Basi

Import **Java.lang** è di default. Questo package contiene in pratica tutte le basi, (String, i wrapper, le exception…). Nel package **java.util** invece ci sono le strutture dati, (ArrayList, HashMap, Set…). Nel **java.time** ci sono LocalDate, Duration, Year…

Il comando **javac** : es. javac file.java legge il contenuto del programma, scritto in un file di testo con estensione .java. Se non ci sono errori, lo trasforma in un bytecode (linguaggio intermedio tra linguaggio macchina e linguaggio di programmazione) e lo salva in un file con estensione .class. Per eseguire il programma in java devo richiamare il bytecode con l'interprete java tramite il comando **java**, quindi scriverò es. java file. La JVM permette la platform indipendence e si occiupa di gestire la memoria e traduce istruzioni java in istruzioni macchina.

Le variabili di istanza e di classe hanno un valore di default quando dichiarate. Quelle locali no. Quindi se dichiari in un metodo String x; System.out.print(x); oppure una variabile wrapper e la stampi darà errore di compilazione. Attenzione agli array, solo loro, anche i locali, hanno sempre un valore di default in base al tipo, per esempio un array new int[3] avrà 3 celle inizializzate e 0, un array di Integer invece 3 celle a null, quindi 0 se è un tipo primitivo, 0.0 se double e . se char, oppure null se è un oggetto. Le variabili di istanza o di classe di tipo stringa quindi avranno null.

Nella condizione del while ci deve essere SEMPRE un valore booleano, se ho while(x = 3) il programma **NON** compila. Infatti sto scrivendo in pratica while(3) che non significa nulla.

Se il main non è scritto correttamente: public static (final facoltativo) void main(String… s (oppure String[] s)) ci sarà un errore **a tempo di esecuzione**. Il **varargs String…** deve essere un parametro, unico nel metodo e sempre l'ultimo dei parametri. Inoltre se in un metodo scrivi s.length() non va bene, perché s è un array, non una stringa!!! Le variabili statiche sono usabili da qualsiasi metodo, ma quelle di istanza possono essere usate solo nei metodi di istanza, nei metodi statici sono usabili SOLO tramite un'istanza. Stessa identica cosa per i metodi, i metodi statici possono essere chiamati da qualsiasi metodo, ma i metodi di istanza devono essere chiamati nei metodi di istanza oppure negli statici tramite un'istanza.

{ qualcosa } all'interno di una classe è detto inizializzatore di istanza. Static { qualcosa } invece inizializzatore statico. Date due classi A e B extends A allora l'ordine di visita è: inizializzatori statici (da superclasse a sottoclasse), inizializzatori di istanza della superclasse, locale superclasse, inizializzatori di istanza della sottoclasse, locale sottoclasse.

Il metodo toString() lo hanno **TUTTI** gli oggetti. L'ereditarietà consente di sovrascrivere un metodo in una sottoclasse, eventualmente modificando il comportamento atteso di altri metodi in una superclasse. Consente agli oggetti di ereditare attributi e metodi comunemente usati. Non è vero che è preferibile allo static, è una scelta stilistica.

Posso avere delle variabili che si chiamano come classi wrapper. int Integer è consentito, il contrario invece no. Per i metodi invece parseInt() restituisce il tipo primitivo mentre valueOf() restiutisce un Wrapper!

Un numero lo puoi dividere con dei underscores (_), l'importante è che non siano all'inizio né alla fine del numero, per esempio 2_9_454 è consentito, ma 123_5_.00 oppure 123.00_ no! double d = new Double(070_090.050_06); d vale 70090.05006

Either of operands significa **UNO DEGLI OPERANDI**.

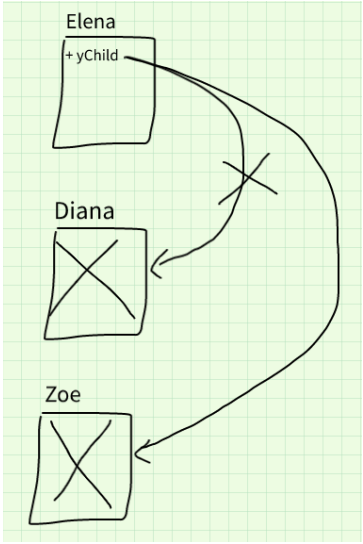
Il metodo **finalize()** appartiene alla classe Object. Subito prima di chiudere un oggetto, il garbage collector si assicura che non ci siano più riferimenti ad esso e chiama il metodo finalize() su di esso. Pertanto, una volta sovrascritto il metodo **finalize()** al suo interno, puoi eseguire tutte le attività di pulizia come chiudere le risorse come la connessione al database, la connessione di rete, ecc.

```
protected void finalize throws Throwable{}
```

Viene richiamato una sola volta durante l'esecuzione di un programma, oppure 0 (se il programma crasha).

Non puoi forzare l'esecuzione del garbage collection in un certo punto del programma, con System.gc() puoi **suggerire** alla JVM di eseguirlo, ma essa può anche non farlo. Se la JVM dovesse accettare il suggerimento allora andrebbe nel metodo **public void finalize() una singola volta passando tutte le variabili create precedentemente**, attenzione perché nel metodo finalize() ci va sempre alla fine del programma.

```
1:  public class Person {
2:     public Person youngestChild;
3:
4:     public static void main(String... args) {
5:        Person elena = new Person();
6:        Person diana = new Person();
7:        elena.youngestChild = diana;
8:        diana = null;
9:        Person zoe = new Person();
10:       elena.youngestChild = zoe;
11:       zoe = null;
12:    }
13: }
```



In questo esempio Sulla riga 9, tutti e tre gli oggetti hanno riferimenti. L'oggetto diana è referenziato tramite l'oggetto elena. Alla riga 10, il riferimento all'oggetto diana è sostituito da un riferimento all'oggetto zoe. Pertanto, solo l'oggetto diana è idoneo per la garbace collected.

Le API (application programming interface) sono dei metodi già esistenti che puoi usare a piacimento per i tuoi scopi, per esempio le API della classe Integer sono valueOf, parseInt… ecc

Un metodo che restituisce qualcosa anche se c'è if(x>10) {return true} else if(x<=10) {return false} deve avere un return finale, il compilatore non capisce che andrà per forza in uno dei blocchi!

