

Esercizio:

Espressioni Aritmetiche (svolgimento)

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 né 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*. **IMPORTANTE:** ricordatevi di creare il progetto Eclipse seguendo le slide che vi ho fornito "Configurare Java in Eclipse".

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

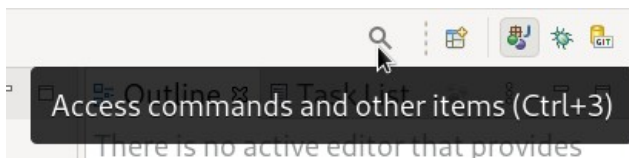
L'esercizio è mirato anche a prendere familiarità con la scrittura di test automatici con JUnit. Per semplicità scriveremo tutti i test in un'unica classe *ExpressionsTest*. Ricordate che è bene separare i test dal codice vero e proprio. Quindi il codice vero sarà nella directory sorgente *src*, mentre 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:

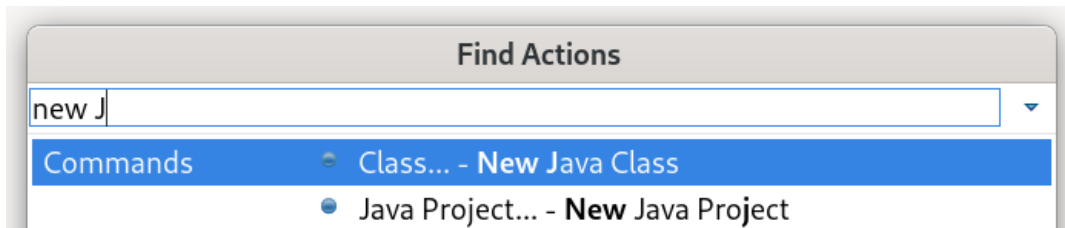
- Implementare il programma gradualmente, una classe alla volta, e passare alla successiva solo dopo aver testato la classe appena implementata. Per questo motivo, **NON** si deve implementare tutti i punti insieme e testarli solo alla fine.
- Familiarizzare con alcuni strumenti di refactoring.
- **NON** 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**:



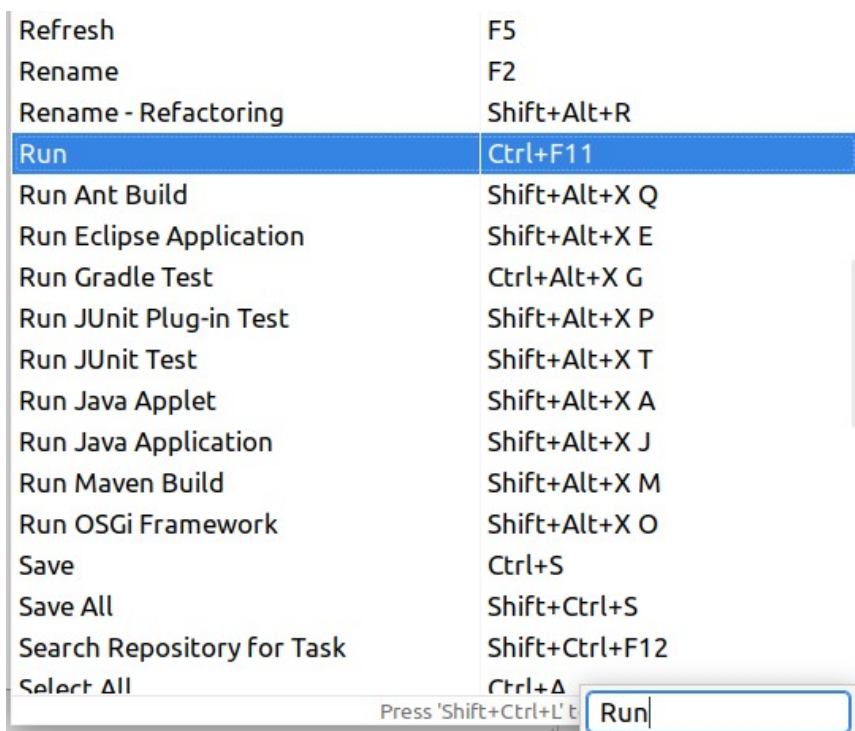
Comparirà una finestra pop-up “Find Actions”. Iniziate a scrivere parti di un comando o voce di menù per vedere tutte le possibili proposte e selezionare quella desiderata. Ad es., per creare una nuova classe Java si può iniziare a scrivere “ne J” (attenzione alle maiuscole):



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.

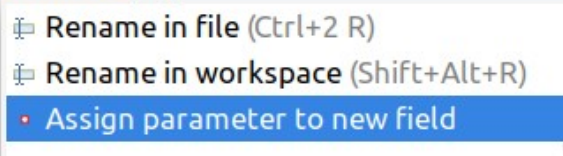
La classe deve avere un campo intero per la costante numerica, ad es., *value* o *constantValue* (ma sono da evitare nomi poco significativi come *x*, *i*, ecc...).

Creare la classe con lo wizard

```
package mp.exercise.expressions;  
  
public class Constant {  
  
}
```

Scrivere il costruttore (suggerimento: iniziare a scrivere “Const” e poi usare il Content Assist che dovrebbe proporre di scrivere il costruttore) che prende l’intero per il campo come parametro; posizionarsi sul parametro, selezionare Ctrl+1 e scegliere la voce per creare e assegnare il parametro a un nuovo campo:

```
public class Constant {  
    public Constant(int constantValue) {  
    }  
}
```



Otteniamo:

```
public class Constant {  
    private int constantValue;  
  
    public Constant(int constantValue) {  
        this.constantValue = constantValue;  
    }  
}
```

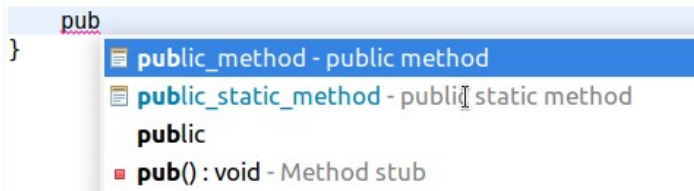
NOTA: Usando questa tecnica, saremo sicuri che le variabili di istanza saranno private e non possiamo dimenticarci di assegnar loro un valore.

ATTENZIONE: al momento non sono necessari *getter/setter*, quindi NON mettiamoli!

