JAVA

Object-Oriented-Programmation (OOP)

È un paradigma di programmazione che prevede di raggruppare in un'unica entità (la classe) sia le strutture dati che le procedure che operano su di esse, creando per l'appunto un "oggetto" software dotato di proprietà (dati) e metodi (procedure) che operano sui dati dell'oggetto stesso.

- **Classe** => concetto stratto, definisce le caratteristiche comuni di un insieme di oggetti
  ex. persona
- **Oggetti** => istanza di una classe
  ex. mario
- **Istanza** => oggetto allocato di una classe

Inoltre ha 4 concetti base:

- **Incapsulamento dei dati** => le informazioni interne sulla definizione delle classi vengono nascoste e rese visibili solo tramite determinate interfacce apposite
- **Ereditarietà** => le classi sono definite secondo una gerarchia, ognuna ha caratteristiche comuni al padre estese con caratteristiche proprie
- **Astrazione** => Il meccanismo con cui si specifica le caratteristiche peculiari di un oggetto che lo differenzia da altri
- **Polimorfismo** => possibilità di creare funzioni con lo stesso nome ma che eseguono istruzioni diverse a seconda del caso

Programma Java

1. Si crea file di testo con codice sorgente per ogni classe:
   *nomeClasse***.java**
2. compilazione
   Javac *nomeClasse1***.java** *nomeClasse2***.java**
3. esecuzione
   Java *nomeClasseMain*

Importare codice librerie

Importare libreria completa:

Import Java.*nomePacchetto***.*;**

Importare solo classe:

Import Java.*NomePacchetto***.***nomeClasse***;**

Invocazione metodi

- Metodo su istanza, permette modificarne variabili e stato:

    *nomeIstanza***.***nomeMetodo***(** *parametri***);**

- Metodo statico, ovvero metodo non associato ad una istanza ma direttamente alla classe, la quale permette di effettuare operazioni su altre variabili e/o oggetti:

    *NomeClasse***.***NomeMetodo***(** *parametri***);**

Ex. system.out.println(…) => metodo che permette di scrivere su terminale

Sintassi e tipi

Java è un linguaggio fortemente tipizzato ed usa stessa sintassi e tipi primitivi di C, i tipi complessi (equi. struct) sono salvati mediante la creazione di oggetti, definita come:

NomeClasse NomeVariabile **=** **new** *nomeClasse***(** *parametri***)**;

Anche le stringhe sono salvate come oggetti, tramite:
String *nomeStringa* **= "***stringa***";**

È possibile convertire una stringa in intero tramite la classe integer:
int *nomeVariabile* **= Integer.ParseInt(** *stringa***);**

Il passaggio inverse si effettua tramite:
String *nomeNuovaVariabile* **=** *nomeVecchiaVariabile***.toString();**

Array e matrici

Un oggetto array si definisce con:

*tipo***[]** *nomeArray* **= new** *tipo***[** *lunghezza***];**

Definizione di una matrice:

*tipo***[][]** *nomeArray* **= new** *tipo***[** *righe***][** *colonne***];**

Accesso a array, come in C:
*nomeArray***[** *indice***];**

Per calcolare la lunghezza dell'array si usa l'attributo:
*nomeArray***.length**

OSS le stringhe sono gli unici oggetti che non necessitano del new

Garbage collection

Operazione di recupero della memoria deallocando oggetti che hanno perso il riferimento e non sono più usati dal programma. Svolta a run-time dal sistema.

Modificatori variabili e metodi Java

Si usano per modificare visibilità e proprietà aggiuntive a variabili e metodi, secondo l'espressione regolare: [1|2|3] [4] [5] *tipo*

1. **Public** => varibile/metodo è accessibile in ogni classe del codice
2. **Private** => accessibile solo all'interno della classe in cui è dichiarata, permette di avere incapsulamento delle variabili, che non possono quindi essere visualizzate all'esterno della classe a meno di getter o setter
3. **Protected** => accessibile in classi dello stesso package
   default => solo nello stesso package
4. **Final** => costante, non è possibile modificarne il valore
5. **Static** => in comune in tutte le istanze della classe

Definizione classe

Ogni classe è formata da 2 parti:

- **Campi dati** => variabili che formano gli attributi della classe
- **Campi operazione** => metodi della classe