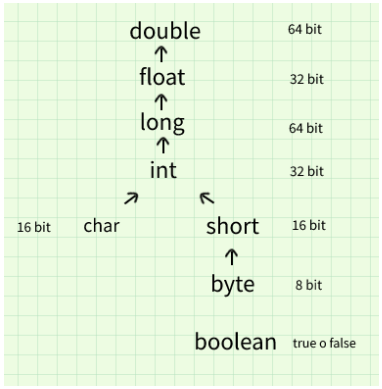
Switch

Switch accetta nei vari case i tipi primitivi (**byte**, **short**, **char** e **int**) e le **stringhe** (istanze della classe String). E accetta nei suoi execution path solo **espressioni letterali** (es. case 3:) oppure variabili costanti **final** (inizializzate ovviamente), considerando final int x = 5 puoi scrivere per esempio **case 5:** oppure **case x:** , ma se la x non fosse final **non puoi usarla** nel case. Ovviamente il case deve essere univoco, non puoi avere case 5: e case x: (con x che vale 5). Attenzione ai tipi, perché se un case è una stringa allora gli altri case non possono essere di altri tipi. E se ho switch(int) allora non posso avere case "stringa" perché la stringa non può essere convertita in intero, per i 4 tipi consentiti invece non c'è alcun problema di casting, funziona ogni combinazione.

Primitivi

I numeri con la virgola o con la d o la D sono double! Per i float si scrive 3f o 3F, per i long 3l o 3L, puoi scrivere char c = 65, quel 65 ok che è un intero ma è come se stessi scrivendo c = 'A'. Grazie all'autoboxing e all'autounboxing puoi avere mettere int in Integer e viceversa senza acuna conversione. Il metodo byteValue(T x) è della classe Integer e converte x in un byte, se x è un carattere allora va bene grazie alla ASCII, ma se è la stringa "a" allora errore a runtime, "5" invece tutto ok.

I primitivi possono essere convertiti implicitamente (è java che lo fa) secondo questa scala.




```java
public static void main(String[] args) {
    byte b1 = 7;        //non esiste un modo per rappresentare un numero byte, quindi quel 7 è visto come byte
    Byte b2 = 7;        //autoboxing
    short s1 = 7;       //non esiste un modo per rappresentare un numero short, quindi quel 7 è visto come short
    Short s2 = 7;       //autoboxing
    char c1 = 7;        //grazie al codice ascii il 7 è visto come un carattere
    Character c2 = 7;   //grazie al codice ascii il 7 è visto come un carattere, autoboxing
    int i1 = 7;         //intero -> intero
    Integer i2 = 7;     //intero, autoboxing
    long l1 = 7;        //intero -> long
    Long l2 = 7;        //ERRORE, se ci fosse 7l allora quel 7 sarebbe visto come long e ci sarebbe stato l'autoboxing
    float f1 = 7;       //intero -> float
    Float f2 = 7;       //ERRORE, dovrebbe essere 7f
    double d1 = 7;      //intero -> double
    Double d2 = 7;      //ERRORE, dovrebbe essere 7d oppure 7.0


    byte b3 = Short.parseShort("7");        //ERRORE, short non può essere convertito in byte
    Byte b4 = Short.parseShort("7");        //ERRORE, short non può essere convertito in Byte
    short s3 = Short.parseShort("7");       //ok
    Short s4 = Short.parseShort("7");       //ok
    char c3 = Short.parseShort("7");        //ERRORE, short non può essere convertito in char
    Character c4 = Short.parseShort("7");   //ERRORE, short non può essere convertito in Character
    int i3 = Short.parseShort("7");         //ok
    Integer i4 = Short.parseShort("7");     //ERRORE, short non può essere convertito in Integer
    long l3 = Short.parseShort("7");        //ok
    Long l4 = Short.parseShort("7");        //ERRORE, short non può essere convertito in Long
    float f3 = Short.parseShort("7");       //ok
    Float f4 = Short.parseShort("7");       //ERRORE, short non può essere convertito in Float
    double d3 = Short.parseShort("7");      //ok
    Double d4 = Short.parseShort("7");      //ERRORE, short non può essere convertito in Double
```

Puoi però convertire qualsiasi tipo in un altro, basta usare il casting esplicito!!! Attenzione ai wrapper e al metodo **parse()**, esso restituisce un tipo puro (primitivo) di quel tipo, per esempio se avessi char x = Integer.parseInt("7"); avrei un errore perché un int non può andare in char. char c = 7 funziona perché quel 7 è visto come un carattere codificato nella tab ascii. Posso scrivere byte b = 'x'; oppure posso scrivre short s = 'x'; questo perché quel carattere x è come se lo stessi definendo un carattere a 8 bit, o a 16 bit, è come se fosse un vero e proprio byte o short. Se invece provassi a fare byte b = c; oppure short s = c; con char c = 'x'; avrei un errore! Il metodo **valueOf()** invece restituisce un wrapper. Le somme sono consentite, avremo sempre il tipo più grande, per esempio (byte) 2 + 1 sarà la somma di un byte e un intero quindi restituirà 3 intero, oppure (char) 65 + 1 darà 66 intero, anche se (char) 65 da solo sarebbe 'A'.

```java
char c = 'x';
byte b1 = 'x';      //OK
byte b2 = c;        //ERRORE, non è possibile convertire un carattere in byte
short s1 = 'x';     //OK
short s2 = c;       //ERRORE, non è possibile convertire un carattere in short
```

Operatori

Gli && o || sono applicabili solo ai booleani, idem per & e |. I secondi sono i classici operatori logici tra booleani, quindi true & false darà false, true & true darà true ecc. && invece confronta la prima espressione, se è vera confronta anche la seconda. || confronta la prima, se è vera non ha bisogno di confrontare la seconda. L'operatore XOR, ^ , da vero quando i due booleani in analisi sono discordi.

Ecco il corretto ordine dal più alto di precedenza

operatori postfissi	`[] . (params) expr++ expr--`
operatori unari	`++expr --expr +expr -expr !`
instanze e cast	`new (type)expr`
aritmetici	`* / %`
aritmetici	`+ -`
di relazione	`< > <= >=`
uguaglianza	`== !=`
AND	`&&`
OR	`||`
condizionale	`? :`
assegnamento	`= += -= *= /= %= &= |=`

```
int plan = 1;
int x;

//Basta ricordare che se c'è un post incremento o decremento si considera solo il primo
//Se c'è un pre incremento o decremento allora al secondo operatore la variabile già sarà cambiata
//x = plan++ + --plan;    //1 + 0 e poi incrementa = 2
//x = plan++ + ++plan;    //1 + 2 e incrementa = 4
//x = plan-- + --plan;    //1 + 0 e decrementa = 0
//x = plan-- + ++plan;    //1 + 2 e poi si decrementa = 2

//x = plan++ + plan--;    //1 + 1 e poi incrementa = 3
//x = plan++ + plan++;    //1 + 1 e poi incrementa = 3
//x = plan-- + plan--;    //1 + 1 e poi decrementa = 1
//x = plan-- + plan++;    //1 + 1 e poi decrementa = 1

//x = ++plan + --plan;    //2 + 1 = 3
//x = ++plan + ++plan;    //2 + 3 = 5
//x = --plan + ++plan;    //0 + 1 = 1
//x = --plan + --plan;    //0 + -1 = -1

//x = ++plan + plan--;    //2 + 2 = 4
//x = ++plan + plan++;    //2 + 2 = 4
//x = --plan + plan++;    //0 + 0 = 0
//x = --plan + plan--;    //0 + 0 = 0
```

