTH,…).

```
int anno = dataAttuale.get(GregorianCalendar.YEAR);

int mese = dataAttuale.get(GregorianCalendar.MONTH) + 1;        //i mesi partono da 0

int giorno = dataAttuale.get(GregorianCalendar.DAY_OF_MONTH);

int ore = dataAttuale.get(GregorianCalendar.HOUR);

int minuti = dataAttuale.get(GregorianCalendar.MINUTE);

int secondi = dataAttuale.get(GregorianCalendar.SECOND);
```

Confronti tra date:

```
GregorianCalendar data1 = new GregorianCalendar(2008, 11, 18);

GregorianCalendar data2 = new GregorianCalendar(2007, 11, 10);

if (data1.before(data2)){

System.out.println("data 1 precede data 2"); }

else if (data1.after(data2)) {

System.out.println("dati 2 precede data 1"); }

else { System.out.println("Le date sono uguali"); }
```

File

Flussi in scrittura: Classe ***ObjectOutputStream***

L'oggetto da inserire nel flusso deve essere serializzabile altrimenti viene sollevata la ***NotSerializableException***

- Appartenere a una classe che implementa l'interfaccia Serializable
- Serializable non ha metodi

```
MyClass mc = new MyClass(…);

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("mc.dat"));

out.writeObject(mc);                                    //MyClass implementa Serializable
```

Flussi in lettura: Classe ***ObjectInputStream***

Legge un ***Object*** da file, restituisce un riferimento a tale ***Object***

- L`output necessita di un cast
- Può lanciare un'eccezione controllata di tipo ***ClassNotFoundException***

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("mc.dat"));

MyClass mc = (MyClass) in.readObject();
```

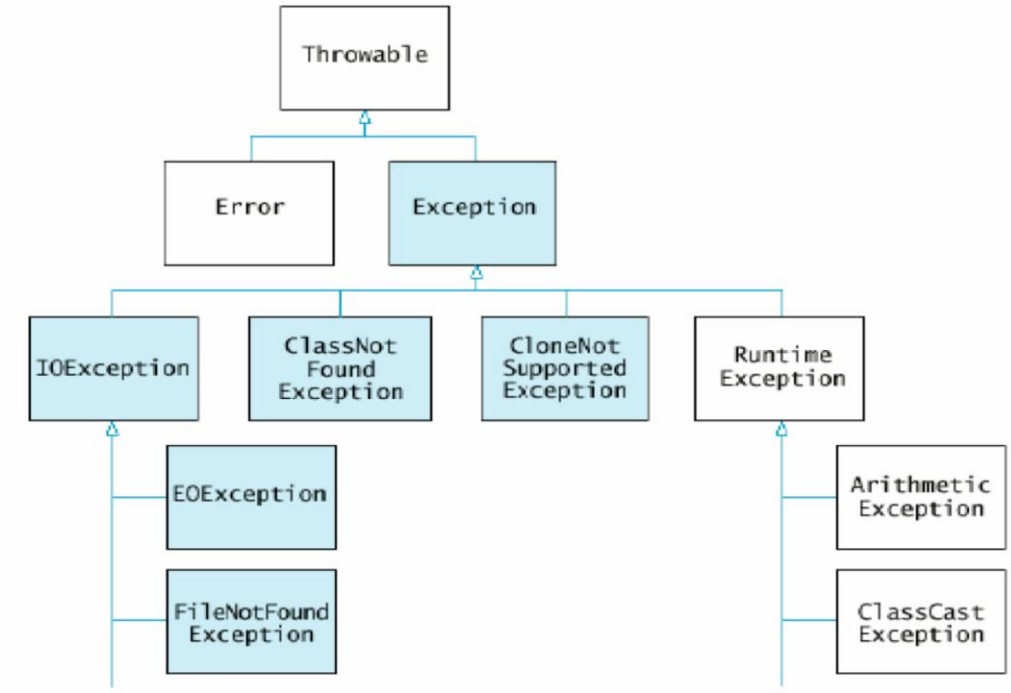
Esempio:

```
public class ObjectInputStreamExample {
   public static class Person implements Serializable {
      public String name = null;
      public int age  =  0;
   }
   public static void main(String[] args) throws IOException, ClassNotFoundException {
      ObjectOutputStream objectOutputStream = new ObjectOutputStream(new FileOutputStream("person.txt"));
      Person person = new Person();
      person.name = "Jakob Jenkov";
      person.age  = 40;
      objectOutputStream.writeObject(person);
      objectOutputStream.close();
      ObjectInputStream objectInputStream = new ObjectInputStream(new FileInputStream("person.txt"));
      Person personRead = (Person) objectInputStream.readObject();
      objectInputStream.close();
      System.out.println(personRead.name);
      System.out.println(personRead.age);
```

Categorie di Eccezioni

- **eccezioni non controllate**: (Tutte le sottoclassi di **RunTimeException**)
  l dovute a circostanze che il programmatore **può evitare**, correggendo il programma.
  Esempio:
  NullPointerException: uso di un riferimento null.
  IndexOutofBoundException: accesso ad elementi esterni ai limiti di un array.
  Non è obbligatorio scrivere un codice per gestire questo tipo di eccezione.
  Il programmatore può prevenire queste anomalie, correggendo il codice.

- **eccezioni controllate**: (Tutte le sottoclassi di **IOException**)
  dovute a circostanze esterne che il programmatore **non può evitare,** il compilatore vuole sapere cosa fare nel caso si verifichi l'eccezione.
  Esempio:
  EOFException: terminazione inaspettata del flusso di dati in ingresso.
  Può essere provocata da eventi esterni: errore del disco, interruzione del collegamento di rete.
  Il gestore dell'eccezione si occupa del problema.



Per lanciare un`eccezione, usiamo la parola chiave **throw** (lancia), seguita da un oggetto di tipo eccezione:

throw exceptionObject;

Syntax `throw exceptionObject;`

Example

```
if (amount > balance)
{
   throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

Most exception objects can be constructed with an error message.

A new exception object is constructed, then thrown.

This line is not executed when the exception is thrown.

Le eccezioni di runtime (**RuntimeException**) possono essere utilizzate per segnalare problemi dovuti ad input errati.

Syntax

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
```

Example