Software Development (cs2500)

Lectures 4 and 5: Expressions, Conditions, and Iteration

M.R.C. van Dongen

October 4, 2010

Contents

1 Outline

These lecture notes do not correspond to any particular part in the book. The main purpose of them is to fill in some of the gaps which aren't explained in the book. After studying these notes you should be able to write basic arithmetic expressions in Java, know how Java's evaluates its expressions, understand the notions of *precedence* and *associativity*, make decisions using `if` statements, use Java's bounded and unbounded iteration constructs, and be able to put basic *invariants* in your programs. Invariants are not studied in the book.

2 Arithmetic Expressions

This section studies arithmetic expressions in Java and how such expressions are *evaluated*. Here *evaluating* an expression means computing its value and making sure the expression's possible *side-effects* take place. Here a *side-effect* is a change in state of the overall computation, e.g. a variable's value may change during the computation, the computation may output something, or read in some input. To appreciate the difficulties, try to predict the output of the following computation.

```java
int a = 2 * 3 + 1;
int b = 2 * (3 + 1);
int c = (2 * 3) + 1;
System.out.println( a );
System.out.println( b );
System.out.println( c );
```

As you may have guessed this outputs the numbers 7, 8, and 7 on separate lines. In the remainder of this section we shall study Java's mechanism for evaluating similar as well as more complicated expressions.

2.1 Why Bother

There are two reasons for having evaluation rules. They are related to "common sense" conventions. For example, when you write '`1 + 2 * 3`' you expect '`1 + (2 * 3)`', not '`(1 + 2) * 3`'. Likewise, when you write '`1 - 2 - 3`' you expect '`(1 - 2) - 3`', not '`1 - (2 - 3)`'.

2.2 Operators

For simplicity we shall restrict our computations to arithmetic. Most of the time your programs will use only a few operators:

Assignment: '='

Addition: '+'

Subtraction: '−'

Multiplication: '∗'

Division: '/'

Remainder: '%'

Plus: unary '+'; and

Negation: unary '−'.

Most arithmetic operators are defined for integer and floats. The only operator which is not defined for floats is integer remainder: '%'.

Arguably, integer division and remainder are not properly defined in the presence of one or more negative operands. For example, the sign of the result is the same as the sign of the first operand. (Except when the result is zero and the first operand is not.) Aside from this, remainder works as expected. You get a run-time error if the second operand is zero. Let ⟨`lhs`⟩ be an integer and let ⟨`rhs`⟩ be a non-zero integer, then

- ⟨`lhs`⟩ `/` ⟨`rhs`⟩ gives the integral part of dividing ⟨`lhs`⟩ by ⟨`rhs`⟩. However, the sign of the result is equal to the sign of ⟨`lhs`⟩.
- ⟨`lhs`⟩ `%` ⟨`rhs`⟩ gives the remainder of the division.

In all cases we have the following equality:

$$\langle \texttt{lhs} \rangle = ((\langle \texttt{lhs} \rangle \,/\, \langle \texttt{rhs} \rangle) * \langle \texttt{rhs} \rangle) + (\langle \texttt{lhs} \rangle \,\%\, \langle \texttt{rhs} \rangle) \ .$$

The equality operator in `Java` is written using two equality signs: `==`, so `1 == 1` is `true`, and `1 == 2` is `false`. The following are some examples.

- `4 / 2 == 2`, so `4 % 2 == 0`.
- `3 / 2 == 1`, so `3 % 2 == 1`.
- `2 / 2 == 1`, so `2 % 2 == 0`.
- `1 / 2 == 0`, so `1 % 2 == 1`.

- `0 / 2 == 0`, so `0 % 2 == 0`.
- `7 / 3 == 2`, so `7 % 3 == 1`.
- `19 / 5 == 3`, so `19 % 5 == 4`.

Computing remainders is useful for *modular* (clock) arithmetic. We have

- `0 % 2 == 0;`
- `1 % 2 == 1;`
- `2 % 2 == 0;`
- `3 % 2 == 1;`
- `4 % 2 == 0;`
- ....

Note that the numbers in the last column are of the form `0, 1, 0, 1`, ... It is an infinite sequence of blocks of numbers. Each block is of the form '`0, 1`'.