JavaBeans

I metodi JavaBeans ben strutturati sono

1. public T getT();
2. public void setX(T x);
3. public boolean isX();

Period

E' una classe **immutabile** (cioè che crea oggetti che non cambiano direttamente, come le stringhe devi usare il risultato di un metodo, non puoi cambiare l'oggetto stesso) nel pacchetto **java.time**, e permette di creare periodi di date, anni, mesi, settimane e giorni. Ha vari metodi tra cui of(int years, int months, int days), ofDays(int days), ofMonths(int months), ofWeeks(int weeks) e ofYears(int years), restituiscono un oggetto period, per esempio Period.ofWeeks(2) restituisce P14D. Period.of(1,30,500) -> P1Y30M500D

Predicate

E' un'interfaccia funzionale nel package **java.util.function** che ha un metodo boolean test(T t).

LocalDate

Classe immutabile nel **java.time** package. Per creare un'istanza devi usare il metodo of(int anno, int mese, int giorno) non puoi scrivere new LocalDate()

`static LocalDate`	`of(int year, int month, int dayOfMonth)` Obtains an instance of LocalDate from a year, month and day.
`static LocalDate`	`of(int year, Month month, int dayOfMonth)` Obtains an instance of LocalDate from a year, month and day.

LocalDateTime

Classe immutabile nel **java.time** package. Per creare un'istanza devi usare i metodi of(). Es LocalDate a = LocalDateTime.of(2012, 6, 30, 12, 00); non puoi scrivere new LocalDateTime()

static LocalDateTime	of(int year, int month, int dayOfMonth, int hour, int minute) Obtains an instance of LocalDateTime from year, month, day, hour and minute, setting the second and nanosecond to zero.
static LocalDateTime	of(int year, int month, int dayOfMonth, int hour, int minute, int second) Obtains an instance of LocalDateTime from year, month, day, hour, minute and second, setting the nanosecond to zero.
static LocalDateTime	of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond) Obtains an instance of LocalDateTime from year, month, day, hour, minute, second and nanosecond.
static LocalDateTime	of(int year, Month month, int dayOfMonth, int hour, int minute) Obtains an instance of LocalDateTime from year, month, day, hour and minute, setting the second and nanosecond to zero.
static LocalDateTime	of(int year, Month month, int dayOfMonth, int hour, int minute, int second) Obtains an instance of LocalDateTime from year, month, day, hour, minute and second, setting the nanosecond to zero.
static LocalDateTime	of(int year, Month month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond) Obtains an instance of LocalDateTime from year, month, day, hour, minute, second and nanosecond.
static LocalDateTime	of(LocalDate date, LocalTime time) Obtains an instance of LocalDateTime from a date and time.

LocalTime

Classe immutabile nel **java.time** package. Per creare un'istanza devi usare i metodi of(). L'unità più piccola: nanosecondi. Le classi Date e Time sono immutabili, quindi NON hanno metodi set.

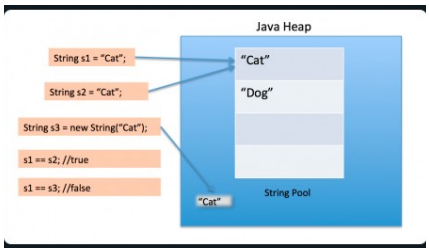
static LocalTime	of(int hour, int minute) Obtains an instance of LocalTime from an hour and minute.
static LocalTime	of(int hour, int minute, int second) Obtains an instance of LocalTime from an hour, minute and second.
static LocalTime	of(int hour, int minute, int second, int nanoOfSecond)

DateTimeFormatter

Classe immutabile **nel java.time.format.*** package. I metodi ofPattern("MM-dd-yyyy"); e format(LocalDate) sono i più importanti. Se nella ofPatterns c'è mm, che sono i minuti, e la LocalDate è senza minuti allora avremo un'eccezione a Runtime, **UnsupportedTemporalTypeException**.

Stringhe

Le stringhe sono final, immutabili, nel senso che se faccio String a = new String("mela"); a.concat(" rossa"); l'oggetto a non cambia, i metodi restituiranno **un altro oggetto**, infatti il concat restituisce un nuovo oggetto con il nuovo valore, **NON MODIFICA** a. Dovrei fare a = a.concat("rossa"). Se scrivo String x = 'a'; avrò un errore, ci vogliono per forza le virgolette. Ecco un diagramma che spiega chiaramente come viene mantenuto lo String Pool nello spazio java heap e cosa succede quando usiamo modi diversi per creare stringhe. L'operatore == confronta il riferimento all'oggetto, mentre il metodo equals della classe Object confronta il contenuto. Per la lunghezza si usa il metodo length(), da non confondere con l'attributo .length degli array.



s1 == s2 è true, s1 == s3 è false, s1.equals(s2) è true, s1.equals(s3) è true. Se scrivessi String s4 = s3 allora i due sarebbero lo stesso oggetto.

Uniche ad avere il metodo startsSwith(String) e il metodo replace(String,String) che può avere come parametri anche due char oppure due StringBuilder. substring(int) che restituisce la stringa a partire da quella posizione fino alla fine, substring(int inizio, int lunghezza) restituisce la stringa a partire da inizio di quella lunghezza. Il toUpperCase() restituisce una NUOVA stringa maiuscola, quindi se scrivessi s1.toUpperCase() == s1.toUpperCase() avrei false.

