Esercizio: Espressioni, Visitor e Static Factory Methods

IMPORTANTE: quest'esercitazione è MOLTO differente da quella su espressioni e visitor degli anni passati.

In questa esercitazione useremo il pattern *Visitor.* Partiremo da delle classi (e interfacce) per modellare delle espressioni simili a quelle viste nella prima esercitazione. In particolare gestiremo costanti intere e booleane, espressioni intere (somma di due sottoespressioni) e booleane (uguaglianza di due sottoespressioni e "and" logico di due sottoespressioni). Tali classi e interfacce per le espressioni non avranno metodi a parte i *getter* e i metodi *accept* previsti dal pattern *Visitor.*

In particolare, NON ci sarà il metodo *eval* in *Expression*: la valutazione delle espressioni sarà effettuata da una classe a parte che si basa sul pattern *Visitor.* Implementeremo un *Visitor* "generico", NON void (quindi i vari metodi del pattern restituiranno un valore, il cui tipo sarà appunto generico). Implementeremo un visitor concreto per la valutazione delle espressioni; la valutazione potrebbe fallire con un'eccezione nel caso non fosse *ben tipata* (es., se provassimo a sommare un intero con un booleano o a fare l'and di un booleano e un intero). In un secondo momento implementeremo anche un visitor concreto che rappresenterà il *type system* per le nostre espressioni: calcolerà il tipo di ogni espressione e effettuerà controlli di tipo, sollevando eccezioni nel caso un'espressione non sia *ben tipata* (es., vedere sopra).

Per rappresentare il risultato della valutazione e il concetto di "tipo" (risultato del type system) implementeremo delle classi apposite applicando il pattern *Static Factory Methods*.

Non dobbiamo mai usare "instanceof" e down-cast per queste implementazioni.

Anche questa esercitazione ha lo stesso schema delle altre volte. L'esercizio è suddiviso in diversi passi, da svolgere in sequenza. Inoltre, l'esercizio ha gli seguenti obiettivi delle altre esercitazioni. Ricordate che è fondamentale testare quello che si è implementato prima di passare al passo successivo.

1 Inizio

Per semplicità, è meglio non partire dalle classi e interfacce che avevamo già implementato durante la prima esercitazione, in quanto si dovrebbero fare prima diverse modifiche. Quindi usate il progetto di partenza che trovate su Moodle, *mp.exercise.expressions.visitor.partenza.zip*, e importatelo nel vostro workspace Eclipse (è uno zip, quindi ricordatevi di usare "Select archive file" quando importate il progetto). Il progetto è già configurato con JUnit 4 e con la cartella per i test.

In particolare partiremo dall'interfaccia *Expression* che è vuota.

```
package mp.exercise.expressions.visitor;

public interface Expression {
```

```
}
```

Le due classi per rappresentare le costanti intere e booleane:

```
package mp.exercise.expressions.visitor;

public class IntConstant implements Expression {

    private int value;

    public IntConstant(int value) {
        this.value = value;
    }

    public final int getValue() {
        return value;
    }
}
```

```
package mp.exercise.expressions.visitor;

public class BooleanConstant implements Expression {

    private boolean value;

    public BooleanConstant(boolean value) {
        this.value = value;
    }

    public final boolean getValue() {
        return value;
    }
}
```

Le classi *Sum*, *And* e *Equal* sottoclassi di *BinaryExpression* col solo costruttore che prende i due parametri per le sottoespressioni da passare al costruttore della classe base.

```
package mp.exercise.expressions.visitor;

public abstract class BinaryExpression implements Expression {

    private Expression left;
    private Expression right;

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

    public final Expression getLeft() {
        return left;
    }

    public final Expression getRight() {
        return right;
    }

}
```

Quindi, rispetto alle classi della prima esercitazione, non abbiamo il metodo *eval* (la valutazione avverrà tramite pattern *Visitor*). I metodi *getter* dovranno essere pubblici per permettere ai visitor concreti di accedere allo "stato" delle espressioni (che comunque rimarranno immutabili). Ricordate che i *getter* sono un compromesso da accettare quando si usa il pattern *Visitor.*

2 Rappresentare il risultato della valutazione

Implementare la classe *ExpressionEvalResult* con queste caratteristiche:

- la classe deve essere astratta e non deve essere possibile crearne sottoclassi dall'esterno
- le implementazioni concrete di tale classe NON devono essere accessibili dall'esterno
- deve essere istanziabile solo attraverso gli static factory methods seguenti (che creeranno risultati con all'interno un intero o un booleano, rispettivamente, passato come parametro):
  - public static ExpressionEvalResult ofInt(int value)
  - public static ExpressionEvalResult ofBoolean(boolean value)
- deve avere i metodi:
  - public int asInt()
  - public boolean asBoolean()