Per creare velocemente un metodo (pubblico in questo caso), possiamo usare uno dei *template* messi a disposizione da Eclipse tramite il Content Assist. Basta iniziare a scrivere “pub” e richiamare il Content Assist (Ctrl+Space) per accedere al template corrispondente (template simili sono disponibili per metodi privati, protected, ecc.):

```
public final class Constant {
    private int constantValue;

    public Constant(int constantValue) {
        this.constantValue = constantValue;
    }
}
```



Subito dopo averlo creato, Eclipse ci dà la possibilità di modificare alcune parti della dichiarazione del nuovo metodo, evidenziando le parti modificabili. Per spostarsi fra le varie sezioni si usa il tasto Tab. Attenzione: queste parti modificabili preselezionate rimangono disponibili solo dopo aver creato il metodo: se ci spostiamo su un altro editor o su un'altra applicazione diversa da Eclipse queste parti non saranno più editabili con questa funzionalità e dovremo modificarle spostandoci manualmente nell'editor. Spostiamoci (con Tab) sul tipo e specifichiamo "int" poi spostiamoci sul nome (con Tab) e specifichiamo *eval*. Notare le parti editabili nello screenshot:

```
public int name() {
}
```

Implementiamo *eval*, cioè, banalmente:

```
public int eval() {
    return constantValue;
}
```

Testare il metodo (fate riferimento alla lezione su JUnit; suggerimento: posizionati sul nome della classe, usando il Ctrl+1 si ha accesso alla voce "Create new JUnit test..." che aprirà il wizard che avevamo visto nella lezione di JUnit; ovviamente si dovrà scegliere poi il nome della classe di test come sotto.), ad es.:

```
package mp.exercise.expressions;

import static org.junit.Assert.*;

import org.junit.Test;

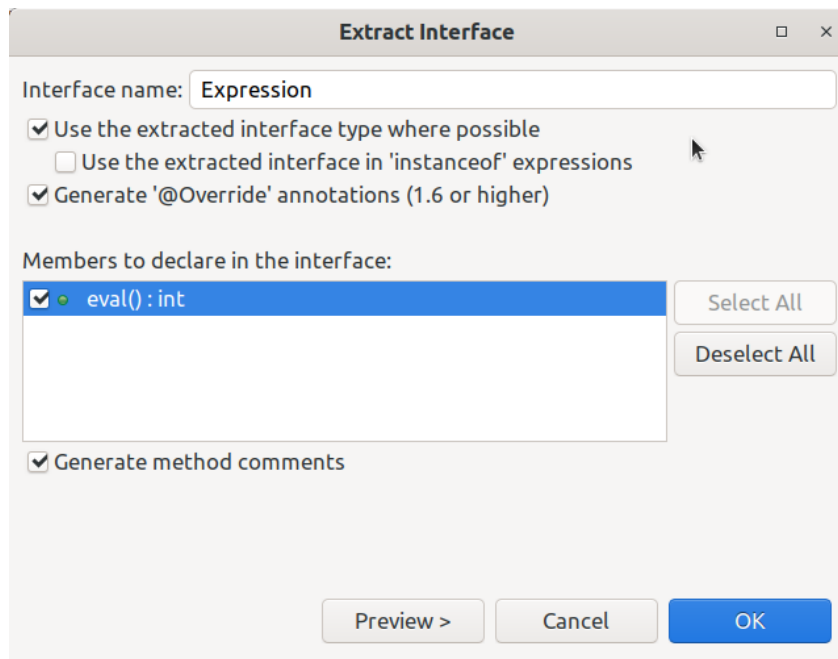
public class ExpressionsTest {

    @Test
    public void testConstantEval() {
        assertEquals(5, new Constant(5).eval());
    }
}
```