Metodi

char	**charAt**(int index) Returns the `char` value at the specified index.

int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
int	**codePointBefore**(int index) Returns the character (Unicode code point) before the specified index.
int	**codePointCount**(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this `String`.
int	**compareTo**(**String** anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(**String** str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(**String** str) Concatenates the specified string to the end of this string.
boolean	**contains**(**CharSequence** s) Returns true if and only if this string contains the specified sequence of char values.
boolean	**contentEquals**(**CharSequence** cs) Compares this string to the specified `CharSequence`.
boolean	**contentEquals**(**StringBuffer** sb) Compares this string to the specified `StringBuffer`.
boolean	**endsWith**(**String** suffix) Tests if this string ends with the specified suffix.
boolean	**equals**(**Object** anObject) Compares this string to the specified object.
boolean	**equalsIgnoreCase**(**String** anotherString) Compares this `String` to another `String`, ignoring case considerations.
static **String**	**format**(**String** format, **Object**... args) Returns a formatted string using the specified format string and arguments.
byte[]	**getBytes**() Encodes this `String` into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
void	**getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array.
int	**hashCode**() Returns a hash code for this string.
int	**indexOf**(int ch) Returns the index within this string of the first occurrence of the specified character.
String	**intern**()

	Returns a canonical representation for the string object.
boolean	**isEmpty**() Returns `true` if, and only if, **length()** is `0`.
int	**lastIndexOf**(int ch) Returns the index within this string of the last occurrence of the specified character.
int	**length**() Returns the length of this string.
boolean	**matches**(**String** regex) Tells whether or not this string matches the given **regular expression**.
int	**offsetByCodePoints**(int index, int codePointOffset) Returns the index within this `String` that is offset from the given `index` by `codePointOffset` code points.
boolean	**regionMatches**(boolean ignoreCase, int toffset, **String** other, int ooffset, int len) Tests if two string regions are equal.
String	**replace**(**CharSequence** target, **CharSequence** replacement) Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
String	**replaceAll**(**String** regex, **String** replacement) Replaces each substring of this string that matches the given **regular expression** with the given replacement.
String	**replaceFirst**(**String** regex, **String** replacement) Replaces the first substring of this string that matches the given **regular expression** with the given replacement.
String[]	**split**(**String** regex) Splits this string around matches of the given **regular expression**.
boolean	**startsWith**(**String** prefix) Tests if this string starts with the specified prefix.
CharSequence	**subSequence**(int beginIndex, int endIndex) Returns a new character sequence that is a subsequence of this sequence.
String	**substring**(int beginIndex) Returns a new string that is a substring of this string.
char[]	**toCharArray**() Converts this string to a new character array.
String	**toLowerCase**() Converts all of the characters in this `String` to lower case using the rules of the default locale.
String	**toString**() This object (which is already a string!) is itself returned.
String	**toUpperCase**()

	Converts all of the characters in this `String` to upper case using the rules of the default locale.
String	**trim**() Returns a copy of the string, with leading and trailing whitespace omitted.
static **String**	**valueOf**(**Object** obj) Returns the string representation of the `Object` argument.

Array

Sintassi: int[] x = new int[3]; creo un array di 3 elementi settati di default a 0. Se invece avessi creato un array di 3 Integer allora avrei avuto 3 elementi settati di default a null, sono degli oggetti. Le parentesi quadre possono essere o prima o dopo il nome della variabile. Se c'è un'inizializzazione allora NON serve la dimensione, anche {} vuote sono un'inizializzazione. NON è un tipo primitivo, quindi le conversioni int -> double ecc **NON** possono esistere. Nel senso che non posso scrivere double a = new int[3]; e non posso scrivere int[] x = new Integer[] però posso scrivere double a = new double[] {1,2,3}, essi saranno 1.0,2.0 e 3.0.

Ovviamente passato in un metodo esso cambia anche fuori da esso, essendo passato ovviamente per riferimento. Se ho un metodo con varargs **public void metodo(String… s)** posso passarci una stringa e un array, invece nel metodo **public void metodo(String[] s)** posso passarci SOLO un array. La lunghezza si prende con x.length. Per usare i metodi devi usare la classe Arrays, i più usati sono Arrays.sort() e Arrays.binarySearch() per ordinare e cercare rispettivamente e Arrays.toString(). La binarySearch(), con array ordinato, se non trova l'elemento allora restituisce l'indice negativo di dove dovrebbe stare -1. Arrays.asList(array) invece trasforma l'array in una lista, quindi se prima per esempio nel print avevi [Ljava.lang.String;@7960847b ora avrai [null, null]

Gli array **bidimensionali** invece si definiscono: String [][] x = new String[dim][]; la dimensione (dim >= 0) è obbligatoria per la prima parentesi se non c'è l'inizializzazione, ovviamente se poi vuoi stampare l'elemento [0][0] ti darà NullPointerException. Se invece dichiari x = new String[3][0]; puoi farlo ma se provi a stampare l'elemento [0][0] ti darà indexOutOfBoundsException, stessa cosa se non metti dimensioni e inizalizzi con {}. Attenzione agli array bidimensionali perché l'inizalizzazione è { {"mela","pera"..},{..} } (doppie parentesi). Se ho String [][][] x, y; allora entrambe saranno array a tre dimensioni, se avessi String [][] x, y[]; allora x sarebbe di 2 dim e y di 3 dim; Non posso avere String [][] x, [] y;

Se nel metodo c'è String… s e non passo nulla allora ho un array vuoto, quindi se faccio Arrays.toString(s) mi stampa [].

Gli errori a tempo di compilazione vengono prima di quelli a tempo di esecuzione, quindi se ho int[][]x; x[2][1] = 3; x[1][1] = "ciao" avrò l'errore a tempo di compilazione Stringa convertita in intero, il NullPointerException non parte perché non compila.

Liste

Oggetti nel java.util.* e sono mutabili. Puoi scrivere List x = new <tutte le sottoclassi di List>, non puoi fare new List in quanto è un'interfaccia. ArrayList è una classe che la implementa, ArrayList<String> l = new ArrayList<>(1); stai creando una lista di un elemento, però è una struttura che si espande man mano che aggiungi elementi, quindi sei fai 3 add avrà 3 elementi. Per la lunghezza si usa il metodo size().

Quindi l'operatore diamante può essere omesso da entrambe le parti, se messo a sinistra però deve contenere un tipo, a destra invece può essere anche vuoto. Se non definisci quindi un tipo è come se stessi creando una lista di Object quindi la lista può contenere tutti i tipi di oggetto, per esempio sia String che interi!

Attenzione al metodo remove() perché c'è il metodo remove(int index) e quello remove(Object o), se la lista è di interi allora userà di default il metodo con l'index!!!

Quando converti un array in una lista attenzione perché **non puoi aggiungere né rimuovere valori**, puoi solo modificarli (con set(int index, T nuovoValore)).

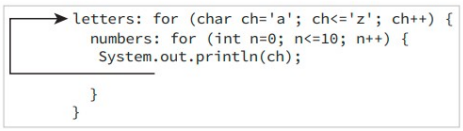
- String[] array = {"Natural History", "Science", "Art"};
- List museums = Arrays.asList(array);

Se provo a fare museums.add(String) oppure una remove() avrò una UnsupportedOperationException, quindi un'eccezione a Runtime, non un errore a tempo di compilazione.