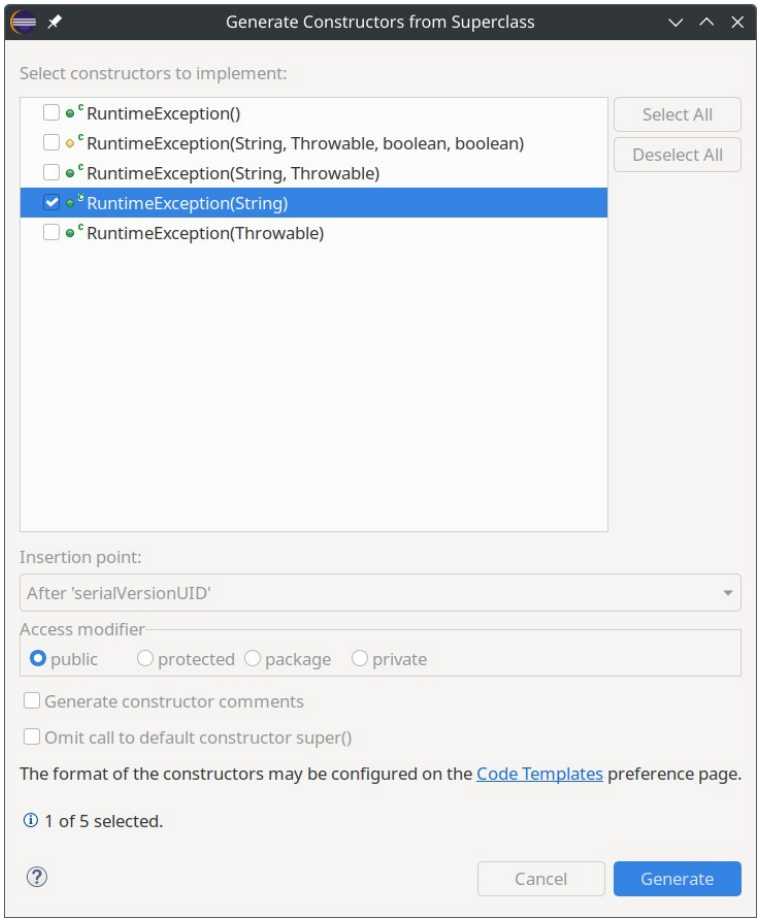
    che restituiscono l'intero o il booleano interni al risultato o sollevano un'eccezione non controllata *ExpressionEvalResultException* (che deve essere implementata) se si sta provando ad accedere al valore del tipo sbagliato (per il messaggio dell'eccezione, vedere sotto)
- deve avere il metodo (astratto):
  - public abstract Object getValue()

    che restituisce il valore interno al risultato, da implementare in modo appropriato.

In particolare, i seguenti test (da inserire in *ExpressionEvalResultTest*) devono avere successo:

```
@Test
public void testValidInt() {
    assertEquals(10, ExpressionEvalResult.ofInt(10).asInt());
}

@Test
public void testInvalidInt() {
    ExpressionEvalResultException thrown = assertThrows(ExpressionEvalResultException.class,
         () -> ExpressionEvalResult.ofBoolean(false).asInt());
    assertEquals("Not a valid int: false", thrown.getMessage());
}

@Test
public void testValidBoolean() {
    assertEquals(true, ExpressionEvalResult.ofBoolean(true).asBoolean());
}

@Test
public void testInvalidBoolean() {
    ExpressionEvalResultException thrown = assertThrows(ExpressionEvalResultException.class,
         () -> ExpressionEvalResult.ofInt(10).asBoolean());
    assertEquals("Not a valid boolean: 10", thrown.getMessage());
}
```

La classe dell'eccezione non controllata *ExpressionEvalResultException* può essere creata facilmente col wizard (attenzione, essendo "non controllata" deve estendere *RuntimeException*, NON *Exception*). Poi si usa il quickfix per il warning del *serialVersionUID*, per farci generare un serial id di default, e aggiungiamo il costruttore che prende la stringa col messaggio d'errore, tramite il menù "Generate Constructors from Superclass", selezionando il costruttore che interessa a noi:



Questa è una possibile implementazione di *ExpressionEvalResult* che rispetta tutti i punti sopra esposti e fa avere successo ai test. Notare l'uso degli static factory methods che nascondono completamente l'implementazione concreta tramite classe anonima. Notare che non è necessario l'uso di campi (ne' nella classe ne' nelle implementazioni anonime) perché l'oggetto della classe anonima è chiuso rispetto al parametro del ambito di visibilità circostante (come avverrebbe con le lambda expression). Le implementazioni di *asInt* e *asBoolean* di default sollevano l'eccezione; sono ridefiniti in modo appropriato nelle classi anonime.

```
package mp.exercise.expressions.visitor;

public abstract class ExpressionEvalResult {

    private ExpressionEvalResult() {
    }

    public abstract Object getValue();

    public int asInt() {
        throw new ExpressionEvalResultException("Not a valid int: " + getValue());
    }

    public boolean asBoolean() {
        throw new ExpressionEvalResultException("Not a valid boolean: " + getValue());
```

```
    }

    public static ExpressionEvalResult ofInt(int value) {
        return new ExpressionEvalResult() {
            @Override
            public int asInt() {
                return value;
            }

            @Override
            public Object getValue() {
                return value;
            }
        };
    }

    public static ExpressionEvalResult ofBoolean(boolean value) {
        return new ExpressionEvalResult() {
            @Override
            public boolean asBoolean() {
                return value;
            }

            @Override
            public Object getValue() {
                return value;
            }
        };
    }
}
```