  Se la classe può essere allocata necessita di un costruttore, con le proprietà:
  - Non può essere static
  - Ha lo stesso nome della classe
  - Non ha un tipo di ritorno

```
public class nomeClasse{
        //campi dati
        private tipo nomeVariabile1;
        private tipo nomeVariabile2;
        …

        //costruttore
        public nomeClasse( tipo va1, tipo var2, …){
                nomeVariabile1 = va1;
                nomeVariabile2 = var2;
                …
        }
        …

        //metodi
        …
}
```

THIS
Permette di identificare in modo univoco le variabili di istanza

```
public class Persona {
    // variabili di istanza (campi dati)
    private String nome;
    private int eta;

    // costruttore
    public Persona(String n, int e) {
        nome = n;
        eta = e;
    }
    ...
}
```

Getters and setters

Permettono di estrarre valori di variabili private delle classi.

```
Public tipo getNomeVariabile(){
        return nomeVariabile;
}

Public tipo setNomeVariabile( tipo var){
        this.NomeVariabile = var;
}
```

Ereditarietà

Permette di creare sottoclassi che consentono di estendere la classe genitore con attributi e metodi aggiuntivi, o modificando quelli già esistenti.
Si definiscono con:

Class *NomeClasse* **extends** *classeGenitore* **{**… **}**

OSS non si ha ereditarietà multipla, ovvero al massimo un genitore

Dentro il costruttore è inoltre possibile richiamare il costruttore della classe genitore che effettua automaticamente tutte le assegnazioni già definite, con:

super(*var1, var2, …***);**

OSS è possibile assegnare oggetti appartenenti alle sottoclassi a variabili della classe genitore, ma non il contrario, poiché la classe figlio ha più attributi e metodi.

Override e polimorfismo

Consistono nel modificare un metodo già definito:
- Override => il metodo ridefinito nella classe figlio usa stesso nome e parametri ma esegue istruzioni differenti
- Polimorfismo => il metodo viene ridefinito nella stessa classe usando stesso nome ma parametri differenti

La classe Object

Tutte le classi derivano dalla classe Object e sono compatibili con essa, inoltre ne ereditano tutti gli attributi ed i metodi, tra cui:
- **toString()** => che permette di convertire oggetto in stringa
- **getClass()** => restituisce classe dell'oggetto
- **clone()** => che restituisce copia dell'oggetto
- **equals(Object** *O***)** => permette di vedere se 2 oggetti sono uguali

Nella realtà tutti questi metodi dovrebbero essere ridefiniti in ogni classe, poiché versione standard ritorna risultati errati

La classe Class

Gli oggetti della classe class corrispondono alle classi definite dai programmi, permettono di stabilire a run-time la classe a cui appartiene l'oggetto:

ex. **Boolean** *x* **= (** *A*.**getClass().equals(** *B*.**getClass()));**
ritorna true se sia A che B sono della stessa classe

Funzione IsIstance

Funzione della classe class, restituisce true se il parametro è un oggetto della classe su cui viene usato il metodo:

```
ex. class B extends class A
        A a1 = new A();       B b1 = new B();
        A.class.isIstance(b1);        // true
        B.class.isIstance(a1);        // false
```

ATTENZIONE Vale anche su oggetti di classi figlie

Classi astratte

È una classe non istanziabile contente solo dichiarazioni di metodi abstract utilizzata come base per la definizione di classi derivate, si definisce con:

abstract class *nomeClasse* **{**… **}**

I metodi della classe si definiscono con:

abstract *tipo nomeMetodo***(** …**);**

Le sottoclassi della classe astratta devono implementare tutti i sui metodi

Interfacce

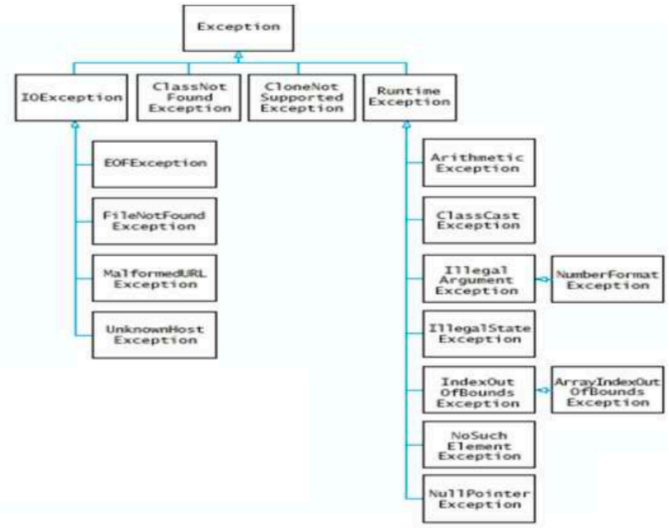
Un interfaccia è un astrazione che definisce un insieme di funzioni solo dichiarate, le quali devono essere definite nella classe che la implementa.
Si usa la sintassi:

class *nomeClasse* **implements** *nomeInterfaccia1, nomeInterfaccia2, …* **{** …**}**