Cicli

Nel for migliorato(for each) non puoi scorrere un array dalla fine al primo elemento e richiede degli oggetti iterabili, per esempio int[], Integer[], ArrayList<String> vanno bene, ma String o StringBuilder no. Ogni for each può essere convertito in altri cicli, viceversa non vale.

```
break;
break letters;
break numbers;
```



Esistono anche le labels, delle etichette, esse **NON** possono chiamarsi come una parola chiave in java, per esempio non può esistere la label for, while oppure la label int, la label Integer invece è consentita, Integer non è una parola chiave, e infatti puoi chiamare una variabile Integer.

Con il break seguito dal nome della label si decide di terminare quel ciclo, quindi se dovessimo seguire la freccia nel disegno dovremmo usare **break**, che ci fa uscire dal ciclo interno e quindi salire alla letters, oppure **break numbers** che fa esattamente la stessa cosa, **break letters** non va bene in quanto ci fa scendere, ci fa semplicemente uscire dal ciclo esterno. Se ci fosse **continue letters** allora sarebbe un'altra risposta valida.

StringBuilder:

Si preferisce per creare le stringhe perché riduce il numero di oggetti creati, risparmiando memoria. La sintassi è StringBuilder s = new StringBuilder("parola"); attenzione però perché non è un oggetto di tipo String, quindi non puoi confrontarlo con una stringa in nessun modo. Per fare un confronto sul contenuto puoi usare il toString() e applicare l'equals. Non puoi scrivere StringBuilder s = "ciao"; L'equals tra due StringBuilder non funziona come le stringhe, quindi anche se hanno lo stesso valore darà false se non converti in String. Oppure possiamo avere StringBuilder sb1 = new StringBuilder("a"); StringBuilder sb2 = sb1; in questo caso i due sarebbero lo stesso oggetto, quindi sia == che equals darebbero true.

A differenza delle stringhe hanno il metodo delete(), reverse(), insert(int index, String s), replace(int,int,String) e son oggetti **MUTABILI**, con il metodo append("prova") puoi aggiungere pezzi alla stringa interna dell'oggetto stesso. E questi metodi restituiscono un riferimento all'oggetto stesso, quindi se fai:

- StringBuilder a = new StringBuilder("mela"); e StringBuilder b = a.append(" rossa");

a e b conterranno la stringa "mela rossa" e punteranno allo stesso oggetto, quindi a == b darebbe true!!!

```java
public class Shoot {
 public static void main(String[] args) {
    StringBuilder sb1 = new StringBuilder("mela");
    StringBuilder sb2 = sb1.append(" rossa");

    System.out.println(sb1);
    System.out.println(sb2);
    System.out.println(sb1 == sb2);
    System.out.println(sb1.toString().equals(sb2.toString()));

    String s1 = new String("mela");
    String s2 = s1.concat(" rossa");

    System.out.println(s1);
    System.out.println(s2);
    System.out.println(s1 == s2);
    System.out.println(s1.equals(sb2));
 }
}
```

esult
mpiled and executed in 0.943 sec(s)

```
mela rossa
mela rossa
true
true
mela
mela rossa
false
false
```

Attenzione ai metodi **substring(int) e substring(int,int)** che restituiscono una **NUOVA** stringa.

Il metodo **insert** funziona in questo modo: StringBuilder s = new StringBuilder("mela"); s.insert(2,"rossa"); produce “merossala”, attenzione perché se scrivi StringBuilder s = new StringBuilder().insert(s.length(),”rossa”) avrai un errore dato che s non è stata ancora inizializzata a quel punto. Avrai un errore anche se dichiari s e subito dopo provi a fare insert(1,”ciao”), dato che s ancora è vuota puoi mettere solo 0.

Il metodo **delete** si usa con 2 parametri (int inizio, int fineNonInclusa), quindi s.delete(1,1) non fa nulla, s.delete(1,2) cancello la e. fineNonInclusa può anche sforare la lunghezza di s. Se inizio è < alla fineNonInclusa allora avremo una StringIndexOutOfBoundExc

Metodi

StringBuilder	append(boolean b) Appends the string representation of the boolean argument to the sequence.
StringBuilder	appendCodePoint(int codePoint) Appends the string representation of the codePoint argument to this sequence.
int	capacity() Returns the current capacity.
char	charAt(int index) Returns the char value in this sequence at the specified index.
int	codePointAt(int index) Returns the character (Unicode code point) at the specified index.
int	codePointBefore(int index) Returns the character (Unicode code point) before the specified index.
int	codePointCount(int beginIndex, int endIndex) Returns the number of Unicode code points in the specified text range of this sequence.
StringBuilder	delete(int start, int end) Removes the characters in a substring of this sequence.

StringBuilder	deleteCharAt(int index) Removes the char at the specified position in this sequence.
void	ensureCapacity(int minimumCapacity) Ensures that the capacity is at least equal to the specified minimum.
void	getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) Characters are copied from this sequence into the destination character array dst.
int	indexOf(String str) Returns the index within this string of the first occurrence of the specified substring.
StringBuilder	insert(int offset, boolean b) Inserts the string representation of the boolean argument into this sequence.
int	lastIndexOf(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
int	length() Returns the length (character count).
int	offsetByCodePoints(int index, int codePointOffset) Returns the index within this sequence that is offset from the given index by codePointOffset code p
StringBuilder	replace(int start, int end, String str) Replaces the characters in a substring of this sequence with characters in the specified String.
StringBuilder	reverse() Causes this character sequence to be replaced by the reverse of the sequence.
void	setCharAt(int index, char ch) The character at the specified index is set to ch.
void	setLength(int newLength) Sets the length of the character sequence.
CharSequence	subSequence(int start, int end) Returns a new character sequence that is a subsequence of this sequence.
String	substring(int start) Returns a new String that contains a subsequence of characters currently contained in this character se
String	toString() Returns a string representing the data in this sequence.
void	trimToSize() Attempts to reduce storage used for the character sequence.