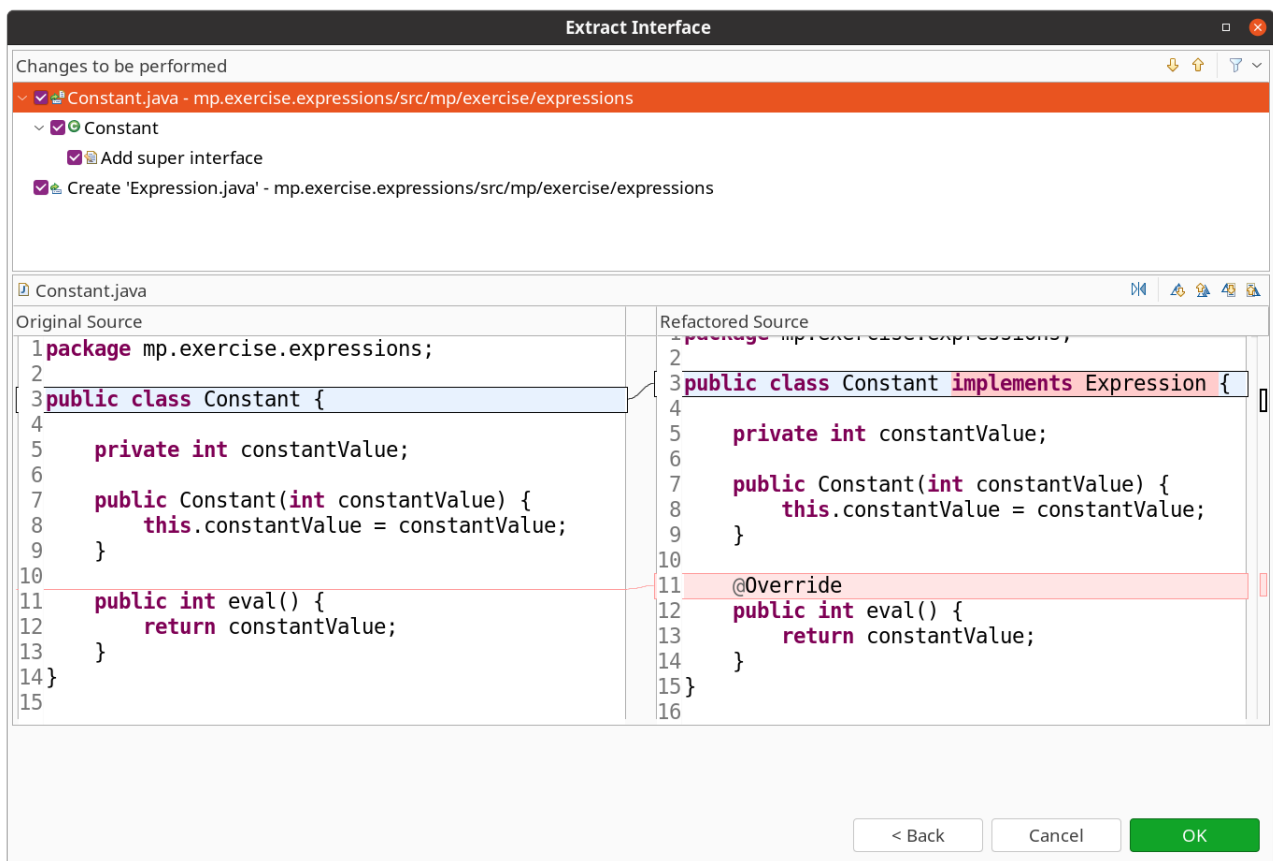
2 Interfaccia per espressioni

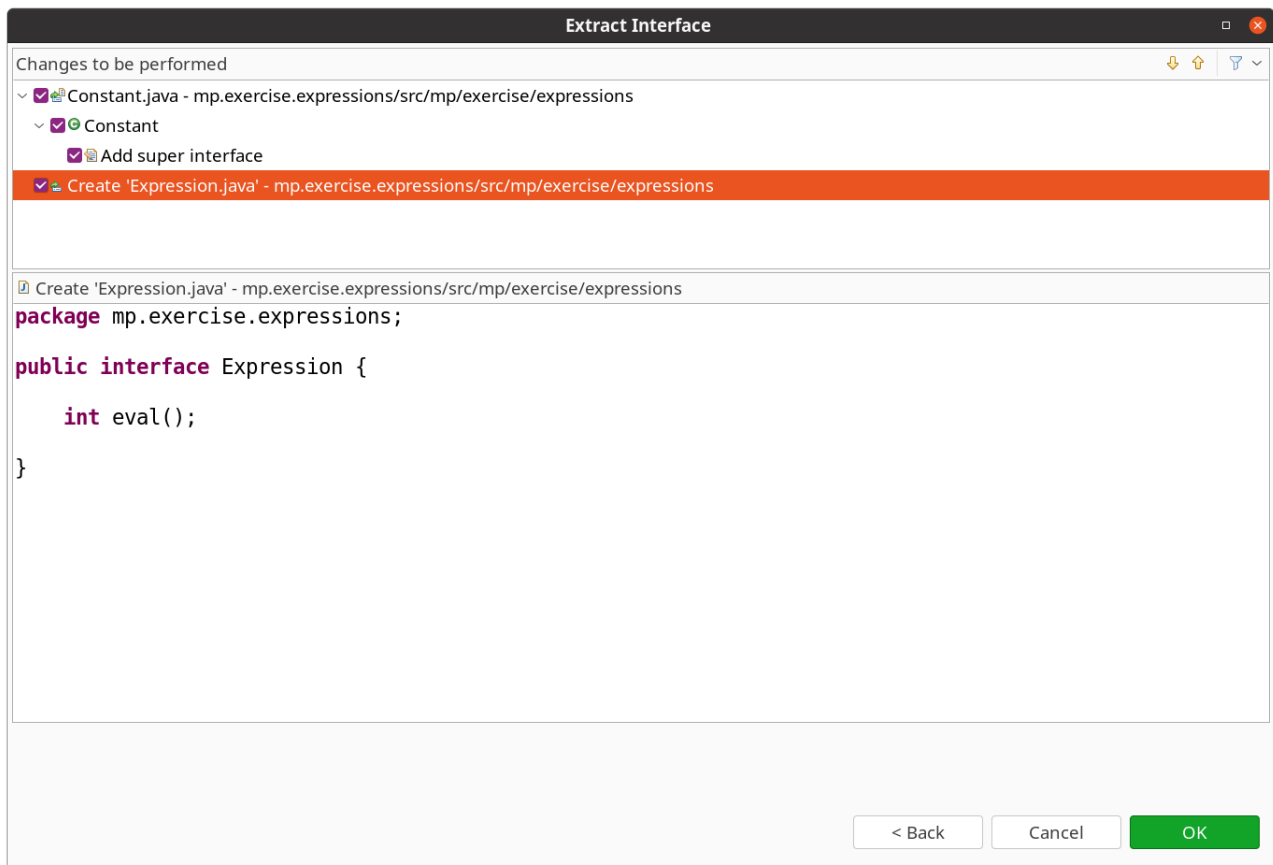
Refactoring: Introdurre un'interfaccia *Expression*, col metodo *eval* e farla implementare a *Constant*. I test precedenti non devono richiedere modifiche e devono continuare a funzionare.

Dall'editor del file di *Constant* selezionare (ad es., dal menù contestuale, oppure usando Ctrl+3) il refactoring "Extract Interface..." e specificare quanto segue:



Se volete, prima di dare "OK" potete usare il pulsante "Preview" per vedere quello che Eclipse farà con questo refactoring:





Come risultato sarà creata l'interfaccia *Expression* e *Constant* sarà aggiornata in modo da implementarla:

```
package mp.exercise.expressions;

public interface Expression {

    int eval();

}

public class Constant implements Expression {
    ...
    @Override
    public int eval() {
        return constantValue;
    }
}
```

NOTA: Usando questa tecnica, non ci dimenticheremo di aggiungere l'annotazione `@Override`.

Rilanciare i test per assicurarsi che tutto funzioni ancora. (Ovviamente non c'è motivo che il test precedente adesso fallisca, ma abituatevi sempre a rilanciare sempre tutti i test dopo ogni modifica e dopo ogni refactoring.)

Per il momento non abbiamo in programma di estendere la classe *Constant*, quindi può valer la pena renderla *final* in modo da non estenderla "per sbaglio":

```
public final class Constant implements Expression {
```

Allo stesso modo, possiamo rendere anche il campo *final* così da avere una classe per creare solo oggetti immutabili:

```
public final class Constant implements Expression {  
  
    private final int value;  
  
    public Constant(int value) {  
        this.value = value;  
    }  
  
    @Override  
    public int eval() {  
        return value;  
    }  
}
```

3 Somma di costanti

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

Usiamo il wizard per creare una nuova classe (In alternativa, se siamo sul nome dell'interfaccia *Expression*, possiamo usare Ctrl+1 e la voce "Create new implementation of ..."). Si deve "aggiungere" *Expression* come interfaccia e selezionare l'opzione per creare un'implementazione fittizia temporanea ("stub") dei metodi astratti:

The screenshot shows the "New Java Class" dialog box. The "Name" field contains "Sum". The "Modifiers" section has "public" selected. The "Superclass" field contains "java.lang.Object". The "Interfaces" section has "mp.exercise.expressions.Expression" selected. The "Which method stubs would you like to create?" section has "Inherited abstract methods" checked. The "Do you want to add comments?" section has "Generate comments" unchecked. The "Finish" button is highlighted.