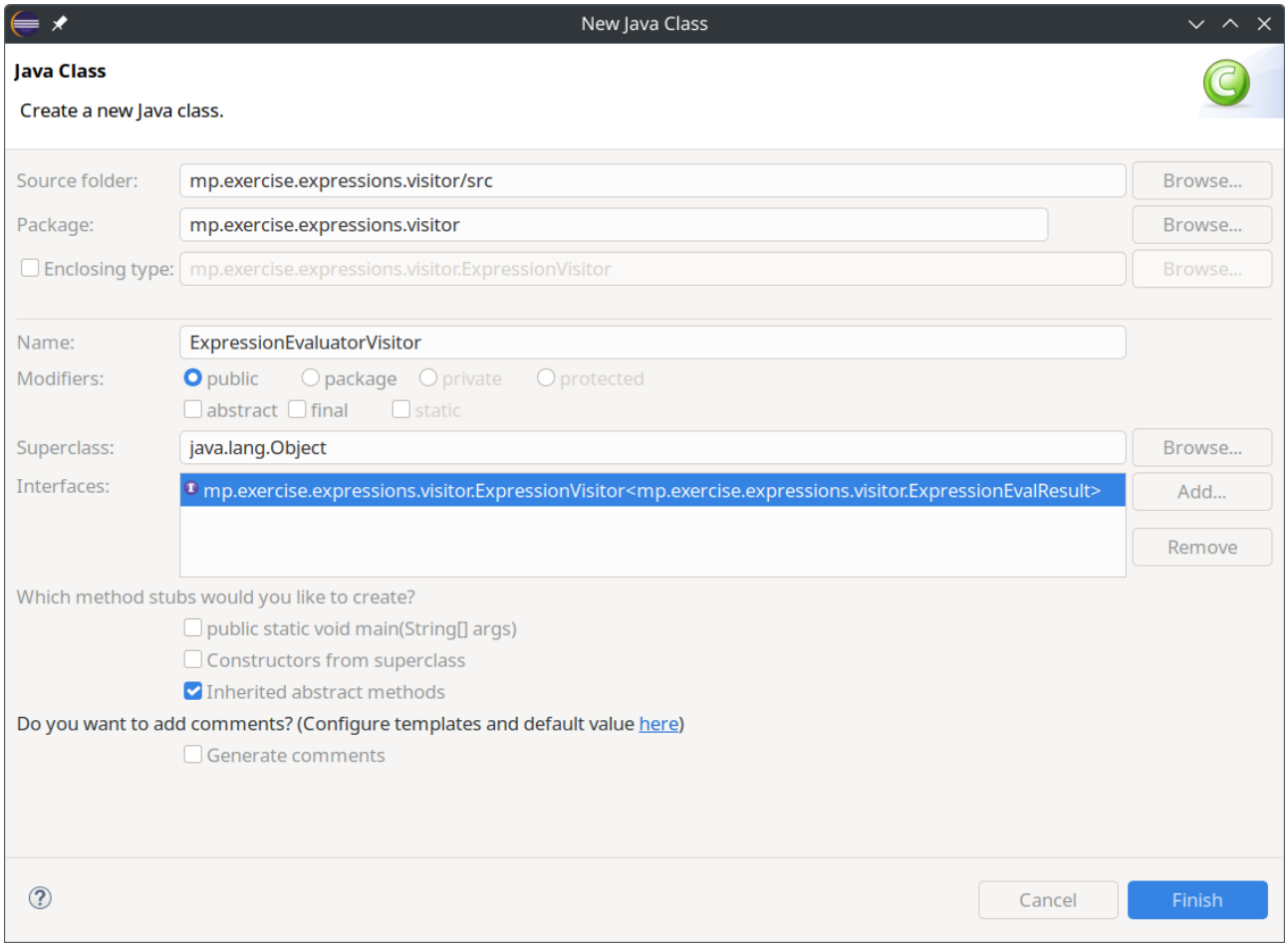
3 Valutazione di costanti intere

Dobbiamo creare un'interfaccia per il visitor generico, *ExpressionVisitor<T>*, coi vari *visitXXX*, es., *visitIntConstant*, *visitSum*, ecc. Questi metodi restituiscono il generico *T*. Poi dobbiamo creare il visitor concreto *ExpressionEvaluatorVisitor* implementare *visitIntConstant* per la valutazione di *IntConstant* (gli altri metodi *visit* possono essere lasciati per ora con l'implementazione di default che non fa niente, cioè, restituisce *null*). I metodi *visit* di questo visitor restituiscono un *ExpressionEvalResult*. Quindi come specifichiamo *ExpressionVisitor* in *implements* per questo visitor concreto? In questo passo basterà aggiungere *accept* solo alla classe *IntConstant* giusto per testare almeno questo caso.

Creiamo l'interfaccia per il visitor generico:

```
package mp.exercise.expressions.visitor;

public interface ExpressionVisitor<T> {

      T visitIntConstant(IntConstant intConstant);

      T visitBooleanConstant(BooleanConstant booleanConstant);

      T visitSum(Sum sum);

      T visitEqual(Equal equal);

      T visitAnd(And and);
}
```

Creiamo col wizard *ExpressionEvaluatorVisitor* che implementa questa interfaccia. L'argomento di tipo sarà *ExpressionEvalResult* (l'argomento deve essere specificato nella casella di testo che appare cliccando sull'interfaccia aggiunta nella finestra di dialogo; funziona il completamento di codice che inserirà il tipo completo di package). Ci facciamo creare le varie implementazioni temporanee dei vari metodi astratti:



Otteniamo questa classe:

```
package mp.exercise.expressions.visitor;

public class ExpressionEvaluatorVisitor implements
ExpressionVisitor<ExpressionEvalResult> {

    @Override
    public ExpressionEvalResult visitIntConstant(IntConstant intConstant) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public ExpressionEvalResult visitBooleanConstant(BooleanConstant booleanConstant) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public ExpressionEvalResult visitSum(Sum sum) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public ExpressionEvalResult visitEqual(Equal equal) {
```

```
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public ExpressionEvalResult visitAnd(And and) {
        // TODO Auto-generated method stub
        return null;
    }

}
```

Implementiamo il metodo per la costante intera seguendo il pattern *Visitor*, usando lo static factory method opportuno di *ExpressionEvalResult* per restituire il risultato:

```
@Override
public ExpressionEvalResult visitIntConstant(IntConstant intConstant) {
       return ExpressionEvalResult.ofInt(intConstant.getValue());
}
```

Ancora non abbiamo implementato la parte *accept* del pattern *Visitor*.

Scriviamo il seguente test (Data una classe, il corrispondente JUnit test può essere creato posizionandosi sul nome della classe e usando il Quick Assist, Ctrl + 1, e selezionando “Create new JUnit test case for…”):

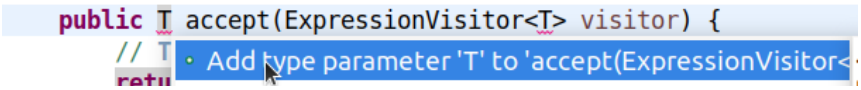
```
public class ExpressionEvaluatorVisitorTest {

    @Test
    public void testEvalIntConstant() {
      IntConstant e = new IntConstant(10);
      ExpressionVisitor<ExpressionEvalResult> visitor = new ExpressionEvaluatorVisitor();
      ExpressionEvalResult result = e.accept(visitor);
      assertEquals(10, result.getValue());
    }
}
```

Il test non compila perché *accept* non esiste in *IntConstant*. Lo creiamo in *IntConstant* tramite il quickfix di Eclipse. Dopo aver selezionato il quickfix per la creazione del metodo *accept* in *IntConstant* avremo la possibilità di spostarci sulle varie parti del metodo appena creato e come sempre ci possiamo spostare con Tab (e tornare indietro con Shift+Tab).