Assuming that ⟨`rhs`⟩ is greater than 1, we also have

- `0 %` ⟨`rhs`⟩ `== 0;`
- `1 %` ⟨`rhs`⟩ `== 1;`
- ...
- `(`⟨`rhs`⟩ `- 1) %` ⟨`rhs`⟩ `==` ⟨`rhs`⟩ `- 1;`
- ⟨`rhs`⟩ `%` ⟨`rhs`⟩ `== 0;`
- `(`⟨`rhs`⟩ `+ 1) %` ⟨`rhs`⟩ `== 1;`
- ....

The numbers in the last column form an infinite sequence of successive numbers of a "clock" with ⟨`rhs`⟩ numbers on its face. The number at the top is `0`, the next number (in clockwise direction) is `1`, and so on. The last number — it is the number before the `0` — is ⟨`rhs`⟩ `- 1`. If we enumerate the numbers in clockwise direction we get `0, 1`, ..., ⟨`rhs`⟩ `- 1`. If we continue counting in clockwise direction from the last position, we get `0, 1`, ..., ⟨`rhs`⟩ `- 1`, and so on.

2.3 Simple Expressions

Simple expressions are easy to evaluate. For example, to evaluate the following expression

$$\langle \texttt{variable} \rangle_1 \, \langle \texttt{binary arithmetic operator} \rangle \, \langle \texttt{variable} \rangle_2 ,$$

we first take the value of $\langle \texttt{variable} \rangle_1$, then take the value of $\langle \texttt{variable} \rangle_2$, and then apply $\langle \texttt{binary arithmetic operator} \rangle$. Note that first taking the value of $\langle \texttt{variable} \rangle_2$, then taking the value of $\langle \texttt{variable} \rangle_1$, and then applying $\langle \texttt{binary arithmetic operator} \rangle$ gives us the same value. In general this does not hold and the order of evaluation matters:

Assignments: The first reason why the order of computations matters is that sub-computations may carry out assignments:

```Java
int a = 2;
int b = a * (a = 1);
```

Side effects: In general, the order of evaluation makes a difference if sub-computations have side-effects or depend on state. Examples: computations that use the same, shared, random number generator, computations that perform input and output, and so on.

2.4 Associativity

An important notion that determines expression evaluation is *associativity*. In the following, let $\oplus$ be a binary operator, and let v_1, ..., v_n be n values.

left-to-right: If $\oplus$ *associates to the left* then

$$v_1 \oplus v_2 \oplus \cdots \oplus v_n = (((v_1 \oplus v_2) \oplus \cdots) \oplus v_n \, .$$

Here the computation starts at the left and expands to the right. If $\oplus$ is left associative, we also say that it is *left-to-right associative*. All arithmetic operators are left associative. Most remaining operators are also left associative.

```Java
int answer = 840 / 10 / 2; // Assigns 42.
```

right-to-left: Only a few operators are *right associative*. If $\oplus$ *associates to the right* then

$$v_n \oplus \cdots \oplus v_2 \oplus v_1 = v_n \oplus (\cdots \oplus (v_2 \oplus v_1)) \, .$$

Here the computation starts to the right and expands to the left. If $\oplus$ is right associative we also say that it is *right-to-left associative*. An important right associative operator is the assignment operator.

```Java
int result1, result2, result3;
result3 = result2 = result1 = 1;
// result3 = (result2 = (result1 = 1));
```

Regardless of operator associativity, the left operand is always evaluated first. If the left-hand-side operand involves an expression with side-effects, then the order of evaluation may make a difference, even for right associative operators. For example, let's assume you write the following:

```Java
int[] n = new int[ 4 ];
int i = 2;
n[ i = 1 ] = i;
```

Here '`int[] n = new int[ 4 ]`' declares an `int` array `n` of size 4. After these statements `i` and `n[ 1 ]` are equal to `1`. As a more complicated example, consider the following:

```Java
int[] n = new int[ 4 ];
int i = 2;
n[ i = 1 ] = n[ i = i + 2 ] = i;
```

After these statements `i`, `n[ 1 ]`, and `n[ 3 ]` are equal to `3`.