OSS al contrario dell'ereditarietà una classe può implementare più interfacce insieme
ATTENZIONE le funzioni delle interfacce non hanno il modificatore abstract

Eccezioni

Classi che rappresentano errori non fatali gestibili dal programmatore



Tipi di eccezioni

Le eccezioni possono essere di 2 tipi:

- **Controllate** => circostanze esterne che non possono essere gestite dal programmatore
  ex. Sottoclassi di IOException
- **Non controllate** => circostanze da evitare, correggibili da programmatore
  ex. Sottoclassi di RunTimeException

Catturare eccezioni

Costrutto che permette di gestire le eccezioni che possono essere generate dal codice:

```
Try {
        // blocco contente istruzioni
        // che possono generare eccezioni
} catch( nomeException e){
        // blocco contenete istruzioni
        // da eseguire in caso di exception
} finally {
        // istruzioni da eseguire in ogni caso
        // non obbligatoria
}
```

OSS usando nel catch
e.**printStackTrace();**
È possibile stampare cause e codice errore

Lanciare eccezioni

È possibile lanciare exceptions nelle funzioni tramite:

throw *nomeException***;**

ex. If (amount > balance)
throw new illegalArgumentException(“saldo insufficiente”);

Segnalare eccezioni

Quando si utilizzano funzioni che possono lanciare exception bisogna segnalarlo nell’intestazione della funzione, tramite:

*tipo nomeFunzione***(** *tipo var1, tipo var2, …***) throws** *NomeException* **{** … **}**

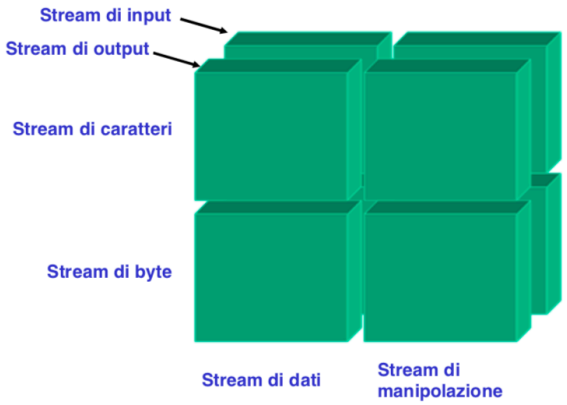
OSS è possibile creare nuove eccezioni creando nuove classi che estendono la classe **throwable**

Import java.io;

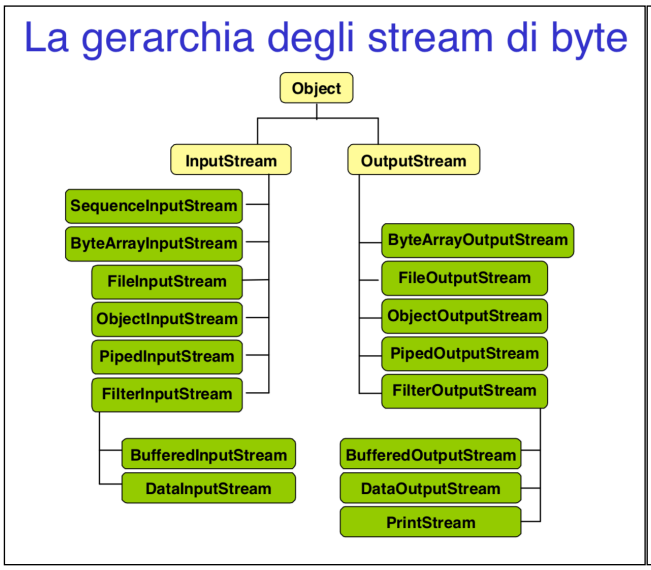
I/O streams

In java tutte le comunicazioni I/O sono gestite dagli stream, che:

- Sono unidimensionali:
  - Stream input
  - Stream output
- Agiscono su 2 tipi di dati:
  - Stream di byte
  - Stream di caratteri (unicode, 16 bit)
- Possono avere 2 diversi scopi:
  - Stream di dati
  - Stream di manipolazione
    (effettuano elaborazioni su altri stream)



OSS tutte le classi degli stream sono fatte per essere incastrata una con l'altra



I/O standards

Stream standard della classe system, sono 2 attributi static che gestiscono l'input da tastiera e l'output video:

- **System.in**
- **System.out**

OSS sono entrambi stream di byte e non di caratteri

Input da tastiera

Essendo system.in uno stream di byte e non di caratteri, per la lettura bisogna prima effettuare la conversione dei byte in caratteri tramite lo stream di manipolazione InputStreamReader, per poi passarli a BufferdReader che crea la stringa, si ha quindi:

BufferedReader *nomeBuffer* **= new BufferedReader(**
new InputStreamReader(system.in);

Una volta creato il buffer contenente la stringa, questa si può leggere tramite:

String *nomeStringa* **=** *nomeBuffer***.readLine();**

Output video

In questo caso pur avendo uno stream di byte non non si creano problemi, si può usare direttamente system.out.print(), oppure si può incapsulare nello stream di manipolazione:

PrintWriter *nomeWriter* **= new PrintWriter(system.out);**

Per poi stampare tramite:

*nomeWriter***.println();**
*nomeWriter***.flush();** //flush serve per ripulire il buffer

ATTENZIONE l'utilizzo di questi metodi generare eccezioni controllate
OSS per effettuare input / output su canali differenti da tastiera e video basta cambiare l'argomento delle classi stream, impostando il canale da utilizzare

File

Sono rappresentati tramite la classe File, anche in questo caso le operazioni di input / output si effettuano tramite gli stream.

ex. creazione file

```
import java.io.File;
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
    try {
      File file = new File("C:/myfile.txt");

      if(file.createNewFile())System.out.println("Success!");
      else System.out.println ("Error, file already exists.");
    }
    catch(IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

ex. lettura file

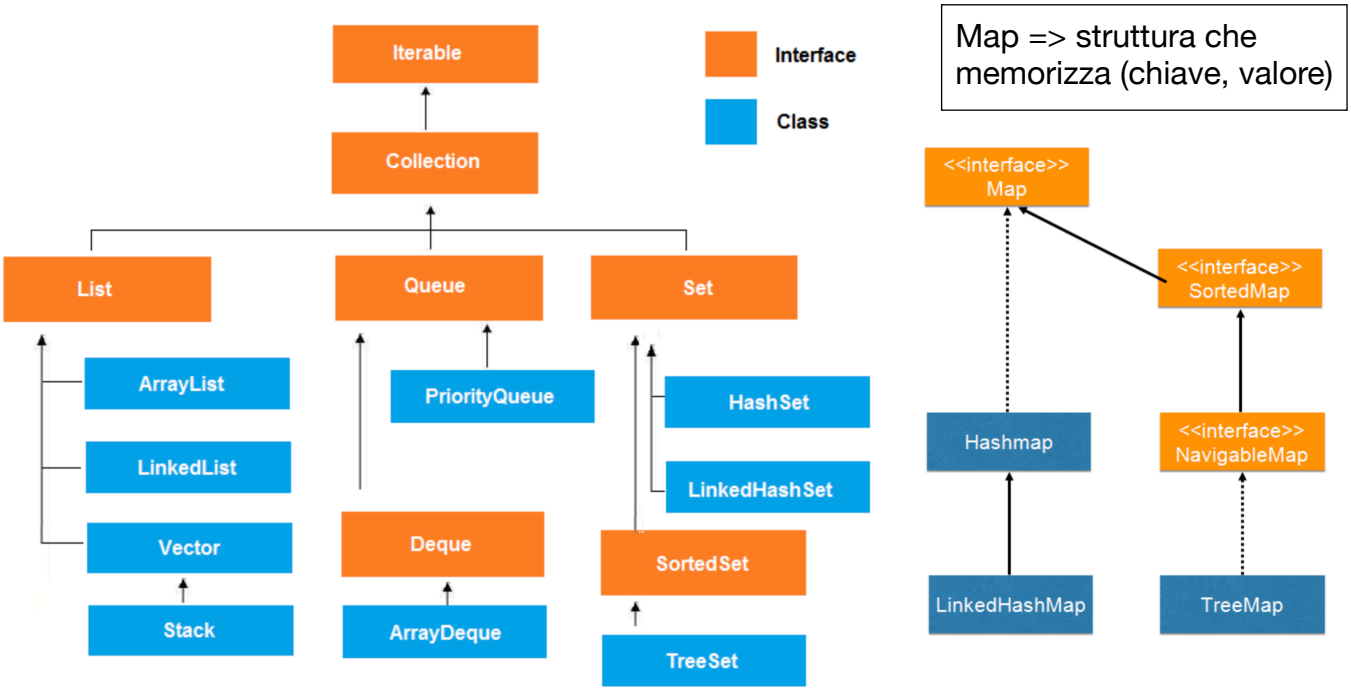
```
import java.io.*;