Lambda:

Java supporta la programmazione funzionale (functional programming) utilizzando le espressioni lambda. Un'espressione lambda è un breve blocco di codice che accetta parametri e restituisce un valore. Le espressioni lambda sono simili ai metodi, ma non necessitano di un nome e possono essere implementate direttamente nel corpo di un metodo. Utilizzano **l'esecuzione posticipata** e possono essere eseguite altrove. **import java.util.function.*;**

Esempio:

- Predicate<StringBuilder> p1 = (StringBuilder b) -> {return true;};
- Predicate<StringBuilder> p2 = (StringBuilder b) -> true;
- Predicate<StringBuilder> p3 = b -> true;
- Predicate<StringBuilder> p4 = (b) -> true;

Attenzione perché il tipo a destra dell'uguale deve essere compatibile con quello del Predicate, e la variabile **NON** deve già essere definita. Quindi se nel main ci fosse (String… b) avremmo un errore. Il Predicate non è detto che debba avere un tipo nell'operatore diamante, in quel caso sarebbe considerato di tipo Object. Predicate dash = c -> c.startsWith("-"); Questa darebbe errore in quanto c.startsWith() è un metodo delle Stringhe, non degli Object. Poi con dash puoi usare il metodo test(<T>) per eseguire i confronti. È un'interfaccia e il suo metodo restituisce un boolean. Questo è un esempio e stampa -5

```
public class PrintNegative {
  public static void main(String[] args) {
      List<Integer> list= new ArrayList<>();
      list.add(-5);
      list.add(0);
      list.add(5);
      print(list, e -> e < 0);
  }
  public static void print(List<Integer> list, Predicate<Integer> p) {
     for (Integer num : list)
         if (p.test(num))
             System.out.println(num);
  }
}
```

Classi

I costruttori sono quei metodi che permettono di creare un'istanza di quella classe, se la classe si chiama A allora il costruttore di default è public A() {}, che c'è implicitamente, ma ATTENZIONE, se tu dichiari un nuovo costruttore, per esempio con un parametro, allora questo va a SOVRASCRIVERE quello senza parametri, quindi è come se non esistesse più, devi scriverlo esplicitamente.

Variabili di istanza **final** DEVONO essere inizializzate dove dichiarate, nel costruttore o in un inizializzatore di istanza. Le variabili di classe **final** invece devono essere inizializzate dove dichiarate oppure in un inizializzatore statico. Il programma altrimenti non compilerebbe.

Solo all'interno dei costruttori puoi usare il metodo this(parametri), che **DEVE** essere la prima cosa nel costruttore, questo metodo va a chiamare il costruttore con quei parametri, quindi ovviamente deve esistere tale costruttore, attenzione perché se chiami this() all'interno del costruttore senza parametri ci sarà un errore dato che vai in loop, e questo errore lo avrai ovviamente anche se crei un loop tra i vari costruttori. Esiste anche il metodo super() che va a chiamare il costruttore della superclasse, anch'esso deve essere dichiarato come prima riga nel costruttore, non può esistere quindi un costruttore che chiama sia this() che super().

Visibilità: Senza nulla sarebbe Package-private, il metodo e/o attributo è visibile solo nelle classi nello stesso pacchetto. Protected invece rende visibile il metodo e/o attributo anche in classi in pacchetti differenti purchè siano una sottoclasse dell'altra.

	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
`public`	+	+	+	+	+
`protected`	+	+	+	+	
no modifier	+	+	+		
`private`	+				

+ : accessible
blank : not accessible

I metodi **NON statici**, **NON private** e **NON final** sono chiamati **virtual methods**, e quindi permettono di essere overridati consentendo il polimorfismo. Attenzione perché se un metodo è void allora non posso scrivere System.out.println(metodo()).

Attenzione ai metodi, se ho la variabile di classe public static int x; e il metodo print(int x) { x++} NON sto andando a modificare la variabile di classe!!! Sto andando semplicemente a modificare quella variabile locale, dovrei usare this.x++

Le variabili statiche sono sempre disponibili a tutte le istanze della classe. In un metodo statico, oppure in un blocco inizializzatore statico, non possono essere usate variabili di istanza. Infatti nel main non posso chiamare variabili di istanza senza servirmi dell'istanza.

In Java ogni cosa è passata per valore, questo significa che se passi un oggetto in un metodo allora puoi cambiare gli attributi di quell'oggetto, ma non l'oggetto stesso. Le stringhe, i booleani, gli interi, uguale, non cambiano in un metodo, gli array se li passi in un metodo puoi cambiare il valore in una cella, ma se in un metodo metti array = null, NON diventa null nel main. Le StringBuilder essendo oggetti se fai la append("mela") cambia anche nel main, è come se stessi cambiando un attributo. Gli oggetti sono dei puntatori, quindi sfrutti il puntatore per modificare ciò che vuoi, ma non puoi modificare l'oggetto stesso.

Per favorire l'encapsulation gli attributi devono essere private, i metodi get e i set sono di aiuto, ma non richiesti per favorire l'encapsulation.

Sottoclassi

Tutte le classi **che non estendono già un'altra classe** estendono di default la classe java.lang.Object.

Se nel main dichiari

- Superclasse s = new sottoclasse(); s.metodo();

Il **tipo dichiarato (object type)** è quello che è a sinistra, mentre il **tipo effettivo (reference type)** è quello che sta a destra. La variabile s quindi è di **tipo dichiarato** Superclasse, e di **tipo effettivo** sottoclasse. **L'object type** si riferisce agli attributi dell'oggetto che esistono in memoria, mentre il **reference type** determina come l'oggetto può essere utilizzato dal chiamante. Se nella sottoclasse scrivo new Superclasse().metodo() è come se avessi creato una variabile di tipo dichiarato superclasse e tipo effettivo superclasse, quindi andrò SOLO nel metodo della superclasse.

Nel nostro esempio metodo() DEVE esserci nella superclasse!!!!! Questo perché **SEMPRE** prima si visita la superclasse e poi si scende alla sottoclasse. Se il metodo è statico allora andiamo dentro quello della superclasse, dato che s è di **tipo dichiarato** superclasse, mentre invece fosse stato di **tipo dichiarato** sottoclasse saremmo andati nel metodo statico della sottoclasse. Stessa cosa con le variabili, si prendono sempre quelle del **tipo dichiarato**. Anche se c'è un hiding dell'attributo nella sottoclasse. Nel nostro esempio s eredita le variabili della superclasse, quindi se nella superclasse cambi la variabile di istanza allora cambierà anche nella sottoclasse.

Attenzione perché non puoi istanziare una classe astratta, ha un costruttore, ma esso può essere chiamato attraverso la sottoclasse, non puoi fare ClassAstratta x = new classeAstratta()

Tutti i costruttori della sottoclasse è come se avessero il metodo super() di default!!! Quindi se nella superclasse tu dichiari il costruttore con qualche parametro allora lo perdi. La sottoclasse quindi almenochè non abbia un super(parametri) va a chiamare di default quello senza parametri, e se non c'è ci sarà errore!