Modifichiamo il metodo creato come segue:

```
    public T accept(ExpressionVisitor<T> visitor) {
        // TODO Auto-generated method stub
        return null;
    }
```

Il metodo non compilerà perché *T* non è definito; usiamo il quickfix per aggiungere il parametro di tipo al metodo (NON alla classe):



ottenendo:

```
    public <T> T accept(ExpressionVisitor<T> visitor) {
        // TODO Auto-generated method stub
        return null;
    }
```

E implementiamolo secondo il pattern:

```
    public <T> T accept(ExpressionVisitor<T> visitor) {
        return visitor.visitIntConstant(this);
    }
```

Il test adesso compila e ha successo.

NOTA: ancora *accept* NON è nell'interfaccia *Expression*.

Nel test usiamo il refactoring "Inline" (imparate la shortcut da tastiera Alt+Shift+I) per liberarci di tutte le variabili locali intermedie (che vengono usate in un solo punto):

```
@Test
public void testEvalIntConstant() {
    assertEquals(10,
        new IntConstant(10)
            .accept(new ExpressionEvaluatorVisitor()).getValue());
}
```

NOTA: volendo potremmo creare un campo nella classe di test, ad es., *evaluatorVisitor*, di tipo *ExpressionEvaluatorVisitor*, che inizializziamo in un metodo annotato con *@Before*. In quel modo prima di ogni test il visitor sarà ricreato e possiamo usarlo direttamente nei test senza crearlo tutte le volte. Peraltro, all'interno di un singolo test possiamo riusare tranquillamente la stessa istanza del visitor in quanto, in questo esempio, il visitor non ha nessuno stato (la situazione sarebbe differente se avessimo dei visitor *void*, che, come abbiamo visto a lezione, di solito si mantengono uno stato). E a tal proposito potremmo addirittura rendere il campo *evaluatorVisitor* statico e inizializzarlo una volta per tutte in un metodo statico *@BeforeClass* invece che in un metodo *@Before*. Scegliete la strategia che preferite.

4 Portare accept nell'interfaccia

In questo passo portiamo la dichiarazione di *accept* nell'interfaccia *Expression*. Implementiamo uno a uno i metodi *accept* nelle varie sottoclassi, seguendo il pattern *Visitor*. I test precedenti devono continuare a passare. Al momento non possiamo testare le implementazioni dei vari *accept*. È comunque difficile sbagliare a scrivere tali implementazioni (il pattern *Visitor* si basa sul controllo statico dei tipi e commettere un errore nell'implementazione di un *accept* porterebbe a un errore da parte del compilatore). Attenzione: un refactoring di una versione (molto) vecchia di Eclipse ha un bug in presenza di metodi generici; se usate l'ultima versione di Eclipse invece non avrete problemi.

Portiamo con "Pull Up" la dichiarazione di *accept* dalla classe *IntConstant* nell'interfaccia *Expression* (specificando che si deve "portare su" la dichiarazione astratta del metodo).

```
public interface Expression {

    <T> T accept(ExpressionVisitor<T> visitor);

}
```

Il refactoring per lasciare il codice corrente compilabile, metterà un'implementazione vuota di *accept* nelle classi che implementano direttamente *Expression*, cioè in *BooleanConstant* e *BinaryExpression*:

```
public abstract class BinaryExpression implements Expression {

...

    @Override
    public <T> T accept(ExpressionVisitor<T> visitor) {
        // TODO Auto-generated method stub
        return null;
    }

}
```

Siccome non vogliamo un'implementazione di *accept* in *BinaryExpression*, ma nelle sue sottoclassi concrete, usiamo il refactoring "Push Down" (che è l'inverso di "Pull Up") in modo da mettere l'implementazione vuota di *accept* di *BinaryExpression* nelle sottoclassi di *BinaryExpression*, cioè *Sum*, *And*, e *Equal*.

Ora implementiamo uno a uno i metodi *accept* nelle varie sottoclassi, seguendo il pattern *Visitor.*

D'ora in poi, non dovremo toccare le classi delle espressioni: tutti gli *accept* sono già implementati in questo passo.

5 Valutazione di costanti booleane

Implementare la valutazione di *BooleanConstant*, similmente a come si è fatto per *IntConstant*.

In particolare, avendo nel passo precedente implementato tutti gli *accept*, si tratta di implementare *visitBooleanConstant*:

```
@Override
public ExpressionEvalResult visitBooleanConstant(BooleanConstant booleanConstant) {
      return ExpressionEvalResult.ofBoolean(booleanConstant.getValue());
}
```

e testare quest'implementazione come avevamo fatto per le costanti intere:

```
@Test
public void testEvalBooleanConstant() {
    assertEquals(true,
        new BooleanConstant(true)
            .accept(new ExpressionEvaluatorVisitor()).getValue());
}
```

6 Valutazione di Equal

Implementare la valutazione di *Equal*, che rappresenta l'uguaglianza, da intendersi come l'uguaglianza tramite *equals* delle valutazioni (ricorsivamente) delle due sottoespressioni. Il risultato *ExpressionEvalResult* deve ovviamente contenere un booleano. Come si effettua la

valutazione ricorsiva delle due sottoespressioni? NON ci si deve preoccupare del tipo delle due sottoespressioni (con *equals* possiamo confrontare anche oggetti di tipo differente, ovviamente il risultato sarà *false*). Scrivere anche diversi test per assicurarsi che la valutazione sia corretta. In particolare, testare anche il caso di uguaglianza di due sottoespressioni di tipo diverso (che deve essere *false*). Ricordate che al momento la valutazione è stata implementata solo per le costanti quindi nel test potrete testare solo *Equal* con due sottoespressioni di tipo *IntConstant* e *BooleanConstant*.