public class Main {
  public static void main(String[] args)  {
    try {
      BufferedReader in = new BufferedReader(new FileReader("c:\\filename"));
      String str;

      while ((str = in.readLine()) != null) {
        System.out.println(str);
      }
      System.out.println(str);
    } catch (IOException e) {
    }
  }
}
```

Java Collection Framework (JCF)

Il JCF è una libreria formata da un insieme di interfacce e classi che le implementano per lavorare con collezioni di oggetti



Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Generazione collezione vuota:

Collection<*classe*> *nomeCollezione* = **new** *ClassColletion*<*classe*>**():**

ex. List<persona> persone = new ArrayList<persona>();

Tipi generici

I tipi generici sono usati in Java per definire classi, interfacce e metodi parametrici rispetto ad un tipo di dati su cui operano.
Per la definizione di un tipo generico C rispetto ad un tipo T si usa la notazione <T> nella dichiarazione della classe C, ovvero le istanze di C saranno definite rispetto al tipo T.

```
Class C <T> {
        …
}
```

ex. C<string> c1 = new C<string>(“ciao”);

Interfaccia Collection

```
public interface Collection<E> {
        int size();
        boolean isEmpty();
        boolean contains(Object element);
        boolean add(E element); // Optional
        boolean remove(Object element); // Optional
        Iterator<E> iterator();
        boolean equals(Object o);

        // Bulk operations
        boolean containsAll(Collection<?> c);
        boolean addAll(Collection<? extends E> c); // Optional
        boolean removeAll(Collection<?> c); // Optional
        boolean retainAll(Collection<?> c); // Optional
        void clear(); // Optional

        // Array Operations
        Object[] toArray();
        <T> T[] toArray(T[] a);
}
```

Di conseguenza le interfacce e classi sottostanti implementeranno tutti questi metodi più altri propri

OSS set non aggiunge nessun metodo

Interfaccia list

```
public interface List<E> extends Collection<E> {
        boolean add(int index, E element); // Optional
        E get(int index);
        E set(int index, E element); // Optional
        int indexOf(Object o);
        int lastIndexOf(Object o);
        boolean remove(int index);

        ListIterator<E> listIterator();
        ListIterator<E> listIterator(int index)

        boolean addAll(int index, Collection<? extends E> c); // Optional
        List<E> subList(int fromIndex, int toIndex);
}
```

Interfaccia Iterator

```
public interface Iterator<E> {
        boolean hasNext();
        E next();
        void remove(); // Optional
}

public interface ListIterator<E> extends Iterator<E> {
        boolean hasPrevious();
        E previous();
        int nextIndex();
        int previousIndex();
        void set(E e); // Optional
}
```

Per scandire una collezione tramite iterator bisogna prima creare un iteratore associato alla collezione:

iterator<*T***>** *nomeIteratore* **=** *nomeCollezione*.**Iterator();**

```
Collection<E> c = ...          // collezione di oggetti di tipo E
…

Iterator<E> it = c.iterator();  // iteratore per la collezione c
while (it.hasNext()) {
        E e = it.next();        // poni l'elemento corrente in e ed avanza
        ...                     // processa l'elemento corrente (denotato da e)
}
```

Interfaccia Map

```
public interface Map<K,V> {
        int size();
        boolean isEmpty();
        void clear(); // Optional
        boolean equals(Object o);

        boolean containsKey(Object key);
        boolean containsValue(Object value);
        V get(Object key);
        V put(K key, V value); // Optional
        V remove(Object key); // Optional

