获取代码的位置信息（Working with source locations）

您可以使用Java代码中实体的位置来查找潜在的错误。 位置允许您推断是否存在空白，在某些情况下，空白可能表明存在问题。

关于源位置（About source locations）

Java提供了一组具有复杂优先级规则的运算符，这些规则有时会使开发人员感到困惑。 例如，OpenJDK Java编译器中的ByteBufferCache类（它是com.sun.tools.javac.util.BaseFileManager的成员类）包含以下用于分配缓冲区的代码：

```
1 ByteBuffer.allocate(capacity + capacity>>1)
```

据推测，作者打算分配一个缓冲区，该缓冲区的大小是可变容量指示的大小的1.5倍。 但是，实际上，运算符+的绑定比运算符>>的绑定更紧密，因此表达式容量+容量>> 1被解析为（容量+容量）>> 1，等于容量（除非存在算术上溢）。

请注意，源代码布局清楚地表明了预期的含义：+周围的空格比>>周围的空格要多，这表明后者的意思是要更紧密地绑定。

我们将开发一个查询来查找这种可疑的嵌套，其中内部表达式的运算符周围的空格比外部表达式的运算符多。 这种模式不一定表明存在错误，但至少会使代码难以阅读并且容易产生误解。

空白不是直接在CodeQL数据库中表示的，但是我们可以从与程序元素和AST节点关联的位置信息中推断出空格的存在。 因此，在编写查询之前，我们需要了解Java标准库中的源位置管理。

定位API（Location API）

对于每个使用Java源代码表示的实体（尤其包括程序元素和AST节点），标准CodeQL库提供了以下谓词来访问源位置信息：

- getLocation返回一个Location对象，该对象描述实体的开始和结束位置。
- getFile返回一个File对象，该对象表示包含实体的文件。
- getTotalNumberOfLines返回实体源代码跨越的行数。
- getNumberOfCommentLines返回注释行的数量。
- getNumberOfLinesOfCode返回非注释行的数量。

例如，假设此Java类是在编译单元SayHello.java中定义的：

```
package pkg;

class SayHello {
    public static void main(String[] args) {
        System.out.println(
            // Display personalized message
            "Hello, " + args[0];
        );
    }
}
```

在main主体中的expression语句上调用getFile返回代表文件SayHello.java的File对象。 该语句总共跨越四行（getTotalNumberOfLines），其中一行是注释行（getNumberOfCommentLines），而三行包含代码（getNumberOfLinesOfCode）。

类位置定义成员谓词getStartLine，getEndLine，getStartColumn和getEndColumn，以分别检索实体开始和结束的行号和列号。 行和列都从1（不是0）开始计数，结束位置是包含端点的，也就是说，它是属于实体源代码的最后一个字符的位置。

在我们的示例中，expression语句从第5行第3列开始（该行的前两个字符是制表符，每个都视为一个字符），并在第8行第4列结束。

类文件定义了这些成员谓词：

- getAbsolutePath返回文件的标准名称。
- getRelativePath返回文件相对于源代码基本目录的路径。
- getExtension返回文件的扩展名。
- getStem返回文件的基本名称，不带扩展名。

在我们的示例中，假设文件A.java位于目录/ home / testuser / code / pkg中，其中/ home / testuser / code是要分析的程序的基本目录。 然后，用于A.java的File对象返回：

- getAbsolutePath是/home/testuser/code/pkg/A.java。
- getRelativePath是pkg / A.java。
- getExtension是java。

- getStem是A。

确定运算符周围的空白(Determining white space around an operator)

让我们首先考虑如何编写一个谓词，该谓词计算给定二进制表达式的运算符周围的空白总数。 如果rcol是表达式的右操作数的开始列，而lcol是表达式的左操作数的结束列，则rcol–（lcol + 1）给出两个操作数之间的字符总数（请注意，我们必须使用lcol +1而不是lcol，因为结尾位置包括在内）。

此数字包括运算符本身的长度，我们需要减去该长度。 为此，我们可以使用谓词getOp，它返回运算符字符串，在运算符字符串的两边都用一个空格包围。 总体而言，用于计算二进制表达式expr的运算符周围的空白量的表达式为：

```
rcol - (lcol+1) - (expr.getOp().length()-2)
```