Si ottiene:

```
public class Sum implements Expression {  
  
    @Override  
    public int eval() {  
        // TODO Auto-generated method stub  
        return 0;  
    }  
  
}
```

Per il momento preoccupiamoci di essere in grado di modellare almeno la somma di due costanti.

Dobbiamo quindi aggiungere due campi *left* e *right* di tipo *Constant*. (In questo contesto *left* e *right* dovrebbero essere abbastanza significativi e far capire che contengono la sottoespressione sinistra e destra; in alternativa potremmo essere ancora più espliciti con *leftExpression* e *rightExpression*, oppure con *leftSubExpression* e *rightSubExpression*). Tali due campi devono essere inizializzati nel costruttore usando due parametri passati. Come fatto prima, creiamo prima il costruttore coi due parametri e poi usiamo il tool di Eclipse per creare automaticamente i campi (possiamo creare entrambi i campi usando la voce “Assign all parameters to new fields”).

```
private Constant left;  
private Constant right;  
  
public Sum(Constant left, Constant right) {  
    this.left = left;  
    this.right = right;  
}
```

A questo punto possiamo implementare *eval* valutando ricorsivamente l’espressione a sinistra e a destra e poi sommando i risultati:

```
@Override  
public int eval() {  
    return left.eval() + right.eval();  
}
```

Scriviamo un test, usando dei valori che non lascino dubbi:

```
@Test  
public void testSumEval() {  
    assertEquals(15,  
        new Sum(  
            new Constant(10),  
            new Constant(5))  
        .eval());  
}
```

Meglio evitare valori che possano lasciare dubbi sulla corretta verifica (es., $2 + 2 = 4$, ma anche $2 * 2 = 4$, quindi passando 2 come valore per *left* e *right* il test non ci garantirebbe che effettivamente abbiamo implementato la somma: potremmo aver per sbaglio implementato la moltiplicazione).

4 Somma di espressioni

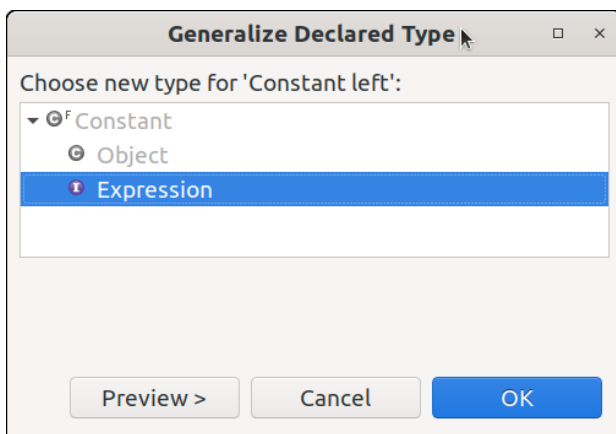
Al momento siamo in grado di rappresentare solo somme di costanti, ma dovremmo rappresentare anche somme di somme, somme di moltiplicazioni, ecc... cioè in generale, somme di *Expression*. Modificare *Sum* in tal senso. I test precedenti non devono richiedere modifiche e devono ancora funzionare. Poi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

L'idea è quella di cambiare il tipo dei campi di *Sum* in *Expression*, ma dobbiamo aggiornare anche i tipi dei parametri del costruttore.

Ovviamente, a parte la sostituzione a mano, si potrebbe usare la funzionalità “Find/Replace” di Eclipse. Tuttavia, questa sostituzione agisce a livello puramente testuale e quindi non ci sarebbe nessun controllo statico sui tipi (ad es., potremmo scrivere un valore errato per il nome del tipo con cui vogliamo sostituire quello esistente).

Per questo, vediamo due modi, basati sugli strumenti di refactoring per Java, per ottenere il nostro scopo.

Possiamo usare il refactoring “Generalize Declared Type”, dopo aver selezionato uno dei due campi, e specificando *Expression*:



Stessa cosa per l'altro campo.

Poi è necessario cambiare il tipo dei parametri nel costruttore, anche in questo caso si può usare lo stesso refactoring sui parametri del costruttore uno alla volta.

In alternativa, invece di usare questo refactoring, possiamo cambiare il tipo dei parametri nel costruttore (da *Constant* a *Expression*). A quel punto il codice non compilerà perché non possiamo assegnare un *Expression* a un campo dichiarato come *Constant*. Usando il quickfix di Eclipse (ad es., usando sempre Ctrl+1) possiamo modificare il tipo dei campi in *Expression*.

```
private Constant left;
private Constant right;

public Sum(Expression left, Expression right) {
    this.left = left;
    this.right = right;
}

@Override
```

Quickfix menu:

- Add cast to 'Constant'
- Change type of 'left' to 'Expression'
- Change type of 'left' to 'Constant'

Resulting code:

```
private Expression left;
private Constant right;
```

Provate entrambi i meccanismi. Alla fine il codice dovrebbe essere così:

```
public class Sum implements Expression {  
  
    private Expression left;  
    private Expression right;  
  
    public Sum(Expression left, Expression right) {  
        this.left = left;  
        this.right = right;  
    } ...  
}
```

Ovviamente, rilanciamo i test. Inoltre aggiungiamo un nuovo test per *Sum* modellando una somma di una somma e una costante:

```
@Test  
public void testComplexSumEval() {  
    assertEquals(15,  
        new Sum(  
            new Sum(  
                new Constant(10),  
                new Constant(2)),  
            new Constant(3))  
        .eval());  
}
```

Volendo, possiamo rendere la classe *final* visto che non prevediamo che debba essere estesa; stesso discorso per quanto riguarda i due campi.

NOTA SULLA FORMATTAZIONE: Eclipse fornisce meccanismi per formattare automaticamente il codice (Dal menù “Source” → “Format”, oppure con la shortcut Ctrl + Shift + F). Tuttavia, non sempre la formattazione automatica porta al risultato desiderato. Ad es., formattando automaticamente i test precedenti, viene portato tutto su un’unica riga, che è meno leggibile della formattazione manuale che abbiamo fatto. Quindi, NON affidatevi sempre e ciecamente alla formattazione automatica di tutto un file. Potete formattare automaticamente solo la parte di codice selezionata, oppure usare i meccanismi di Eclipse per delimitare alcune parti che Eclipse non deve formattare (quest’ultimo meccanismo è lasciato per esercizio).

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.