Implementiamo solo il metodo *visitEqual* in *ExpressionEvaluatorVisitor.* Vogliamo che il valutatore restituisca vero se la valutazione ricorsiva delle sottoespressioni è “uguale”, applicando *equals* al risultato della valutazione delle sottoespressioni. La valutazione delle sottoespressioni avviene richiedendo alle due sottoespressioni di “accettare” “questo” visitor; il risultato restituito da *equals* deve poi essere “avvolto” in un *ExpressionEvalResult*:

```
@Override
public ExpressionEvalResult visitEqual(Equal equal) {
    return ExpressionEvalResult.ofBoolean(
        equal.getLeft().accept(this).getValue()
            .equals(equal.getRight().accept(this).getValue()));
}
```

Avendo per ora implementato (e testato) solo la valutazione delle costanti, nei test possiamo solo usare oggetti *Equal* a cui passiamo due costanti. Testiamo vari scenari (quando due costanti intere sono uguali/diverse, due costanti booleani sono uguali/diverse, due costanti di tipo diverso):

```
@Test
public void testEvalEqualTrue() {
    assertTrue(
        new Equal(new IntConstant(10), new IntConstant(10))
            .accept(new ExpressionEvaluatorVisitor()).asBoolean());
    assertTrue(
        new Equal(new BooleanConstant(true), new BooleanConstant(true))
            .accept(new ExpressionEvaluatorVisitor()).asBoolean());
}

@Test
public void testEvalEqualFalse() {
    assertFalse(
        new Equal(new IntConstant(11), new IntConstant(10))
            .accept(new ExpressionEvaluatorVisitor()).asBoolean());
    assertFalse(
        new Equal(new BooleanConstant(false), new BooleanConstant(true))
            .accept(new ExpressionEvaluatorVisitor()).asBoolean());
}

@Test
public void testEvalEqualDifferentTypesFalse() {
    assertFalse(
        new Equal(new IntConstant(10), new BooleanConstant(true))
            .accept(new ExpressionEvaluatorVisitor()).asBoolean());
}
```

7 Valutazione di Sum

Implementare in *ExpressionEvaluatorVisitor* la valutazione di *Sum*. Ricordate che si tratta di un'operazione pensata per agire su interi. L'*ExpressionEvalResult* deve ovviamente contenere l'intero che rappresenta il risultato della somma. Si devono valutare prima le sottoespressioni e poi effettuare la somma dei due risultati intermedi "interpretandoli" come interi (assumendo che le due sottoespressioni siano *ben tipate* e del tipo giusto, intero). Nei test, sommando espressioni di tipo intero si deve controllare di ottenere il risultato che ci si aspetta; sommando espressioni di tipo differente si deve controllare di avere un'eccezione *ExpressionEvalResultException* col messaggio di errore che ci si aspetta.

```
@Override
public ExpressionEvalResult visitSum(Sum sum) {
    return ExpressionEvalResult.ofInt(
        sum.getLeft().accept(this).asInt()
        +
        sum.getRight().accept(this).asInt()
    );
}
```

Testiamo questa implementazione, sia quando ci si aspetta una somma valida, che quando si prova a sommare un intero e una costante booleana.

```
@Test
public void testEvalValidSum() {
    assertEquals(15,
        new Sum(new IntConstant(5), new IntConstant(10))
            .accept(new ExpressionEvaluatorVisitor()).asInt());
}

@Test
public void testEvalInvalidSum() {
    ExpressionEvalResultException thrown = assertThrows(ExpressionEvalResultException.class,
        () -> new Sum(new IntConstant(5), new BooleanConstant(true))
            .accept(new ExpressionEvaluatorVisitor()));
    assertEquals("Not a valid int: true", thrown.getMessage());
}
```

Riassumendo, quando valutiamo un'espressione assumiamo che sia corretta dal punto di vista dei tipi. Ovviamente, in un'applicazione reale, dovremmo documentare bene questa assunzione. Inoltre, come vedremo poi a lezione, l'idea è che in questa applicazione si dovrebbe valutare un'espressione solo dopo averla "tipata", cioè solo dopo che il nostro type system, che implementeremo nei prossimi punti, non ha sollevato problemi riguardo ai tipi.

8 Valutazione di And

Basandosi su quanto detto nel punto precedente, implementare in *ExpressionEvaluatorVisitor* la valutazione di *And*, pensata per agire su sottoespressioni booleane, restituendo come risultato l'and logico.

Come sopra, valutiamo le sottoespressioni e stavolta recuperiamo i valori booleani del risultato della valutazione delle sottoespressioni, assumendo che le sottoespressioni siano del tipo giusto.

```
@Override
public ExpressionEvalResult visitAnd(And and) {
    return ExpressionEvalResult.ofBoolean(
```

```
            and.getLeft().accept(this).asBoolean()
            &&
            and.getRight().accept(this).asBoolean()
    );
}
```

Come sopra, testiamo i vari casi:

```
@Test
public void testEvalValidAnd() {
    assertTrue(
        new And(new BooleanConstant(true), new BooleanConstant(true))
            .accept(new ExpressionEvaluatorVisitor()).asBoolean());
    assertFalse(
        new And(new BooleanConstant(true), new BooleanConstant(false))
            .accept(new ExpressionEvaluatorVisitor()).asBoolean());
}

@Test
public void testEvalInvalidAnd() {
    ExpressionEvalResultException thrown = assertThrows(ExpressionEvalResultException.class,
        () -> new And(new IntConstant(5), new BooleanConstant(true))
            .accept(new ExpressionEvaluatorVisitor()));
    assertEquals("Not a valid boolean: 5", thrown.getMessage());
}
```