Note that the previous example program is not particularly clear: it takes a long time to figure out what it does. It is much clearer not to rely on the side-effects of the assignments inside the array subscripts and write.

```Java
int[] n = new int[ 4 ];
int i = 2;
i = 1;
i = i + 2;
n[ 1 ] = n[ i ] = i;
```

Even better, write

```Java
int[] n = new int[ 4 ];
int i = 3;
n[ 1 ] = n[ 3 ] = 3;
```

Arguments of methods are also evaluated from left to right. The following contrived example demonstrates this.

```Java
private int add( int first, int second ) {
    return first + second;
}

private void example( ) {
    int number = 0;
    int result = add( number = 1, number + 1 );
    System.out.println( result );
}
```

The method `example` outputs '3'. To see how this works, notice that the methods arguments are evaluated from left to right. The first argument is the expression '`number = 1`', which is an assignment. Since this is the left-most argument, the expression is evaluated. Evaluating the assignment results assigns the value `1` to `number`. Java assignments also result in a value: the assigned value. Therefore, the first argument is `1`. The next argument is the expression '`number + 1`'. Evaluating this expression results in the value 2, so the second argument is 2. Next the method `sum( )` is called with `1` as its first and 2 as its second argument. The method returns 3 and this is what is printed.

The previous example teaches an important lesson:

> 'Reasoning about expressions with sub-assignments and other side-effects is difficult. Avoid side-effects in expressions or else....' — Anonymous Java Lecturer.

2.5 Precedence

In general Java expressions are evaluated from left to right. However, some operators should be applied before others. We say that these operators have a higher *precedence level*. For example, multiplication and division have a higher precedence than addition and subtraction.

```Java
int three = 1 + 1 * 2; // Assigns 3 to three.
```

It is always possible to override operator precedence using parentheses. For example, the following assigns 4 to `four`.

```Java
int three =  1 + 1  * 2; // Assigns 3 to three.
int four  = (1 + 1) * 2; // Assigns 4 to four.
```

Most programmers don't know the exact operator precedence levels. Even if they do, they usually use parentheses for clarity:

```Java
int result = 1 + ((2 * 3) / 4) + 5;
```

3 Decision Making

There are three constructs that affect the flow of control.

- The `if` statement;
- The `if–else` statement; and
- The `switch` statement.

The first two constructs depend on `boolean` expressions. For the moment we shall postpone the discussion of the `switch` statement.

3.1 The `if` Statement

The conditional or `if` statement written as follows:

```Java
if ( condition )
   statement
```

It works as expected: ⟨`statement`⟩ is carried out if and only if (iff) ⟨`condition`⟩ is `true`.

3.2 Comparisons

Usually conditions are made using comparisons, which can be made as follows:

⟨**`fst`**⟩ **`==`** ⟨**`snd`**⟩**:** `true` iff ⟨`fst`⟩ is equal to ⟨`snd`⟩.

⟨**`fst`**⟩ **`!=`** ⟨**`snd`**⟩**:** `true` iff ⟨`fst`⟩ is not equal to ⟨`snd`⟩.

⟨**`fst`**⟩ **`<`** ⟨**`snd`**⟩**:** `true` iff ⟨`fst`⟩ is less than ⟨`snd`⟩.

⟨**`fst`**⟩ **`<=`** ⟨**`snd`**⟩**:** `true` iff ⟨`fst`⟩ is less than or equal to ⟨`snd`⟩.

⟨**`fst`**⟩ **`>`** ⟨**`snd`**⟩**:** `true` iff ⟨`fst`⟩ is greater than ⟨`snd`⟩.

⟨**`fst`**⟩ **`>=`** ⟨**`snd`**⟩**:** `true` iff ⟨`fst`⟩ is greater than or equal to ⟨`snd`⟩.

The following is an example.

```Java
if (temperatureInDegrees < 0) {
    System.out.println( "It's freezing." );
}
```

Note that the braces may be omitted if there's only one statement in the `if` clause.

3.3 Boolean Expressions

The following shows `boolean` operators for negation, conjunction, and disjunction.

`!` ⟨**`expr`**⟩`:` `true` iff ⟨`expr`⟩ is `false`.

⟨**`fst`**⟩ `&&` ⟨**`snd`**⟩`:` `true` iff ⟨`fst`⟩ and ⟨`snd`⟩ are `true`.

⟨**`fst`**⟩ `||` ⟨**`snd`**⟩`:` `true` iff ⟨`fst`⟩ or ⟨`snd`⟩ are `true`.

3.4 The `if-else` Statement

The `if-else` statement is written as follows:

```
if (⟨condition⟩)
  ⟨first statement⟩
else
  ⟨second statement⟩
```
Java

It should not come as a surprise that this carries out ⟨`first statement`⟩ if ⟨`condition`⟩ is ⟨`true`⟩ and carries out ⟨`second statement`⟩ otherwise.