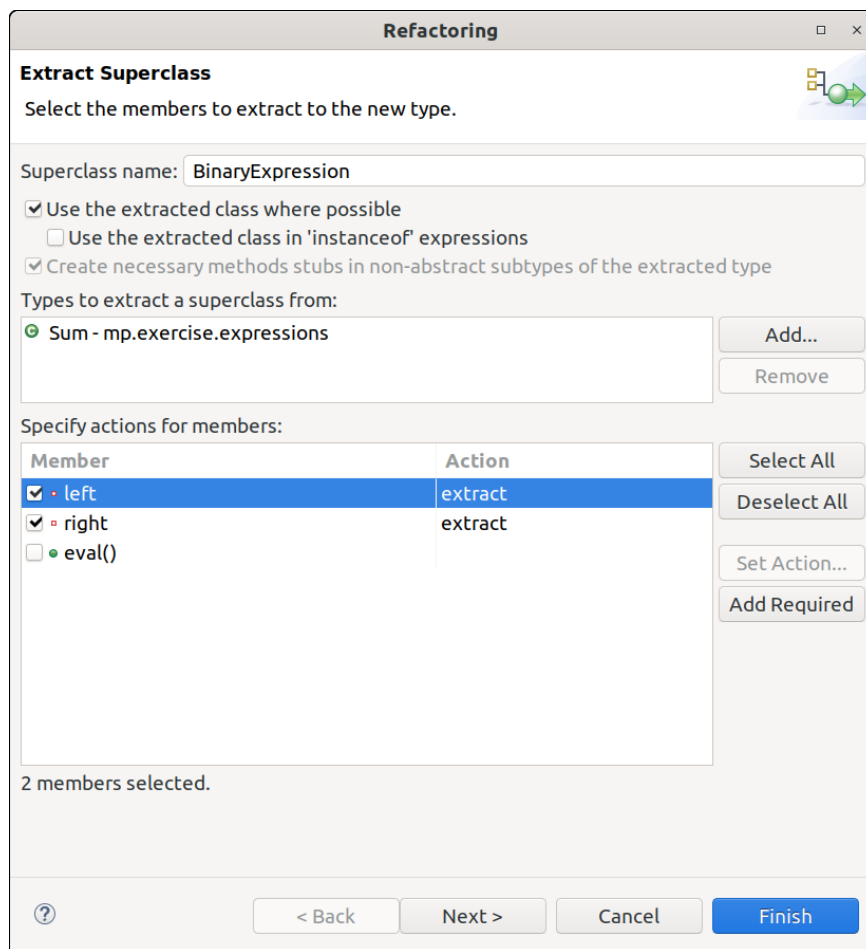
Alla fine, i test esistenti non devono richiedere modifiche e devono continuare a funzionare.

(A volte, quando si crea una classe astratta, si può specificare anche *Abstract* nel nome della classe, ad es., *AbstractBinaryExpression*; in questo esempio però dovrebbe essere abbastanza chiaro che *BinaryExpression* è abbastanza generica da essere astratta.)

Introduciamo la classe base astratta partendo da *Sum* che abbiamo già implementato.

Vediamo due modi per farlo.

Possiamo usare un refactoring simile a quello che abbiamo già usato per creare l'interfaccia. Partendo da *Sum*, usiamo “Extract Superclass”, e selezioniamo i campi che andranno a finire nella superclasse (e ovviamente saranno rimossi dalla classe corrente *Sum*):



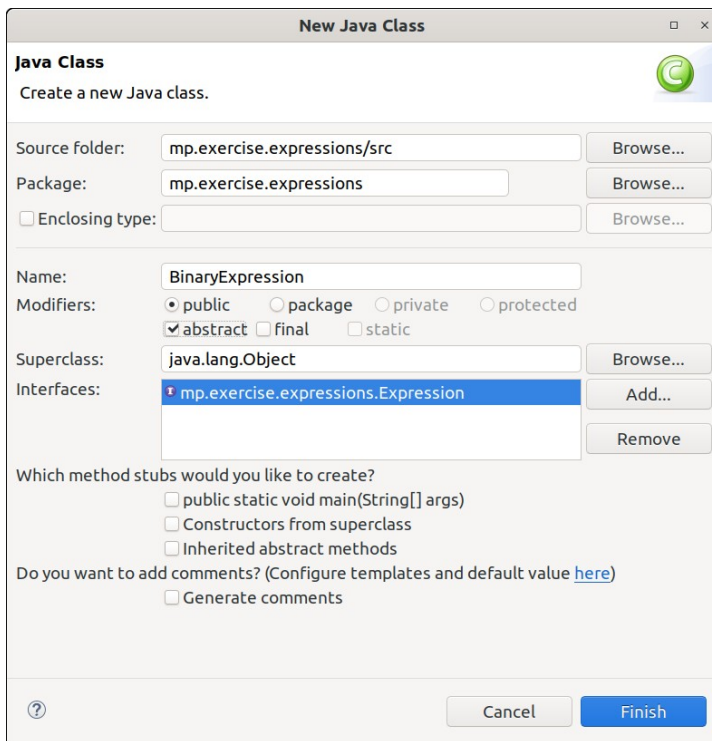
Notare che è possibile estrarre una superclasse da diverse classi esistenti (pulsante “Add...”).

In questo modo però la nuova classe base *BinaryExpression* non implementerebbe automaticamente *Expression*, mentre noi vorremmo che fosse così. Ovviamente dopo questo refactoring potremmo aggiustare manualmente questa cosa.

IMPORTANTE: non ha senso definire come astratto, in una classe astratta, un metodo già definito (come astratto) nell'interfaccia che la classe va a implementare. Per questo NON definiremo come astratto *eval* in *BinaryExpression*: è già parte di *BinaryExpression* essendo dichiarato nell'interfaccia *Expression*.

In alternativa, invece di usare questo refactoring, procediamo come segue.

Creiamo la classe *BinaryExpression* col wizard standard, specificando che implementa *Expression* e che deve essere astratta (senza usare “Inherit abstract methods”):

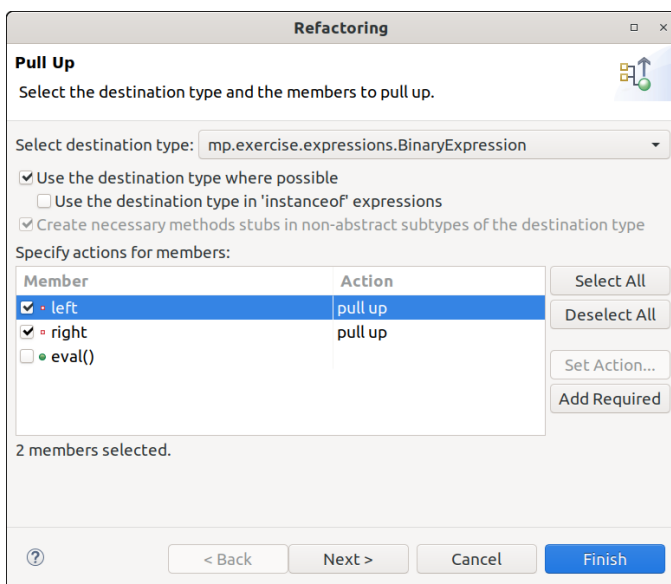


```
public abstract class BinaryExpression implements Expression {
}
```

