# Esercizio:
# Espressioni Aritmetiche

In questo primo esercizio creeremo delle classi (e interfacce) per modellare delle espressioni aritmetiche di interi (per semplicità non ci preoccuperemo di numeri con virgola), come costanti, somme, moltiplicazioni, ecc. Un'espressione complessa sarà rappresentata come un albero e come tale le precedenze delle varie operazioni saranno dettate dalla struttura dell'albero, quindi non ci poniamo il problema della precedenza degli operatori aritmetici ne' delle parentesi per raggruppare espressioni.

Per quanto riguarda il progetto scegliete un nome che preferite. Stessa cosa per il nome del pacchetto come preferite. Nelle soluzioni dell'esempio sarà usato *mp.exercise.expressions*.

Le espressioni devono avere un metodo *eval* che ritorna un intero con la valutazione dell'espressione.

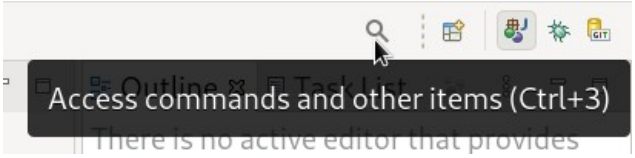
L'esercizio è mirato anche a prendere familiarità con la scrittura di test automatici con JUnit. Per semplicità scriveremo tutti i test in un'unica classe *ExpressionsTest*. Ricordate che è bene separare i test dal codice vero e proprio. Quindi il codice vero sarà nella directory sorgente *src*, mentre per i test li scriveremo in una nuova directory sorgente *tests*. Fate riferimento al tutorial su JUnit, per quel che riguarda la creazione del progetto, la cartella per i test e per come creare il test JUnit (incluso l'aggiunta al classpath di **JUnit 4**).

L'esercizio è suddiviso in diversi passi, da svolgere in sequenza. Inoltre, l'esercizio ha i seguenti obiettivi:

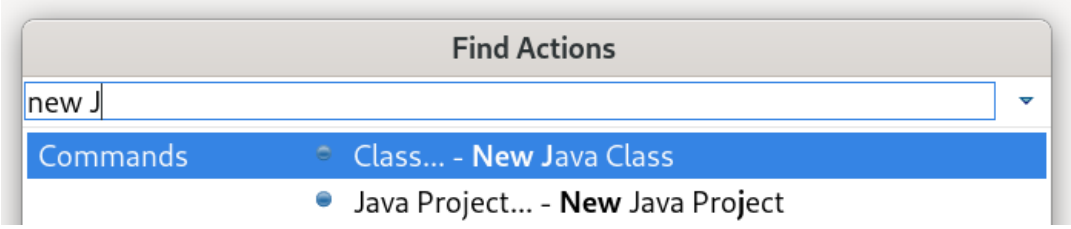
- Implementare il programma gradualmente, una classe alla volta, e passare alla successiva solo dopo aver testato la classe appena implementata. Per questo motivo, NON si deve implementare tutti i punti insieme e testarli solo alla fine.
- Familiarizzare con alcuni strumenti di refactoring.
- NON creare immediatamente astrazioni, ma implementare prima una classe concreta, testarla e poi tramite refactoring creare delle astrazioni (classi astratte e/o interfacce) per un possibile riuso futuro (cioè in vista del prossimo passo dell'esercizio).
- Familiarizzare con gli strumenti dell'IDE per essere produttivi durante l'implementazione, il testing e il refactoring.

**Ricordate**: abituatevi a rilanciare i test dopo ogni modifica (memorizzate le shortcut di Eclipse).

In particolare, in Eclipse, prendete confidenza con la casella in alto a destra a forma di "lente di ingrandimento", detta "Find Actions", accessibile con **Ctrl+3**:



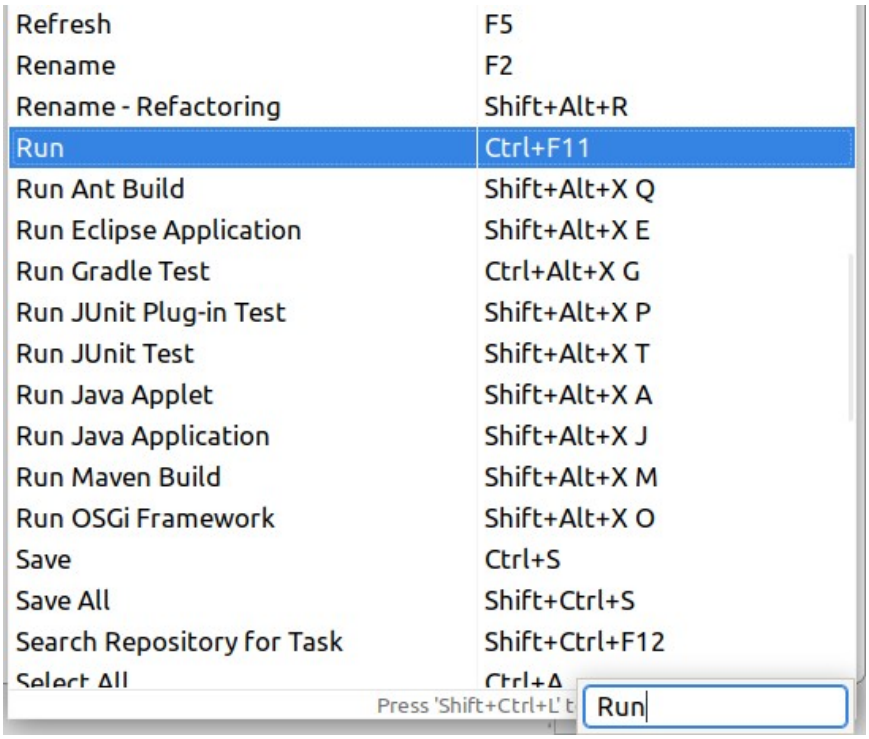
Comparirà una finestra pop-up “Find Actions”. Iniziate a scrivere parti di un comando o voce di menù per vedere tutte le possibili proposte e selezionare quella desiderata. Ad es., per creare una nuova classe Java si può iniziare a scrivere “ne J” (attenzione alle maiuscole):




Allo stesso modo, se avete un file Java aperto, potete velocemente accedere alle voci di Refactoring. Per es., se volete “estrarre” una variabile locale, un’interfaccia, ecc. (cosa che vedremo durante l’esercizio), basta premere Ctrl+3 e iniziare a scrivere “Extr”.

Inoltre useremo spesso **Ctrl+1** (Quick Fix) sia per correggere errori sia per usare possibili suggerimenti per modificare il codice, oltre ovviamente al **Ctrl+Space** per accedere al Content Assist.

Potete scoprire le scorciatoie da tastiera sempre tramite Ctrl+3 (che mostra anche l’eventuale scorciatoia associata). Oppure, tramite lo shortcut **Ctrl+Shift+L** avete accesso alla lista di tutte le scorciatoie (visualizzata in un popup in basso a destra); se iniziate a digitare il popup si sposterà sulla prima voce corrispondente. Ad es., una volta che è comparso il popup, digitando “Run” potete vedere le scorciatoie per i comandi che iniziano con “Run”:




# 1 Espressione costante

Creare una classe *Constant* (che memorizza un valore intero), *eval* e relativo test.

# 2 Interfaccia per espressioni

Refactoring: Introdurre un'interfaccia *Expression*, col metodo *eval* e farla implementare a *Constant*. I test precedenti non devono richiedere modifiche e devono continuare a funzionare.

# 3 Somma di costanti

Creare una classe *Sum* che implementa *Expression* per modellare (al momento) una somma di espressioni costanti. Tali espressioni devono essere passate al costruttore. Implementare e testare *eval*.

# 4 Somma di espressioni

Al momento siamo in grado di rappresentare solo somme di costanti, ma dovremmo rappresentare anche somme di somme, somme di moltiplicazioni, ecc… cioè in generale, somme di *Expression*. Modificare *Sum* in tal senso. I test precedenti non devono richiedere modifiche e devono ancora funzionare. Poi aggiungere un nuovo test per la somma di espressioni non necessariamente costanti (es. somma di una costante e di una somma di due costanti).

# 5 Espressione binaria astratta

Refactoring: Introdurre una classe astratta di base, es., *BinaryExpression*, che contiene le funzionalità di base per tutte le espressioni binarie, cioè la struttura dei suoi oggetti: i due campi per le sottoespressioni. La classe base NON dovrebbe permettere un accesso diretto ai campi alle sottoclassi. È la classe base astratta che si deve occupare dell'inizializzazione dei campi adesso.

Alla fine, i test esistenti non devono richiedere modifiche e devono continuare a funzionare.

# 6 Moltiplicazione di espressioni

Creare una classe *Multiplication* che rappresenta la moltiplicazione di due espressioni. Testarne l'implementazione.

# 7 Altre espressioni binarie

Usando le stesse tecniche, create le classi per l'operazione di sottrazione e di divisione (operazioni binarie). Ovviamente la divisione sarà da intendersi come divisione intera, visto che, per semplicità, le costanti hanno solo valori interi.

# 8 Espressioni unarie

Usando le stesse tecniche

1. implementare l'espressione unaria per la negazione aritimetica (es., "-2"), *Negation*
2. rifattorizzare in una classe base astratta *UnaryExpression*

3. implementare l’espressione unaria per il fattoriale, *Factorial*, per il momento ignoriamo la gestione del caso di errore quando la sottoespressione è negativa (ricordate invece che il fattoriale di 0 è 1).

Un’espressione unaria ha una sola sottoespressione; tenetene conto quando scegliete un nome per l’apposito campo.