```
package dk.itu.smdp2015.church.validation

enum ExpressionType {
        String, Integer, Boolean
}
```

```
package dk.itu.smdp2015.church.validation


import dk.itu.smdp2015.church.model.configurator.Binary
import dk.itu.smdp2015.church.model.configurator.Boolean
import dk.itu.smdp2015.church.model.configurator.Bounded
import dk.itu.smdp2015.church.model.configurator.Constant
import dk.itu.smdp2015.church.model.configurator.Enumerated
import dk.itu.smdp2015.church.model.configurator.Identifier
import dk.itu.smdp2015.church.model.configurator.InRange
import dk.itu.smdp2015.church.model.configurator.Integer
import dk.itu.smdp2015.church.model.configurator.String
import dk.itu.smdp2015.church.model.configurator.Unary
import dk.itu.smdp2015.church.model.configurator.ValueRange

import static dk.itu.smdp2015.church.model.configurator.BinaryOperator.*
import static dk.itu.smdp2015.church.model.configurator.UnaryOperator.*

class ExpressionTypeProvider {

        def dispatch ExpressionType typeFor(Constant constant) {
                switch (constant) {
                        String: ExpressionType.String
                        Boolean: ExpressionType.Boolean
                        Integer: ExpressionType.Integer
                }
        }

        def dispatch ExpressionType typeFor(Binary binary) {
                switch (binary.operator) {
                        case ADDITION:
                                ExpressionType.Integer
                        case LOGICAL_AND:
                                ExpressionType.Boolean
                        case LOGICAL_OR:
                                ExpressionType.Boolean
                        case EQUAL:
                                ExpressionType.Boolean
                        case GREATER_THAN:
                                ExpressionType.Boolean
                        case LESS_THAN:
                                ExpressionType.Boolean
                        case MULTIPLICATION:
                                ExpressionType.Integer
                        case NOT_EQUAL:
                                ExpressionType.Boolean
                        case SUBTRACTION:
                                ExpressionType.Integer
                }
        }

        def dispatch ExpressionType typeFor(Unary unary) {
                switch (unary.operator) {
                        case INVERSION: ExpressionType.Integer
                        case LOGICAL_NOT: ExpressionType.Boolean
                }
        }

        def dispatch ExpressionType typeFor(InRange inrange) {
                ExpressionType.Boolean
        }

        def dispatch ExpressionType typeFor(Identifier identifier) {
                identifier.id.valueRange?.rangeType
        }

        def ExpressionType rangeType(ValueRange range) {
                switch (range) {
                        Enumerated: range.values.get(0)?.typeFor
                        Bounded: range.lowerBound?.typeFor
                }
        }
}
```

```
package dk.itu.smdp2015.church.validation


import dk.itu.smdp2015.church.model.configurator.Binary
import dk.itu.smdp2015.church.model.configurator.Constant
import dk.itu.smdp2015.church.model.configurator.Identifier
import dk.itu.smdp2015.church.model.configurator.InRange
import dk.itu.smdp2015.church.model.configurator.String
import dk.itu.smdp2015.church.model.configurator.Unary

import static dk.itu.smdp2015.church.model.configurator.BinaryOperator.*
import static dk.itu.smdp2015.church.model.configurator.UnaryOperator.*

class ExpressionValueProvider {

        def dispatch Object staticValue(Constant constant) {
                switch (constant) {
                        String: constant.value
                        dk.itu.smdp2015.church.model.configurator.Boolean: constant.value
                        dk.itu.smdp2015.church.model.configurator.Integer: constant.value
                }
        }

        def dispatch Object staticValue(Binary binary) {
                val vleft = binary.left.staticValue
                val vright = binary.right.staticValue
                switch (binary.operator) {
                        case ADDITION:
                                if (vleft instanceof Integer && vright instanceof Integer) {
                                        val ileft = (vleft as Integer).intValue
                                        val iright = (vright as Integer).intValue
                                        new Integer(ileft + iright)
                                }
                        case LOGICAL_AND:
                                if (vleft instanceof Boolean && vright instanceof Boolean) {