A questo punto facciamo estendere questa classe a *Sum* (e rimuoviamo l'**implements** di *Expression* da *Sum* in quanto è già implicato dal fatto che estende *BinaryExpression*):

```
public class Sum extends BinaryExpression {
```

Adesso usiamo il refactoring “Pull Up”, per spostare i due campi da *Sum* in *BinaryExpression*:



Otteniamo le seguenti modifiche:

```
public abstract class BinaryExpression implements Expression {
    protected Expression left;
```

```

    protected Expression right;
}

public class Sum extends BinaryExpression {

    public Sum(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

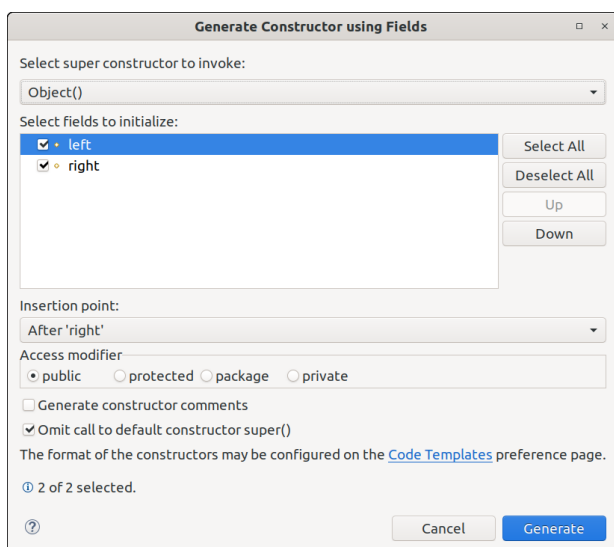
    @Override
    public int eval() {
        return left.eval() + right.eval();
    }
}

```

Ovviamente rilanciamo sempre i test.

Al momento i due campi sono *protected* in *BinaryExpression*, non *private* (come erano originariamente in *Sum*); il refactoring li ha resi tali altrimenti *Sum* non compilerebbe più (ricordate che il refactoring non deve cambiare il comportamento, e ovviamente non può rendere il codice non compilabile). Noi DOBBIAMO evitare l'uso di campi *protected* (meglio avere dei *getter* protetti nella superclasse, e non permettere l'accesso diretto a campi *protected* nelle sottoclassi). Inoltre, al momento i campi protetti di *BinaryExpression* sono inizializzati nel costruttore di *Sum*. Anche questo non è molto “pulito”. Sistemiamo questi problemi tramite dei refactoring e altre modifiche.

Prima di tutto, aggiungiamo il costruttore con parametri in *BinaryExpression*. Usiamo “Generate constructor using fields...” dal menù “Source” (notare che questo NON è un refactoring), e selezioniamo i due campi (selezioniamo l'opzione per omettere la chiamata inutile a *super*):



```

public abstract class BinaryExpression implements Expression {

    protected Expression left;
    protected Expression right;

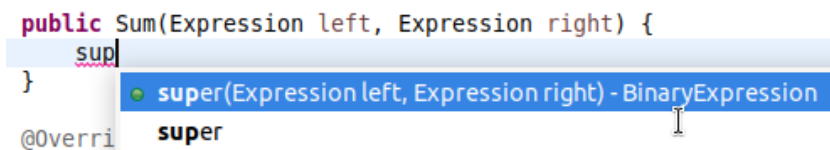
    public BinaryExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }
}

```

```
}  
  
}
```

IMPORTANTE: è bene rendere *protected* il costruttore di questa classe, in modo che sia accessibile solo dalle classe derivate. Infatti, in una classe astratta, è fuorviante (nonostante sia corretto) mettere costruttori pubblici, visto che una classe astratta non può essere istanziata direttamente. Tali costruttori devono essere *protected* in modo che possano essere accessibili solo ai costruttori (*public*) delle classi derivate.

Il codice di *Sum* non compilerà più in quanto *BinaryExpression* adesso non ha un costruttore vuoto implicito, e *Sum* è costretta a chiamare questo nuovo costruttore. In questo modo eviteremo di inizializzare in *Sum* i campi *protected* di *BinaryExpression* (cosa brutta dal punto di vista dell'OO design). Per fare velocemente questa modifica, basta cancellare il body del costruttore di *Sum*, iniziare a scrivere “sup” e poi usare il content assist; la prima proposta è quella che fa al caso nostro:



```
public Sum(Expression left, Expression right) {  
    sup  
}  
@Override  
    super
```

Ottenendo:

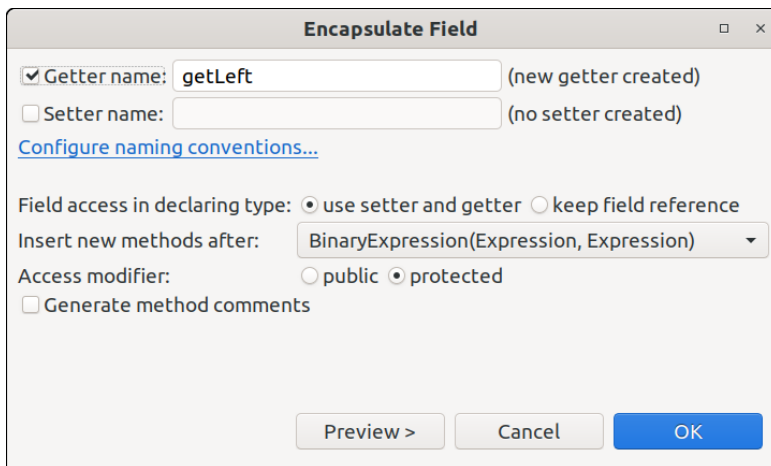
```
public class Sum extends BinaryExpression {  
    public Sum(Expression left, Expression right) {  
        super(left, right);  
    }...  
}
```

E ora il codice di *Sum* torna a compilare.