Una classe **final** non può essere estesa, un metodo **final** non può essere overridato. Attenzione perché se il metodo è private final allora non c'è alcun problema se c'è la sottoclasse con lo stesso metodo, dato che è private non è visibile, quindi il final è solo un trabocchetto, tale metodo non può essere chiamato da s per esempio, proprio perché è private. Una variabile final può essere valorizzata solo la prima volta, però si può aggirare la cosa semplicemente passando tale variabile in un metodo, qui puoi fare tranquillamente l'assegnamento. **OVERRIDE:** Quando una classe estende una classe o implementa un'interfaccia, il metodo che vuole overridare **deve** essere ovviamente di istanza (non static), avere lo stesso numero di parametri e un tipo covariante, cioè compatibile, per esempio interfaccia ha un metodo **Number** e tu overridi con un metodo **Long**, oppure Object e tu overridi con una qualsiasi classe, attenzione perché Integer e int **NON** sono covarianti, idem per Number e double essendo un primitivo. Se la classe estende una classe e implementa un'interfaccia che hanno lo stesso metodo allora il metodo overridato **deve essere compatibile con entrambe**. Inoltre il metodo overridato DEVE avere un modificatore di accesso uguale o più largo dell'originale, quindi es un protected può essere overridato con un public. I metodi overridati non possono lanciare nuove checked eccezioni o più larghe eccezioni del metodo originale.

CASTING: La regola è che qualcosa di piccolo può entrare in qualcosa di grande, altrimenti avremo una ClassCastException, per esempio una sottoclasse può entrare in una superclasse. Attenzione, se ho (int) (double) x; allora la x viene convertita in un intero. Nel seguente esempio infatti posso mettere Dog dentro la superclasse che estende: Animal e dentro l'interfaccia che implementa: Equipment. Non posso implicitamente inserire new Object() in una variabile di tipo List, ma se uso un casting esplicito allora posso. **Il casting esplicito** mi permette di convertire un **tipo effettivo**, quindi l'oggetto non cambia in memoria, più piccolo in uno più grande, per esempio puoi scrivere (Superclasse)so, ma non puoi scrivere Sottoclasse so = (Sottoclasse) su; avrai une errore a **runtime**: **ClassCastException.** Scrivere (String) new Object() darebbe una ClassCastException! **La regola è: puoi convertire esplicitamente un'istanza tipo (effettivo) della sottoclasse in una superclasse o in un'interfaccia che implementa.**

Attenzione all'esempio qui sotto, **devi ricordare che quel get() restituisce un oggetto di tipo effettivo Dog**, quindi il casting è lecito.

```
abstract class Animal {
    protected final int size = 7;
}

interface Equipment {
}

public class Dog extends Animal implements Equipment {

 public Animal get() { return this; }

 public static void main(String[] passes) {
    //CASTING IMPLICITO
    Dog d = new Dog().get(); //Superclasse in sottoclasse implicitamente, errore
    Dog d1 = new Dog(); //Sottoclasse in sottoclasse implicitamente, ok
    Equipment e = new Dog().get();   //Superclasse in interfaccia implicitamente, errore
    Equipment e1 = new Dog();   //Sottoclasse in interfaccia implicitamente, ok
    Animal a = new Dog().get();    //Superclasse in superclasse, ok
    Animal a1 = new Dog();     //Sottoclasse in superclasse, ok

    //CASTING ESPLICITO (trasformi un tipo in un altro)
    Dog d = (Dog)new Dog();                    //Sottoclasse -> sottoclasse in sottoclasse, ok
    Dog d1 = (Dog)new Dog().get();             //Superclasse -> sottoclasse in sottoclasse, ok
    Equipment e = (Equipment)new Dog();        //Sottoclasse -> interfaccia in interfaccia, ok
    Equipment e1 = (Equipment)new Dog().get(); //Superclasse -> interfaccia in interfaccia, ok
    Animal a = (Animal)new Dog();              //Sottoclasse -> superclasse in superclasse, ok
    Animal a1 = (Animal)new Dog().get();       //Superclasse -> superclasse in superclasse, ok

    System.out.print(((Dog)e).size);
    System.out.print(((Dog)a).size);
    System.out.print(((Equipment)d).size); //errore
    System.out.print(((Equipment)a).size); //errore
    System.out.print(((Animal)d).size);
    System.out.print(((Animal)e).size);
 }
```

```java
class Animal {
  int size = 7;
}

public class Dog extends Animal {

    public int size = 10;

    //tipo dichiarato Animal - tipo effettivo Dog
    public Animal get() {
        return this;
    }

    public static void main(String... books1) {

        Dog d1 = new Dog();                 //ok
        //Dog d2 = new Animal();            //errore
        Animal d3 = new Animal();           //ok
        Animal d4 = new Dog();              //ok

        //Dog d5 = new Dog().get();         //errore
        Animal d6 = new Dog().get();        //ok

        //CASTING ESPLICITO
        //Dog d7 = (Animal) new Dog();      //Il casting è ok, Animal in Dog errore.
        //Dog d8 = (Dog) new Animal();      //Il casting è errore
        Animal d9 = (Animal) new Dog();     //ok

        Dog d10 = (Dog) new Dog().get();    //Il casting Animal -> Dog è ok perchè il get da un tipo effettivo Dog
        Animal d11 = (Dog) new Dog().get(); //ok
        //Dog d12 = (Animal) new Dog().get(); //Il casting Animal -> Animal è ok, Animal in Dog errore.
        Animal d13 = (Animal) new Dog().get(); //ok

    }
}
```

L'overload invece è quando ci sono dei metodi con lo stesso nome MA con parametri differenti, il tipo di ritorno, quindi, NON C'ENTRA NULLA. Poi c'è **l'hiding** che è quando una sottoclasse ha la stessa variabile di una superclasse, oppure lo stesso metodo statico.

Una variabile di istanza o classe della superclasse per essere letta dalla sottoclasse senza un classico metodo get() NON deve essere private.

```java
class BubbleException extends Exception {}
class BlueException extends BubbleException {}
class Fish {
 Fish getFish() throws BubbleException {
    throw new RuntimeException("fish!");
 }
}
public class Clownfish extends Fish {
 public final Clownfish getFish() throws BlueException {    //il  metodo overridato può dichiarare eccezioni uguali o più ristrette o eliminarla proprio
    throw new RuntimeException("clown!");
 }
 public static void main(String[] bubbles) throws Exception {
    final Fish f = new Clownfish();
    f.getFish();
    System.out.println("swim!");
 }
}
```

Classi astratte

Le classi **abstract** oltre ai metodi concreti possono avere anche dei metodi **abstract**, (metodi definiti **abstract T metodo()**) cioè senza corpo che DEVONO essere overridati dalle classi che la estendono. Una classe concreta NON può avere metodi abstract. Una classe abstract può avere tranquillamente un **main**. Non possono essere istanziate, non puoi fare new AbstractClass, ma puoi definire variabili di tipo definito AbstractClass e possono avere un costruttore, che può essere chiamato dalla sottoclasse concreta. Se implementa un'interfaccia allora non è obbligata ad overridare il metodo, come invece lo sono le classi concrete. Infatti se una classe concreta estende una classe astratta che implementa un'interfaccia allora la classe concreta deve overridare tutti i metodi (non static e non default) dell'interfaccia. Di default i metodi sono package-private, esattamente come le classi concrete. I metodi abstract non possomno essere private, ovviamente sono lì per essere overridati, quindi non avrebbe senso.