9 Rappresentare i tipi

Implementare la classe *ExpressionType* con queste caratteristiche:

- la classe deve essere astratta e non deve essere possibile crearne sottoclassi dall'esterno
- le implementazioni concrete di tale classe NON devono essere accessibili dall'esterno
- deve esistere una sola implementazione che rappresenta il tipo intero e una sola implementazione che rappresenta il tipo booleano (NOTA: questo NON è il pattern *Singleton*)
- deve essere istanziabile solo attraverso gli static factory methods seguenti (che creeranno un tipo rappresentante un intero e un booleano, rispettivamente):
  - public static ExpressionType ofInt()
  - public static ExpressionType ofBoolean()
- deve avere i metodi:
  - public boolean isInt()
  - public boolean isBoolean()

    che dicono se il tipo rappresenta un intero o un booleano, rispettivamente
- il metodo *toString* dovrebbe restituire una stringa "int" o "boolean" a seconda del tipo

In particolare, questi test (in *ExpressionTypeTest*) devono avere successo:

```
@Test
public void testInt() {
    ExpressionType ofInt = ExpressionType.ofInt();
```

```
    assertTrue(ofInt.isInt());
    assertFalse(ofInt.isBoolean());
    assertEquals("int", ofInt.toString());
}

@Test
public void testBoolean() {
    ExpressionType ofBoolean = ExpressionType.ofBoolean();
    assertTrue(ofBoolean.isBoolean());
    assertFalse(ofBoolean.isInt());
    assertEquals("boolean", ofBoolean.toString());
}

@Test
public void testSameInstance() {
    assertSame(ExpressionType.ofBoolean(), ExpressionType.ofBoolean());
    assertSame(ExpressionType.ofInt(), ExpressionType.ofInt());
}
```

Questa è una possibile soluzione che rispetta tutti i punti sopra. Notare ancora una volta l'uso delle classi anonime usate direttamente per inizializzare le istanze dei due tipi. Possiamo implementare direttamente *isInt* e *isBoolean* basandosi sull'identità dei due oggetti *INT* e *BOOLEAN*. Questo è sicuro perché sappiamo che quei due oggetti sono unici (e ne abbiamo il pieno controllo). Le due implementazioni interne ridefiniscono in modo appropriato *toString*.

```
package mp.exercise.expressions.visitor;

public abstract class ExpressionType {

    private static final ExpressionType INT = new ExpressionType() {
        @Override
        public String toString() {
            return "int";
        }
    };

    private static final ExpressionType BOOLEAN = new ExpressionType() {
        @Override
        public String toString() {
            return "boolean";
        }
    };

    private ExpressionType() {
    }

    public boolean isInt() {
        return this == INT;
    }

    public boolean isBoolean() {
        return this == BOOLEAN;
    }

    public static ExpressionType ofInt() {
        return INT;
    }

    public static ExpressionType ofBoolean() {
        return BOOLEAN;
    }
```

```
}
```

10 Type system

Creare la classe *ExpressionTypeSystemVisitor* che implementa *ExpressionVisitor.* I vari metodi *visit* devono inferire (calcolare) il tipo delle espressioni, quindi che argomento di tipo si deve specificare?

Come sempre usiamo il wizard e ci facciamo generare le implementazioni "vuote" dei metodi astratti (ricordate di specificare *ExpressionType* come argomento di tipo per l'interfaccia *ExpressionVisitor*).

11 Typing di costanti

Implementare il typing di *IntConstant* e *BooleanConstant* in *ExpressionTypeSystemVisitor.* Il risultato sarà un *ExpressionType* appropriamente creato tramite gli static factory methods.

L'implementazione è semplicissima

```
@Override
public ExpressionType visitIntConstant(IntConstant intConstant) {
    return ExpressionType.ofInt();
}

@Override
public ExpressionType visitBooleanConstant(BooleanConstant booleanConstant) {
    return ExpressionType.ofBoolean();
}
```

Testare subito in una classe *ExpressionTypeSystemVisitorTest*:

```
@Test
public void testIntConstantType() {
    assertTrue(new IntConstant(10)
        .accept(new ExpressionTypeSystemVisitor()).isInt());
}

@Test
public void testBooleanConstantType() {
    assertTrue(new BooleanConstant(true)
        .accept(new ExpressionTypeSystemVisitor()).isBoolean());
}
```

12 Typing di Sum e And

Implementare il typing di *Sum*. Il tipo risultato sarà un *ExpressionType* che rappresenta il tipo intero. Però, prima, si deve controllare che le due sottoespressioni abbiano a loro volta *ExpressionType* che rappresenta il tipo intero, altrimenti si solleva un'eccezione non controllata *ExpressionTypeException* (che deve essere implementata) specificando che c'è un "type mismatch" in quanto ci si aspetta il tipo intero e invece si tratta di un altro tipo. Ad es. "Expected int but was

boolean”. Si consiglia di gestire le situazioni di errore dopo essersi assicurati che le situazioni corrette siano gestite correttamente. Poi si implementi il typing di *And*, che è come quello di *Sum*, ma *ExpressionType* deve rappresentare un booleano (così come le sue sottoespressioni).