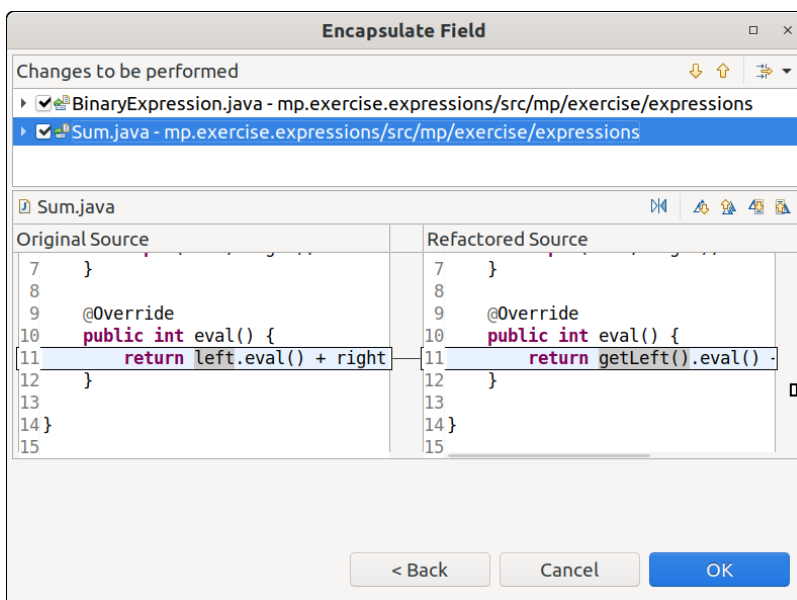
Adesso, liberiamoci dei campi protetti, rendendoli privati in *BinaryExpression*. Ovviamente, *BinaryExpression* dovrà fornire dei *getter* per permettere alle sottoclassi come *Sum* di accedervi.

Un modo per generare i *getter*, in *BinaryExpression*, è usare il comando (ad es., sottomenù del menù contestuale di “Source”) “Generate Getters and Setters”, assicurandoci di dichiararli *protected* e, visto che ci siamo, anche *final*, in quanto non prevediamo per il momento che le sottoclassi debbano sovrascriverli. Però poi dovremo modificare *Sum* in modo che usi i *getter*.

Invece, come alternativa, usiamo il refactoring “Encapsulate Field”, dopo aver selezionato uno dei due campi in *BinaryExpression*. Ovviamente NON selezioniamo il *setter*, in quanto non ci serve.



Notare la selezione di “use setter and getter”. In questo modo, i riferimenti al campo saranno rimpiazzati dal *getter* anche nelle sottoclassi come *Sum*. Potete vedere questo nella “Preview”:



Effettuando la stessa cosa anche per l'altro campo otteniamo quanto segue (purtroppo, dobbiamo manualmente aggiungere *final* ai *getter*):

```
public abstract class BinaryExpression implements Expression {

    protected Expression left;
    protected Expression right;

    public BinaryExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    protected final Expression getLeft() {
        return left;
    }

    protected final Expression getRight() {
        return right;
    }
}
```

```

public class Sum extends BinaryExpression {

    public Sum(Expression left, Expression right) {
        super(left, right);
    }

    @Override
    public int eval() {
        return getLeft().eval() + getRight().eval();
    }

}

```

A questo punto possiamo rendere *private* i due campi in *BinaryExpression* (adesso *Sum* non li usa più direttamente):

```

public abstract class BinaryExpression implements Expression {

    private Expression left;
    private Expression right;
    ...
}

```

6 Moltiplicazione di espressioni

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

A questo punto creare l'espressione moltiplicazione, come derivata di *BinaryExpression*, è semplicissimo.

Tramite lo wizard specifichiamo di generare un costruttore basato su quello della superclasse e uno stub per il metodo astratto *eval*:

New Java Class

Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ package ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)
☒ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

```
public class Multiplication extends BinaryExpression {

    public Multiplication(Expression left, Expression right) {
        super(left, right);
    }

    @Override
    public int eval() {
        // TODO Auto-generated method stub
        return 0;
    }
}
```

Ci basta implementare il metodo *eval* in modo appropriato:

```
@Override
public int eval() {
    return getLeft().eval() * getRight().eval();
}
```

E testarlo:

```
@Test
public void testMultiplicationEval() {
    assertEquals(15,
        new Multiplication(
            new Constant(5),
            new Constant(3))
        .eval());
}
```

Volendo, possiamo rendere la classe *final* visto che non prevediamo che debba essere estesa.

7 Altre espressioni binarie

Usando le stesse tecniche, create (e testate) 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. Inoltre, per la divisione si dovrebbe controllare che la seconda espressione passata non si valuti a 0.

Ricordate inoltre che **le variabili di istanza devono essere tutte private**. Inoltre, visto che abbiamo realizzato classi i cui oggi sono immutabili, possiamo renderle anche *final*. I *setter* non servono e quindi non li mettiamo. Se servono i *getter* per permettere SOLO alle sottoclassi di accedere al valore dei campi definiti nella superclasse li mettiamo *protected*. Se in futuro si rendesse necessario permettere l'accesso al valore dei campi anche a tutte le altre classi (come vedremo in esercitazioni future) allora li renderemo *public* solo allora.

Infine, ricordate che i costruttori delle classi astratte dovrebbero essere *protected* e non *public*, per quanto detto precedentemente in questo documento.

8 Espressioni unarie

Usando le stesse tecniche

1. implementare l'espressione unaria per la negazione aritmetica (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.

Alla fine il risultato di tutto potrebbe essere qualcosa del tipo:

```
package mp.exercise.expressions;

public abstract class UnaryExpression implements Expression {

    private final Expression subExpression;

    protected UnaryExpression(Expression subExpression) {
        this.subExpression = subExpression;
    }

    protected final Expression getSubExpression() {
        return subExpression;
    }
}

package mp.exercise.expressions;

public final class Negation extends UnaryExpression {
```

```

    public Negation(Expression subExpression) {
        super(subExpression);
    }

    @Override
    public int eval() {
        return -(getSubExpression().eval());
    }
}

package mp.exercise.expressions;

import java.util.stream.IntStream;

public final class Factorial extends UnaryExpression {

    public Factorial(Expression subExpression) {
        super(subExpression);
    }

    @Override
    public int eval() {
        int n = getSubExpression().eval();
        // TODO: exception in case of negative n
        return IntStream
            .rangeClosed(2, n)
            .reduce(1, (x, y) -> x * y);
    }
}

```

Notare che per il fattoriale, invece di usare la soluzione a cui magari siete abituati (gestione separata del caso base, e poi ciclo for o soluzione ricorsiva) abbiamo preferito la versione dichiarativa, generando uno stream (in particolare, lo stream ottimizzato per interi di base, *IntStream*) da 2 a n e poi riducendo il tutto con la moltiplicazione, specificando l'elemento "identità" 1 (che verrà usato anche nel caso lo stream sia vuoto, cioè quando n è 0 o 1, non a caso $0! = 1! = 1$). Per esercizio, andate a legervi il Javadoc di *java.util.stream.IntStream.reduce(int, IntBinaryOperator)*.