The following is an example.

```
if (temperatureInDegrees < 0) {
    System.out.println( "It's freezing." );
} else {
    System.out.println( "It's not freezing." );
}
```
Java

The following is a common "problem" with starting programmers. Assume that the condition ⟨`condition`⟩ is side-effect free. Next consider the following.

```
if ( condition ) {
    //  condition  is true
     statements
} else if (!  condition ) {
    //  condition  is not true
    ⟨more statements⟩
}
```
Java

In this previous example, the condition for the `else` statement is completely superfluous (it's a tautology: it *has* to be true). Writing the additional condition is very confusing because writing it suggests the condition is needed. It is much clearer to write:

```Java
if ( condition ) {
    // condition is true
    statements
} else {
    // condition is not true
    more statements
}
```

In a similar vein, assume that all conditions are side-effect free. Writing the following is also confusing:

```Java
if ( condition 1) {
    // condition 1 is true.
    statements 1
} else if (! condition 1 && condition 2) {
    // condition 1 is false and
    // condition 2 is true.
    statements 2
} else if (! condition 1 && ! condition 2) {
    // condition 1 is false and
    // condition 2 is false.
    statements 3
}
```

The following is much clearer:

```Java
if ( condition 1) {
    // condition 1 is true.
    statements 1
} else if ( condition 2) {
    // condition 1 is false and
    // condition 2 is true.
    statements 2
} else {
    // condition 1 is false and
    // condition 2 is false.
    statements 3
}
```

3.5 Dangling `else`

When a conditional statement is parsed, Java always tries to match an `else` clause with the nearest preceding `if` clause, so

```Java
if (condition1)
    if (condition2)
        stuff
    else
        more stuff
```

is equivalent to

```Java
if (condition1) {
    if (condition2) {
        stuff
    } else {
        more stuff
    }
}
```

It helps if you ass the braces in your programs.

This equivalence leads to an error which is known in the literature as the *dangling* `else`. The problem manifests itself in programs like the following:

```Java
if (condition1)
    if (condition2)
        stuff
else
    more stuff
```

The program is deliberately indented in the wrong way: the `else` branch actually belongs to the *second* `if`. Because of layout problems like this, it becomes easy to think that the `else` actually belongs to the first `if` — especially if the statements in ⟨`stuff`⟩ take many lines. As suggested in the previous paragraph you can avoid this by adding additional braces.

3.6 The Conditional Operator

The conditional operator is related to the `if-else` statement. The difference is that it returns a value.

```Java
(⟨condition⟩ ? ⟨first⟩ : ⟨second⟩)
```

- Returns ⟨`first`⟩ if ⟨`condition`⟩ is `true`;
- Returns ⟨`second`⟩ if ⟨`condition`⟩ is `false`.

The following is an example.

```Java
String str = ( temperatureInDegrees < 0
             ? "It's freezing!"
             : "It's not freezing!" );
System.out.println( str );
```

4 Iteration

This section studies Java's bounded and unbounded iteration constructs.

4.1 The `for` Statement

The `for` construct is mainly used for *bounded iteration*. Here a *bounded iteration* is an iteration which usually depends on a single *induction variable* or *loop variable* that *counts* the number of iterations and exits the loop when the variable's value reaches and/or exceeds a predefined threshold value. Each iteration usually increments or decrements the induction variable's value by a fixed value — usually 1. Usually, bounded iteration in Java is implemented using the `for` statement. The following is the syntax for the `for` statement.

```
for ( initialisation ; condition ; update )
    statement
```
Java

⟨ ⟩ ⟨ ⟩ ⟨ ⟩

1. ⟨The statement⟩ starts by carrying out ⟨`initialisation`⟩, which should be a single statement or declaration. The purpose of ⟨`initialisation`⟩ is to initialise the variables that "control" the loop. These variables are usually called *induction* variables. Is is also to use an empty statement for ⟨`initialisation`⟩.

2. Next the `for` construct continues by executing a sequence of zero or more "loops". The `boolean` expression ⟨`condition`⟩ is used to determine if the next loop should be carried out. If ⟨`condition`⟩ is true then the next loop is executed. Otherwise, the `for` construct stops. If a loop is executed, then it starts with `statement`.

3. The statement ⟨`update`⟩ is carried out at the end of each loop. Usually, ⟨`update`⟩ adjusts the value of the induction variable by incrementing or decrementing its value.