                                        val bleft = (vleft as Boolean).booleanValue
                                        val bright = (vright as Boolean).booleanValue
                                        new Boolean(bleft && bright)
                                }
                        case LOGICAL_OR:
                                if (vleft instanceof Boolean && vright instanceof Boolean) {
                                        val bleft = (vleft as Boolean).booleanValue
                                        val bright = (vright as Boolean).booleanValue
                                        new Boolean(bleft || bright)
                                }
                        case EQUAL:
                                if (vleft != null && vleft.class.equals(vright.class)) {
                                        new Boolean(vleft.equals(vright))
                                }
                        case GREATER_THAN:
                                if (vleft instanceof Integer && vright instanceof Integer) {
                                        val ileft = (vleft as Integer).intValue
                                        val iright = (vright as Integer).intValue
                                        new Boolean(ileft > iright)
                                }
                        case LESS_THAN:
                                if (vleft instanceof Integer && vright instanceof Integer) {
                                        val ileft = (vleft as Integer).intValue
                                        val iright = (vright as Integer).intValue
                                        new Boolean(ileft > iright)
                                }
                        case MULTIPLICATION:
                                if (vleft instanceof Integer && vright instanceof Integer) {
                                        val ileft = (vleft as Integer).intValue
                                        val iright = (vright as Integer).intValue
                                        new Integer(ileft * iright)
                                }
                        case NOT_EQUAL:
                                if (vleft != null && vleft.class.equals(vright.class)) {
                                        new Boolean(!vleft.equals(vright))
                                }
                        case SUBTRACTION:
                                if (vleft instanceof Integer && vright instanceof Integer) {
                                        val ileft = (vleft as Integer).intValue
                                        val iright = (vright as Integer).intValue
                                        new Integer(ileft - iright)
                                }
                }
        }

        def dispatch Object staticValue(Unary unary) {
                val vinner = unary.inner.staticValue
                switch (unary.operator) {
                        case INVERSION:
                                if (vinner instanceof Integer) {
                                        val iinner = (vinner as Integer).intValue
                                        new Integer(-iinner)
                                }
                        case LOGICAL_NOT:
                                if (vinner instanceof Boolean) {
                                        val binner = (vinner as Boolean).booleanValue
                                        new Boolean(!binner)
                                }
                }
        }

        def dispatch ExpressionType staticValue(InRange inrange) {
                null
        }

        def dispatch ExpressionType staticValue(Identifier identifier) {
                null
        }
}
```

```
package dk.itu.smdp2015.church.validation


import dk.itu.smdp2015.church.model.configurator.Binary
import dk.itu.smdp2015.church.model.configurator.Bounded
import dk.itu.smdp2015.church.model.configurator.ConfiguratorPackage
import dk.itu.smdp2015.church.model.configurator.Constraint
import dk.itu.smdp2015.church.model.configurator.Enumerated
import dk.itu.smdp2015.church.model.configurator.Expression
import dk.itu.smdp2015.church.model.configurator.InRange
import dk.itu.smdp2015.church.model.configurator.Unary
import dk.itu.smdp2015.church.model.configurator.ValueRange
import javax.inject.Inject
import org.eclipse.emf.ecore.EReference
import org.eclipse.xtext.validation.Check

import static dk.itu.smdp2015.church.model.configurator.BinaryOperator.*
import static dk.itu.smdp2015.church.model.configurator.UnaryOperator.*
import dk.itu.smdp2015.church.model.configurator.Parameter

/**
 * Custom validation rules.