e i rispettivi test

```

@Test
public void testNegationEval() {
    assertEquals(-3,
        new Negation(new Constant(3)).eval());
    assertEquals(3,
        new Negation(new Constant(-3)).eval());
}

@Test
public void testFactorialEval() {
    assertEquals(1, // 0!
        new Factorial(new Constant(0)).eval());
    assertEquals(120, // 5!
        new Factorial(
            new Sum(new Constant(1),
                new Constant(4))).eval());
}

```

NOTA: Come abbiamo visto a lezione (lezione su LSP), NON ha senso far ereditare *UnaryExpression* da *BinaryExpression* per riusare la definizione di *left* o *right*. Le due classi astratte hanno in comune solo il fatto di essere un *Expression*. Eventuali tentativi di riuso di questo tipo portano solo a svantaggi.

9 Alternativa (da evitare!)

La classe *BinaryExpression* si mantiene anche l'informazione sull'operatore dell'istanza dell'espressione binaria, ad es., con una stringa (o un enum?). L'operatore è passato al costruttore dalle sottoclassi in modo appropriato. *BinaryExpression* implementa *eval* in base all'operatore nel modo opportuno. (Notare che *BinaryExpression* è sempre *abstract* per non permettere che venga istanziata direttamente, anche se non ha nessun metodo astratto.)

Queste sono le classi modificate (rilanciando i test, tutti hanno successo):

```
public abstract class BinaryExpression implements Expression {

    private Expression left;
    private Expression right;
    private String operator;

    public BinaryExpression(Expression left, Expression right, String operator){
        this.left = left;
        this.right = right;
        this.operator = operator;
    }

    @Override
    public int eval() {
        int leftEval = left.eval();
        int rightEval = right.eval();
        switch (operator) {
            case "+":
                return leftEval + rightEval;
            case "*":
                return leftEval * rightEval;
            default:
                throw new UnsupportedOperationException(operator);
        }
    }
}

public class Sum extends BinaryExpression {

    public Sum(Expression left, Expression right) {
        super(left, right, "+");
    }
}

public class Multiplication extends BinaryExpression {

    public Multiplication(Expression left, Expression right) {
        super(left, right, "*");
    }
}
```

Come abbiamo visto a lezione, nelle slide del “*Open-Closed Principle*”, **soluzioni come questa sono DA EVITARE!!!** Peraltro, non sono soluzioni Object-Oriented (sempre per quanto visto a lezione).

10 Alternativa (non necessariamente migliore)

Un’alternativa per l’implementazione delle espressioni binarie è quella di implementare la logica “ricorsiva” direttamente in *BinaryExpression* astruendo dalla “composizione” dei risultati delle sottoespressioni e delegare alle sottoclassi l’implementazione della composizione dei due risultati intermedi.

Usando gli strumenti di Eclipse scriviamo prima l’implementazione di *eval* in termini del metodo astratto *composeEval*. Il codice non compila perché il metodo non esiste ancora. Usiamo il quickfix (es., con Ctrl+1) per crearlo. Dopo averlo creato, Eclipse ci dà la possibilità di modificare alcune parti della dichiarazione del nuovo metodo creato, evidenziando le parti modificabili. Per spostarsi fra le varie sezioni si usa il tasto Tab. Spostiamoci sul primo parametro e diamo un nome sensato, e poi ci spostiamo sull’altro facendo altrettanto:

```
public abstract class BinaryExpression implements Expression {  
  
    private Expression left;  
    private Expression right;  
  
    public BinaryExpression(Expression left, Expression right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    @Override  
    public int eval() {  
        return composeEval(left.eval(), right.eval());  
    }  
}  
  
@Override  
public int eval() {  
    return composeEval(left.eval(), right.eval());  
}  
  
protected abstract int composeEval(int leftEval, int eval2);
```

Alla fine otteniamo:

```
public abstract class BinaryExpression implements Expression {  
  
    private Expression left;  
    private Expression right;  
  
    public BinaryExpression(Expression left, Expression right) {  
        this.left = left;  
        this.right = right;  
    }  
}
```

```

@Override
public int eval() {
    return composeEval(left.eval(), right.eval());
}

protected abstract int composeEval(int leftEval, int rightEval);
}

```

Abbiamo intenzionalmente rimosso i *getter* in quanto adesso le sottoclassi non hanno bisogno di accedere alle sottoespressioni e si devono preoccupare solo di implementare il metodo astratto, che riceverà come argomenti i risultati intermedi delle due sottoespressioni.

Il codice delle sottoclassi non compilerà più ovviamente. Aggiustiamolo rimuovendo l'implementazione di *eval* e facendo creare lo stub per il nuovo metodo astratto (sempre col quickfix di Eclipse) e lo implementiamo in modo appropriato:

```

public class Sum extends BinaryExpression {

    public Sum(Expression left, Expression right) {
        super(left, right);
    }

    @Override
    protected int composeEval(int leftEval, int rightEval) {
        return leftEval + rightEval;
    }
}

```

```

public class Multiplication extends BinaryExpression {

    public Multiplication(Expression left, Expression right) {
        super(left, right);
    }

    @Override
    protected int composeEval(int leftEval, int rightEval) {
        return leftEval * rightEval;
    }
}

```

L'implementazione di *eval* in *BinaryExpression* potrebbe anche essere resa *final*, in modo da non permettere alle sottoclassi di ridefinire *eval*, ma solo di implementare il metodo astratto (vedremo a lezione che questo è simile a un pattern).

Ovviamente rilanciamo i test per assicurarci che tutto funzioni ancora. Notare che i test non devono essere ricompilati in quanto l'interfaccia pubblica delle varie espressioni non è cambiata.

Nelle sottoclassi non abbiamo guadagnato molto, in quanto dobbiamo comunque implementare la logica di quella particolare espressione, però

- non ci dobbiamo preoccupare di calcolare i valori intermedi (chiamate ricorsive)
- non è necessario accedere alle sottoespressioni, che sono completamente nascoste nella superclasse

Tuttavia potremmo aver perso un po' in leggibilità, in quanto abbiamo appunto perso la visione delle sottoespressioni.

Di volta in volta bisogna chiedersi se questa è una cosa positiva e se quello che si è guadagnato è abbastanza da compensare quello che si è perso.

In ogni caso, è fondamentale effettuare una tale analisi ed essere consapevoli delle conseguenze positive e negative.