        Set<K> keySet();
        Collection<V> values();
}
```

Collezioni ordinate e interfaccia comparable

Esistono due modi per definire collezioni ordinate:

- SortedSet => insiemi ordinati, niente ripetizioni
- SortedMap => mappe ordinate per chiave, ammette ripetizioni

L'ordinamento è stabilito dalle interfacce:

```
public interface Comparable<T> {
        Int compareTo(T o);
}
```

```
public interface Comparator<T> {
        int compare(T o1, T o2);
}
```

I metodi ritornano:

-1 => o1 è più piccolo di o2
0 => sono uguali
1 => 01 è più grande di o2

OSS In compareTo o1 corrisponde a this e o2 a o

Threads

In java i threads corrispondo ad istanze della classe Thread, le quali svolgono la funzione di interfaccia con JVM che crea effettivamente i threads.

Implementazione

1. Creare la classe contenente le istruzioni che implementa l'interfaccia **Runnable** ed è formata da:
   - Costruttore (opzionale, solo se bisogna passare variabili)
   - Metodo **run{ …}** che contiene istruzioni
   - Eventuali metodi di supporto a run

2. Creare istanze di Thread ella funzione che genera i threads, con:

   Thread *nomeThread* **= new Thread(new** *ClasseRunnable***(** *variabiliCostruttore***);**

   una volta creata l'istanza si avvia il thread tramite l'istruzione:

   nomeThread.**start();**

   OSS currentThread() ritorna quale thread correntemente in uso

```
public class MiaClasseRunnable implements Runnable {
        private String nome;

        public MiaClasseRunnable(String n) {
                nome = n;
        }

        public void run() {
                for (int i = 0; i < 10; i++) {
                        System.out.println(nome + ": " + i);
                }
                System.out.println(nome + ": DONE! ");
        }
}
```

```
public class Main {
        public static void main(String[] args) {
                MiaClasseRunnable ja = new MiaClasseRunnable("Jamaica");
                Thread mioThread = new Thread(ja);
                mioThread.start();
        }
}
```

Sleep

Metodo che permette di fermare l'esecuzione di un thread per un determinato periodo di tempo, permettendo di generare intervalli nella sua esecuzione, sintassi:

Thread.sleep(*millisecondi***);**

```
public void run() {
   for (int i=0;i<100;i++){
      System.out.print(c);
      try {
            Thread.sleep((long)(Math.random() * 10));
      } catch (InterruptedException e) {}
   }
}
```

Join

Si usa per sincronizzare l'esecuzione di più thread, ovvero la funzione chiamante si blocca ed aspetta la fine dell'esecuzione dei thread su cui è stato lanciato join prima di continuare con le sue istruzioni.

```
public class MainConcorrenteJoin {
    public static void main(String[] args) {
            CountDown h = new CountDown("Huston", 10);
            CountDown c = new CountDown("CapeCanaveral", 10);
            Thread t1 = new Thread(h);
            Thread t2 = new Thread(c);
            t1.start(); // NB invoco h.run() sul thread t1
            t2.start(); // NB invoco c.run() sul thread t2
            try {
                    t1.join();
           t2.join();
            } catch (InterruptedException e) {
                    // qui va codice da eseguire
           // se il main viene risvegliato da un interrupt
                    // prima di aver fatto join
            }
            System.out.println("Si parte!!!");
    }
}
```

- Huston: 10
- CapeCanaveral: 10
- Huston: 9
- CapeCanaveral: 9
- Huston: 8
- CapeCanaveral: 8
- Huston: 7
- CapeCanaveral: 7
- Huston: 6
- CapeCanaveral: 6
- Huston: 5
- CapeCanaveral: 5
- Huston: 4
- CapeCanaveral: 4
- Huston: 3
- CapeCanaveral: 3
- Huston: 2
- CapeCanaveral: 2
- Huston: 1
- CapeCanaveral: 1
- Huston: Go!
- CapeCanaveral: Go!
- Si parte!!!

OSS senza i join() il main sarebbe eseguito concorrentemente ai 2 threads, quindi si avrebbe il "si parte" subito dopo il 10 e non alla fine

Synchonized

Modificatore da applicare ai metodi, impone che solo un thread alla volta può usare il metodo.
Si utilizza su metodi che eseguono side-effect su strutture dati del sistema

Import java.net

Sockets

Permettono la comunicazione tramite la rete