Interfacce

Utili quando stai lavorando ad un progetto con un altro team e stai sviluppando codice che utilizza una classe che l'altro team non ha ancora finito di scrivere. Consente una facile integrazione una volta che il codice dell'altro team è completo. Permettono l'ereditarietà multipla, infatti puoi scrivere implements A,B,C.. alla fine è questa la differenza con le classi abstract. L'interfaccia **estende** un'altra.

Tutti i metodi non possono essere **final** (anche i defualt e static) e vengono applicati implicitamente **SEMPRE** public (e non puoi mettere altri modificatori di visibilità) e **abstract** (senza corpo) che quindi devono essere per forza overridati dalle classi che implementano l'interfaccia, puoi definirli però anche **default** (con un corpo), **static** (con un corpo). Le variabili invece vengono applicate implicitamente **public final static.**

I metodi default servono per aggiungere retrocompatibilità alle interfacce già esistenti, in questo modo infatti puoi aggiungere metodi ad una vecchia interfaccia e usarli, senza dover cambiare tutto ciò che la implementa.

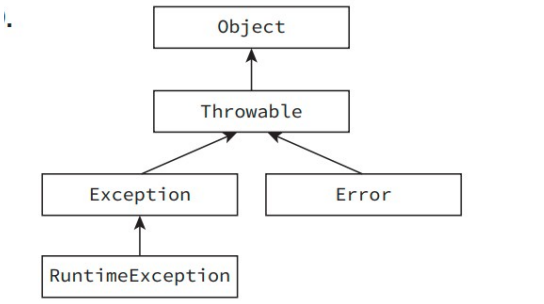
Una classe non può ereditare due interfacce che dichiarano lo stesso metodo default, a meno che la classe non lo sovrascriva.

Nel main puoi avere: Interfaccia x = new ClasseCheImplementaInterfaccia(). Nelle conversioni conta il tipo effettivo.

Eccezioni

Ogni try deve essere accompagnato da un catch o da un finally. Try catch e finally devono avere per forza le {}, anche con una sola istruzione. Se ci sono catch e finally allora l'ordine è per forza try catch finally, e il finally verrà sempre eseguito, anche se nel try c'è un return "ciao", prima andremo nel finally e poi torneremo nel chiamante restituendogli "ciao". Se dentro un catch c'è un altro catch le variabili usate nelle loro parentesi tonde (la variabile in un catch è obbligatoria) devono chiamarsi in modi diversi!

Il giusto ordine è



Tipi di Eccezioni: Due categorie: Le eccezioni controllate sono dovute a circostanze esterne che il programmatore non può evitare, il compilatore vuole sapere cosa fare nel caso si verifichi l'eccezione, quindi **devono essere DICHIARATE** (con throws… nel metodo) **o GESTITE** (con try catch()). Le eccezioni non controllate sono dovute a circostanze che il programmatore può evitare, correggendo il programma.

Le **controllate** sono quelle che crei tu e le ClassNotFoundException, CloneNotSupportedException, IOException (e sottoclassi), tutte le altre (RuntimeException) sono non controllate, sono quelle che producono uno stack trace a runtime.

Attenzione al main, perché se il main chiama uno di questi metodi che lanciano o dichiarano un'eccezione CONTROLLATA, anche lui **deve dichiararla** (trows Exception di un livello uguale o più ampio (a differenza dell'override) di quella del metodo) **o gestirla** (try catch (livello uguale o più ampio)). Se una superclasse ha il metodo public void m() throws Exception allora nella sottoclasse posso avere il metodo overridato che la ingoia, quindi avrò public void m(), ora se nel main la variabile che usa il metodo sarà di tipo dichiarato superclasse allora il main deve dichiarare o gestire l'eccezione, se la variabile fosse di tipo dichiarato sottoclasse allora non ce ne sarebbe bisogno.

Non controllate:

- **NullPointerException** può essere lanciata quando fai array[i] e l'array è vuoto.

- **ArrayIndexOutOfBoundsException** quando tenti di accedere ad un elemento oltre la lunghezza dell'array
- **ClassCastException** quando fai un assegnamento incompatibile, per esempio Boolean[] list = (Boolean[]) new Object()

La regola è: se viene lanciata una checked Exception essa deve essere dichiarata o gestita, SEMPRE. Una sottoclasse che overrida un metodo che lancia un'eccezione controllata **può non lanciarla**, può quindi ingoiarla, basta che non ne lanci una nuova o una più ampia. Un metodo può dichiarare più eccezioni, varrà quella più larga. Se il finally lancia un'eccezione sarà sempre lei ad essere lanciata alla fine, dato che verrà sempre eseguito alla fine, quindi è la sua eccezione che conta, se nel catch() c'è una throw new Exception() e nel finally c'è throw new RuntimeException() allora sarà quest'ultima per forza ad essere lanciata, una unchecked exception, il metodo quindi non deve dichiarare o gestire alcuna eccezione.

```
class Problem extends RuntimeException {}   //deve usare la keyword extends ovviamente, sono classi e non interfacce

public class BiggerProblem extends Problem {    //se non estendesse Problem non sarebbe sottoclasse di RuntimeException,
                                                 //non sarebbe quindi sottoclasse di Throwable e quindi non potrei lanciarla
                                                 //né catcharla, avrei errore: cannot be converted to Throwable
    public static void main(String uhOh[]) {
        try {
            throw new BiggerProblem();  //questo va nel primo catch
            //throw new Problem();      //questo va nel secondo catch
                                        //non possono esserci entrambi, darebbe errore: unreachable statement
        } catch (BiggerProblem re) {
            System.out.print("Problem?");
        } catch (Problem e) {
            System.out.print("Handled");
        } finally {
            System.out.print("Fixed!");
        }
    }
}
```

Gli **ERRORI** invece vengono generati quando l'applicazione è entrata in uno stato finale e irrecuperabile, per esempio quando l'applicazione ha esaurito la memoria, The application runs out of memory. Un'applicazione quindi non dovrebbe mai cacciare un Error, però può farlo, serve sempre la variabile come nome, quindi avremo catch(Error e).