 *
 * see http://www.eclipse.org/Xtext/documentation.html#validation
 */
class ConfiguratorValidator extends AbstractConfiguratorValidator {

        public static val INVALID_BOUND = 'invalidBound'
        public static val INVALID_ENUMERATION = 'invalid enumeration'
        public static val INVALID_BINARYTYPE = 'invalid binary operand type'
        public static val WRONG_TYPE = "dk.itu.smdp2015.church.WrongType"

        @Inject extension ExpressionTypeProvider
        @Inject extension ExpressionValueProvider

        @Check
        def checkEnumeratedExpressionIsConstant(Enumerated it) {
                values.forEach[
                        if (staticValue == null) {
                                error('Enumerated item should be a constant.', 
ConfiguratorPackage.Literals.ENUMERATED__VALUES,
                                        INVALID_ENUMERATION)
                        }]
        }

        @Check
        def checkBoundedExpressionUpperBoundIsConstant(Bounded bounded) {
                if (bounded.upperBound.staticValue == null) {
                        error('Upper bound should be a constant.', ConfiguratorPackage.Literals.BOUNDED__UPPER_BOUND,
                                INVALID_BOUND)
                }
        }

        @Check
        def checkBoundedExpressionLowerBoundIsConstant(Bounded bounded) {
                if (bounded.lowerBound.staticValue == null) {
                        error('Lower bound should be a constant.', ConfiguratorPackage.Literals.BOUNDED__LOWER_BOUND,
                                INVALID_BOUND)
                }
        }

        @Check
        def checkBoundedExpressionLowerIsBelowUpper(Bounded bounded) {
                val lowerVal = bounded.lowerBound?.staticValue
                val upperVal = bounded.upperBound?.staticValue
                var c = -1;
                if (lowerVal instanceof Integer && upperVal instanceof Integer) {
                        c = (lowerVal as Integer).compareTo(upperVal as Integer)
                }
                if (lowerVal instanceof String && upperVal instanceof String) {
                        c = (lowerVal as String).compareTo(upperVal as String)
                }
                if (lowerVal instanceof Boolean && upperVal instanceof Boolean) {
                        c = (lowerVal as Boolean).compareTo(upperVal as Boolean)
                }
                if (c >= 0) {
                        error('Lower bound should be less than upper bound', 
ConfiguratorPackage.Literals.BOUNDED__LOWER_BOUND,
                                INVALID_BOUND)
                }
        }

        @Check
        def checkEnumeratedSequence(Enumerated enumerated) {
                enumerated.values.forEach [ v |
                        if (enumerated.values.filter[staticValue == v.staticValue].size != 1)
                                error('Enumerated values should be unique',
ConfiguratorPackage.Literals.ENUMERATED__VALUES,
                                        INVALID_BOUND)
                ]
        }

        @Check
        def checkType(Constraint constraint) {
                val literal = ConfiguratorPackage.Literals.CONSTRAINT__EXPRESSION
                val type = getTypeAndCheckNotNull(constraint.expression, literal)
                checkExpectedType(type, ExpressionType.Boolean, literal)
        }

        @Check
        def checkType(Binary binary) {
                val leftLiteral = ConfiguratorPackage.Literals.BINARY__LEFT
                val rightLiteral = ConfiguratorPackage.Literals.BINARY__RIGHT
                val binaryLiteral = ConfiguratorPackage.Literals.BINARY__OPERATOR
                val leftType = getTypeAndCheckNotNull(binary.left, leftLiteral)
                val rightType = getTypeAndCheckNotNull(binary.right, rightLiteral)
                switch (binary.operator) {
                        case ADDITION: {
                                checkExpectedType(leftType, ExpressionType.Integer, leftLiteral)
                                checkExpectedType(rightType, ExpressionType.Integer, rightLiteral)
                        }
                        case LOGICAL_AND: {
                                checkExpectedType(leftType, ExpressionType.Boolean, leftLiteral)
                                checkExpectedType(rightType, ExpressionType.Boolean, rightLiteral)
                        }
                        case LOGICAL_OR: {
                                checkExpectedType(leftType, ExpressionType.Boolean, leftLiteral)
                                checkExpectedType(rightType, ExpressionType.Boolean, rightLiteral)
                        }
                        case EQUAL: {
                                if (leftType != rightType) {
                                        error("expected the same type, but the types are " + leftType + " and " +
rightType, binaryLiteral,
                                                WRONG_TYPE)
                                }
                        }
                        case GREATER_THAN: {
                                checkExpectedType(leftType, ExpressionType.Integer, leftLiteral)
                                checkExpectedType(rightType, ExpressionType.Integer, rightLiteral)
                        }
                        case LESS_THAN: {