Preoccupiamoci prima di *visitSum*. In prima istanza preoccupiamoci del caso “felice” assumendo che entrambe le sottoespressioni siano di tipo intero (senza visitarle) e banalmente implementiamo *visitSum* così:

```
@Override
public ExpressionType visitSum(Sum sum) {
	return ExpressionType.ofInt();
}
```

Scriviamo subito il test per questo caso (ovviamente, rispettando l’assunzione che le due sottoespressioni siano di tipo intero):

```
@Test
public void testSumType() {
	assertTrue(new Sum(new IntConstant(1), new IntConstant(10))
		.accept(new ExpressionTypeSystemVisitor()).isInt());
}
```

Adesso preoccupiamoci di effettuare il controllo sul tipo delle sottoespressioni; iniziamo con la sottoespressione di sinistra:

```
@Override
public ExpressionType visitSum(Sum sum) {
	ExpressionType leftType = sum.getLeft().accept(this);
	if (!leftType.isInt()) {
		throw new ExpressionTypeException("Expected int but was " + leftType);
	}
	return ExpressionType.ofInt();
}
```

Il codice non compila perché non esiste ancora la classe per la nostra eccezione. Facciamocela creare tramite quickfix. Vogliamo che sia un unchecked exception, quindi dobbiamo specificare *RuntimeException* come superclasse. Come già fatto per l’altra eccezione, usiamo il quickfix per far aggiungere il *serialVersionUID*. Aggiungiamo il costruttore che prende la stringa col messaggio d’errore (usare il menù “Generate Constructors from Superclass”, scegliendo il costruttore giusto).

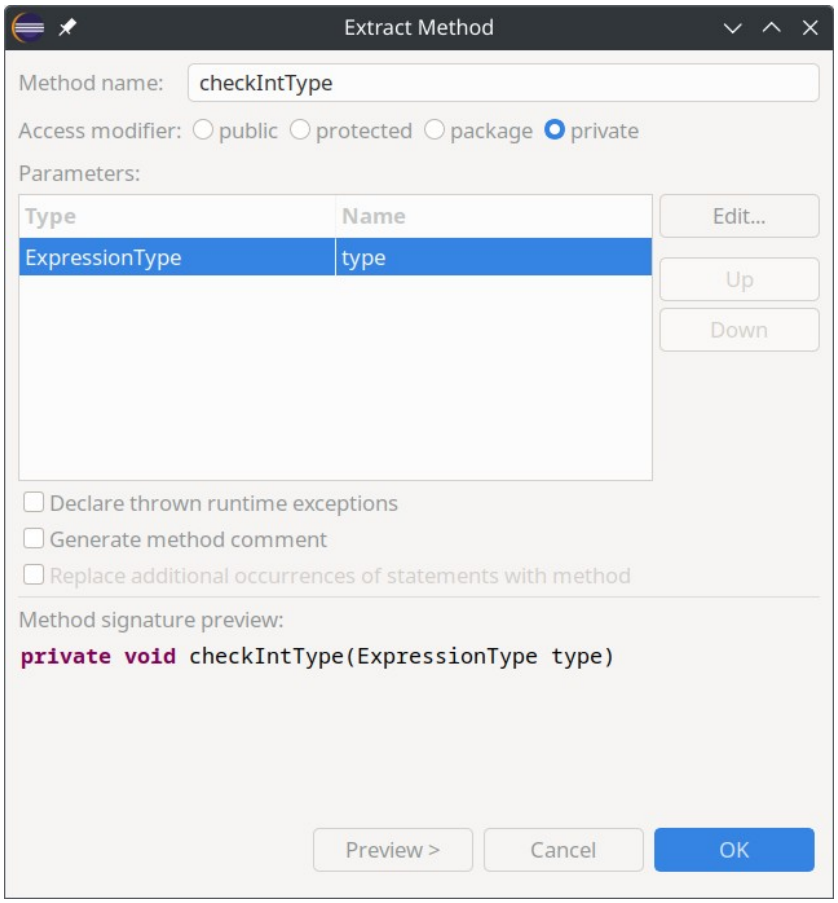
Adesso il codice del visitor compila.

Scriviamo un test che verifica che se l’espressione a sinistra non ha tipo intero si riceve l’eccezione col messaggio atteso:

```
@Test
public void testSumNonIntegerLeftType() {
	ExpressionTypeException thrown = assertThrows(ExpressionTypeException.class,
		() -> new Sum(new BooleanConstant(true), new IntConstant(10))
			.accept(new ExpressionTypeSystemVisitor()));
	assertEquals("Expected int but was boolean", thrown .getMessage());
}
```

Dovremo fare lo stesso col tipo della sottoespressione destra, quindi selezioniamo tutto l’if e usiamo “Extract Method”, dando un nome come *checkIntType* e diamo un nome più generico al parametro

che viene dato al metodo, es., *type*, visto che lo useremo anche per il tipo della sottoespressione destra (ovviamente il metodo sarà privato perché serve solo internamente):



Il risultato dopo il refactoring sarà:

```
@Override
public ExpressionType visitSum(Sum sum) {
	ExpressionType leftType = sum.getLeft().accept(this);
	checkIntType(leftType);
	return ExpressionType.ofInt();
}

private void checkIntType(ExpressionType type) {
	if (!type.isInt()) {
		throw new ExpressionTypeException("Expected int but was " + type);
	}
}
```

Facciamo l'inline della variabile locale *leftType* (usando il refactoring inline) e implementiamo il controllo a destra richiamando il metodo *checkIntType*:

```
@Override
public ExpressionType visitSum(Sum sum) {
	checkIntType(sum.getLeft().accept(this));
	checkIntType(sum.getRight().accept(this));
	return ExpressionType.ofInt();
}
```

Aggiungiamo il test apposito:

```
@Test
```

```
public void testSumNonIntegerRightType() {
	ExpressionTypeException thrown = assertThrows(ExpressionTypeException.class,
		() -> new Sum(new IntConstant(10), new BooleanConstant(true))
			.accept(new ExpressionTypeSystemVisitor()));
	assertEquals("Expected int but was boolean", thrown .getMessage());
}
```