但是，很明显，这仅在整个表达式位于一行上时才有效，我们可以使用上面介绍的谓词getTotalNumberOfLines进行检查。 我们现在可以定义用于计算运算符周围空白的谓词：

```
int operatorWS(BinaryExpr expr) {
    exists(int lcol, int rcol |
        expr.getNumberOfLinesOfCode() = 1 and
        lcol = expr.getLeftOperand().getLocation().getEndColumn()
and
        rcol = expr.getRightOperand().getLocation().getStartColumn
() and
        result = rcol - (lcol+1) - (expr.getOp().length()-2)
    )
}
```

请注意，我们使用存在来引入我们的临时变量lcol和rcol。 您可以通过仅将lcol和rcol内联到它们的使用中来编写谓词，而无需使用它们，但要付出一些可读性。

查找可疑的嵌套(Find suspicious nesting)

这是我们查询的第一个版本：

```
import java

// Insert predicate defined above
```

```
4
5 from BinaryExpr outer, BinaryExpr inner,
6     int wsouter, int wsinner
7 where inner = outer.getAChildExpr() and
8     wsinner = operatorWS(inner) and wsouter = operatorWS(outer) a
  nd
9     wsinner > wsouter
10 select outer, "Whitespace around nested operators contradicts pre
  cedence."
```

where子句的第一个联合词将inner限制为外部操作数，第二个联合词绑定wsinner和wsouter，而最后一个联合词选择可疑情况。

首先，我们可能很想在第一个合取物中编写inner = external.getAnOperand（）。 但是，这并不完全正确：getAnOperand会从结果中去除所有括号，这通常是有用的，但并不是我们想要的：如果内部表达式周围有括号，那么程序员可能知道它们是什么 这样做，并且查询不应标记该表达式。

优化查询（Improving the query）

如果运行此初始查询，我们可能会注意到由不对称空格引起的一些误报。 例如，以下表达式被标记为可疑，尽管在实践中不太可能引起混淆：

```
1 i< start + 100
```

请注意，我们的谓词operatorWS计算了运算符周围的空格总量，在这种情况下，<表示一个空格，+表示两个空格。 理想情况下，我们希望排除运算符前后的空白量不同的情况。 目前，CodeQL数据库记录的信息不足以解决这个问题，但是作为近似值，我们可能要求空格字符的总数为偶数：

```
1 import java
2
3 // Insert predicate definition from above
4
5 from BinaryExpr outer, BinaryExpr inner,
6     int wsouter, int wsinner
7 where inner = outer.getAChildExpr() and
8     wsinner = operatorWS(inner) and wsouter = operatorWS(outer) a
  nd
9     wsinner % 2 = 0 and wsouter % 2 = 0 and
```

```
    wsinner > wsouter
select outer, "Whitespace around nested operators contradicts precedence."
```

误报的另一个来源是关联运算符：在x + y + z形式的表达式中，第一个加号在语法上嵌套在第二个加号内，这是因为Java中的+关联到左侧。 因此该表达式被标记为可疑。 但是，由于+开头是关联的，因此嵌套操作符的方式无关紧要，因此这是一个误报。 为了排除这些情况，让我们定义一个新类，该类使用关联运算符标识二进制表达式：

```
class AssociativeOperator extends BinaryExpr {
    AssociativeOperator() {
        this instanceof AddExpr or
        this instanceof MulExpr or
        this instanceof BitwiseExpr or
        this instanceof AndLogicalExpr or
        this instanceof OrLogicalExpr
    }
}
```

现在，我们可以扩展查询以丢弃外部表达式和内部表达式都具有相同的关联运算符的结果：

```
import java

// Insert predicate and class definitions from above

from BinaryExpr inner, BinaryExpr outer, int wsouter, int wsinner
where inner = outer.getAChildExpr() and
    not (inner.getOp() = outer.getOp() and outer instanceof AssociativeOperator) and
    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and
    wsinner % 2 = 0 and wsouter % 2 = 0 and
    wsinner > wsouter
select outer, "Whitespace around nested operators contradicts precedence."
```

请注意，这次我们再次使用getOp来确定两个二进制表达式是否具有相同的运算符。 现在，运行我们经过改进的查询可以找到概述中所述的Java标准库错误。 它还在以下位置标记了以下可疑代码

```
KEY_SLAVE = tmp[ i+1 % 2 ];
```

Whitespace建议程序员打算在零和一之间切换i，但实际上该表达式被解析为i +（1%2），与i + 1相同，因此可以简单地对i进行递增。