                                checkExpectedType(leftType, ExpressionType.Integer, leftLiteral)
                                checkExpectedType(rightType, ExpressionType.Integer, rightLiteral)
                        }
                        case MULTIPLICATION: {
                                checkExpectedType(leftType, ExpressionType.Integer, leftLiteral)
                                checkExpectedType(rightType, ExpressionType.Integer, rightLiteral)
                        }
                        case NOT_EQUAL: {
                                if (leftType != rightType) {
                                        error("expected the same type, but the types are " + leftType + " and " +
rightType, binaryLiteral,
                                                WRONG_TYPE)
                                }
                        }
                        case SUBTRACTION: {
                                checkExpectedType(leftType, ExpressionType.Integer, leftLiteral)
                                checkExpectedType(rightType, ExpressionType.Integer, rightLiteral)
                        }
                }
        }

        @Check
        def checkType(Unary unary) {
                val innerLiteral = ConfiguratorPackage.Literals.UNARY__INNER
                val innerType = getTypeAndCheckNotNull(unary.inner, innerLiteral)
        switch (unary.operator) {
            case INVERSION: {
                checkExpectedType(innerType, ExpressionType.Integer, innerLiteral)
            }
            case LOGICAL_NOT: {
                checkExpectedType(innerType, ExpressionType.Boolean, innerLiteral)
            }
        }
    }

    @Check
    def checkType(InRange inRange) {
        val leftLiteral = ConfiguratorPackage.Literals.IN_RANGE__PARAMETER
        val rightLiteral = ConfiguratorPackage.Literals.IN_RANGE__RANGE
        val leftType = getTypeAndCheckNotNull(inRange.parameter.valueRange, leftLiteral)
        val rightType = getTypeAndCheckNotNull(inRange.range, rightLiteral)
        if (leftType != rightType) {
            error("expected the same type, but the types are " + leftType + " and " + rightType, rightLiteral,
                WRONG_TYPE)
        }
    }

    @Check
    def checkType(Enumerated range) {
        val firstType = getTypeAndCheckNotNull(range.values.get(0),
ConfiguratorPackage.Literals.ENUMERATED__VALUES)
        range.values.forEach [
            val nextType = getTypeAndCheckNotNull(ConfiguratorPackage.Literals.ENUMERATED__VALUES)
            if (firstType != nextType)
                error("expected the same type, but the types are " + firstType + " and " + nextType,
                    ConfiguratorPackage.Literals.ENUMERATED__VALUES, WRONG_TYPE)
        ]
    }

    @Check
    def checkType(Bounded range) {
        val lowerType = getTypeAndCheckNotNull(range.lowerBound,
ConfiguratorPackage.Literals.BOUNDED__LOWER_BOUND)
        val upperType = getTypeAndCheckNotNull(range.upperBound,
ConfiguratorPackage.Literals.BOUNDED__UPPER_BOUND)
        if (lowerType != upperType) {
            error("expected the same type, but the types are " + lowerType + " and " + upperType,
                ConfiguratorPackage.Literals.BOUNDED__UPPER_BOUND, WRONG_TYPE)
        }
    }

    @Check
    def checkType(Parameter parameter) {
        if (parameter.^default != null) {
            val defType = getTypeAndCheckNotNull(parameter.^default,
ConfiguratorPackage.Literals.PARAMETER__DEFAULT)
            val rangeType = getTypeAndCheckNotNull(parameter.valueRange,
                ConfiguratorPackage.Literals.PARAMETER__VALUE_RANGE)
            if (defType != rangeType) {
                error("expected the same type, but the types are " + defType + " and " + rangeType,
                    ConfiguratorPackage.Literals.PARAMETER__DEFAULT, WRONG_TYPE)
            }
        }
    }

    @Check
    def checkDefaultValue(Parameter parameter) {
        if (parameter.^default != null) {
            val defVal = parameter.^default.staticValue
            val range = parameter.valueRange
            switch (range) {
                Enumerated:
                    if (!range.values.exists[staticValue == defVal])
                        error('Default value should be among the listed values',
                            ConfiguratorPackage.Literals.PARAMETER__DEFAULT, INVALID_BOUND)
                // Bounded: to be done...
            }
        }
    }

    def private checkExpectedType(ExpressionType actualType, ExpressionType expectedType, EReference reference) {
        if (actualType != expectedType) {
            error("expected type " + expectedType + ", actual type is " + actualType, reference, WRONG_TYPE)
```

```
                }
        }

        def private ExpressionType getTypeAndCheckNotNull(Expression expression, EReference reference) {
                var type = expression?.typeFor
                if (type == null)
                        error("unknown type", reference, WRONG_TYPE)
                type
        }

        def private ExpressionType getTypeAndCheckNotNull(ValueRange range, EReference reference) {
                var type = range?.rangeType
                if (type == null)
                        error("unknown type", reference, WRONG_TYPE)
                type
        }
}
```