Applicando la stessa metodologia (singoli passi e refactoring), implementiamo il typing di *And*, che alla fine dovrebbe essere:

```
@Override
public ExpressionType visitAnd(And and) {
	checkBooleanType(and.getLeft().accept(this));
	checkBooleanType(and.getRight().accept(this));
	return ExpressionType.ofBoolean();
}

private void checkBooleanType(ExpressionType type) {
	if (!type.isBoolean()) {
		throw new ExpressionTypeException("Expected boolean but was " + type);
	}
}
```

E i corrispettivi test:

```
@Test
public void testBooleanType() {
	assertTrue(new And(new BooleanConstant(true), new BooleanConstant(false))
		.accept(new ExpressionTypeSystemVisitor()).isBoolean());
}

@Test
public void testBooleanNonBooleanLeftType() {
	ExpressionTypeException thrown = assertThrows(ExpressionTypeException.class,
		() -> new And(new IntConstant(10), new BooleanConstant(true))
			.accept(new ExpressionTypeSystemVisitor()));
	assertEquals("Expected boolean but was int", thrown .getMessage());
}

@Test
public void testBooleanNonBooleanRightType() {
	ExpressionTypeException thrown = assertThrows(ExpressionTypeException.class,
		() -> new And(new BooleanConstant(true), new IntConstant(10))
			.accept(new ExpressionTypeSystemVisitor()));
	assertEquals("Expected boolean but was int", thrown .getMessage());
}
```

Volendo, potremmo anche scrivere un test per un’espressione molto complessa, visto che adesso il nostro type system gestisce tutte le espressioni, anche se non è necessario.

13 Typing di Equal

Il tipo di un’espressione *Equal* deve essere un *ExpressionType* che rappresenta un booleano, indipendentemente dal tipo delle due sottoespressioni. Comunque, le due sottoespressioni devono a loro volta essere ben tipate. Notare che NON ci dobbiamo preoccupare di controllare che le due sottoespressioni siano di un particolare tipo, ma che vengano tipate. Testare opportunamente, sia situazioni corrette che situazioni in cui ci si aspetta un’eccezione. Al momento nei test possiamo usare creare un *Equal* con qualsiasi altra sottoespressione, incluse *Sum* e *And*.

Partiamo dal gestire il caso più semplice, senza tipare le sottoespressioni:

```
@Override
public ExpressionType visitEqual(Equal equal) {
	return ExpressionType.ofBoolean();
}
```

E testiamo questo caso (notare che nel test le due sottoespressioni non sono dello stesso tipo):

```
@Test
public void testEqualType() {
	assertTrue(new Equal(new BooleanConstant(true), new IntConstant(10))
		.accept(new ExpressionTypeSystemVisitor()).isBoolean());
}
```

Adesso, assicuriamoci di visitare anche le due sottoespressioni, di cui non ci interessa il risultato (ma solo che non vengano sollevate eccezioni):

```
@Override
public ExpressionType visitEqual(Equal equal) {
	equal.getLeft().accept(this);
	equal.getRight().accept(this);
	return ExpressionType.ofBoolean();
}
```

Il test precedente deve ancora avere successo. Adesso ci assicuriamo che se ci sono dei problemi di tipo nelle sottoespressioni questi vengano rilevati quando si tipa un'espressione *Equal*:

```
@Test
public void testEqualInvalidLeftType() {
	ExpressionTypeException thrown = assertThrows(ExpressionTypeException.class,
		() ->
			new Equal(
				new And(new BooleanConstant(true), new IntConstant(10)),
				new IntConstant(0))
			.accept(new ExpressionTypeSystemVisitor()));
	assertEquals("Expected boolean but was int", thrown .getMessage());
}

@Test
public void testEqualInvalidRightType() {
	ExpressionTypeException thrown = assertThrows(ExpressionTypeException.class,
		() ->
			new Equal(
				new IntConstant(0),
				new And(new BooleanConstant(true), new IntConstant(10)))
			.accept(new ExpressionTypeSystemVisitor()));
	assertEquals("Expected boolean but was int", thrown .getMessage());
}
```

14 Riassumendo

Il software che abbiamo creato ci permette di creare delle espressioni e di valutarle. La valutazione però, come abbiamo visto, assume che l'espressione da valutare sia corretta dal punto di vista dei tipi (l'*ExpressionEvalutatorVisitor* può sollevare delle eccezioni durante la valutazione se le sottoespressioni non sono del tipo giusto). Infatti, prima di valutare un'espressione, il client

dovrebbe prima tiparla con un *ExpressionTypeSystemVisitor*: se non ottiene eccezioni, allora l'assunzione dell'*ExpressionEvalutatorVisitor* sarà rispettata.

Quindi il client è costretto a interagire col nostro sistema usando i due visitor e nell'ordine giusto quando vuole valutare un'espressione. Vedremo nelle prossime lezioni un pattern che permette di fornire un modo più semplice, e che nasconde ulteriori dettagli interni, per interagire col nostro sistema.

Notare che è possibile provare formalmente che il type system che abbiamo implementato e la valutazione sono "sound": data un'*Expression* se il type system computa un tipo senza eccezioni, allora la valutazione andrà a buon fine senza eccezioni.

Il sistema che abbiamo creato NON è pensato per essere esteso dai client, perché vogliamo avere il controllo sulle espressioni, sul loro tipaggio e valutazione. Se si deve introdurre una nuova espressione, sappiamo che dovremo mettere mano ad alcune nostre classi. Se i client non hanno scritto implementazioni di *ExpressionVisitor*, il loro codice continuerà a compilare senza problemi anche se introduciamo nuove espressioni.

15 Possibile estensione

Aggiungere *StringConstant*: aggiornare le classi per la valutazione (*ExpressionEvalResult* e *ExpressionEvalVisitor*) e per il type system (*ExpressionType* e *ExpressionTypeSystem*) in modo appropriato. Di due espressioni stringa, al momento, si può valutare solo l'uguaglianza.

Notare che le classi che devono essere cambiate saranno modificate aggiungendo nuovo codice: il codice esistente non deve essere modificato.