The following is an example that outputs all binary digits.

```
int digit; // Declare induction variable.
for ( digit = 0; digit <= 1; digit = digit + 1 ) {
    System.out.print( "Next binary digit is " );
    System.out.println( digit );
}
```
Java

The following is an alternative and *better* way to implement the previous example.

```
for ( int digit = 0; digit <= 1; digit = digit + 1 ) {
    System.out.print( "Next binary digit is " );
    System.out.println( digit );
}
```
Java

The main advantage is that this version keeps the induction/loop variable `digit` *local* to the `for` statement. Stated differently, the improved version restricts the visibility of the variable to the `for` construct. By restricting the visibility of `digit` to the loop, this avoids certain kinds of errors. For example, any reference outside the `for` loop to an undeclared variable called `digit` will now result in a compiler error.

4.2 The `while` Statement

This construct is mainly used for *unbounded iteration*. Here a *unbounded iteration* is an iteration with a complicated termination condition. The following is the syntax for the `while` statement.

```
while (  condition  )                                                    Java
    statement
```

⟨ ⟩

This carries out ⟨`statement`⟩ while ⟨`condition`⟩ holds. So each loop consists of a single execution of ⟨`statement`⟩, but the iteration is only carried out if ⟨`condition`⟩ is true. The following is an example.

```
int[] keys = { 1, 4, 6, 8 };                                             Java
int index = 0;
int key = 6;
while ( index != keys.length && keys[ index ] != key ) {
    index = index + 1;
}
if (index != keys.length) {
    System.out.println( ”key is in keys.” );
} else {
    System.out.println( ”key is not in keys.” );
}
```

4.3 The `do-while` Statement

Java also has a `do-while` statement. This almost works like the `while` statement. However, this time the condition is evaluated *after* the loop.

```
do                                                                       Java
    statement
while (  condition  );
```

A *compound statement* is a sequence of statements which are grouped together using braces:

⟨ ⟩

⟨ ⟩

```
{                                                                    Java
    ⟨statement⟩1;
    ⟨statement⟩2;
    ⟨    ⟩
    ⋮
}
```

Even if the ⟨statement⟩ in the do-while statement is not a compound statement (does not have braces), it is clearer if you add braces in the do-while loop:

```
do {                                                                 Java
    statement
} while (  condition  );
    ⟨         ⟩
```

The do-while statement which is listed at the start of this section is equivalent to the following:

```
 statement                                                           Java
while (  condition  )
    statement
⟨         ⟩
```

Most applications can be implemented without using the do-while statement, and this is usually clearer. However, for some applications it may be useful. For example,

```
int number = -1;                                                     Java
// Read number until the number is positive.
do {
    number =  read next number ;
    if (number <= 0) {
        output error: number must be positive ;
    }
} while (number <= 0);
```

5 Increment and Decrement

This section explains pre- and post-increment and decrement operators, which are commonly used in for and while loops to adjust the values of the induction variables in such loops. Unfortunately, there operators are not always well understood.

Java has four operators for incrementing and decrementing variables. These operators do not involve an explicit "assignment" operator (=) but they *do* have side-effects. It is usually "safe" to use these operators in isolation. For obvious reasons — they have side-effects — they should be avoided in non-trivial situations.

The *post-increment* operator is commonly used to increment lvalues in loops. (Remember that an lvalue is something which you can assign a value to.) You use it as follows: '⟨lvalue⟩ ++'. Before

explaining what this does, note that '⟨lvalue⟩ ++' is an *expression*, which returns a value. Effectively it does two things:

- It returns the *initial* value of ⟨lvalue⟩.
- As a side-effect it increments ⟨lvalue⟩.

When used in isolation, this effectively increments ⟨lvalue⟩. Care should be taken in other situations.

The following is a typical example of post-increment.

```
for (int var = START; var < LIMIT; var ++) {
    // START <= var && var < LIMIT.
     stuff
}
```
Java

⟨ ⟩

In this example, the expression var ++ increments the variable var at the end of each iteration. The expression is used in isolation and works as "expected": if ⟨stuff⟩ does not change the value of var then inside the for statement var will have values ranging from START to LIMIT - 1 (including LIMIT - 1). Make sure you understand this.

The following is another typical example of post-increment. However, here the post-increment operator is not used in isolation.

```
for (int var = START; var ++ < LIMIT; ) {
    // START < var && var <= LIMIT.
     stuff
}
```
Java

Just as in the previous example, the expression var ++ increments the variable var. However, the expression var ++ is part of the more complex expression var ++ < LIMIT. To evaluate this more complex expression you first evaluate the first operand, then evaluate the second operand, and then evaluate the values which result from these evaluations. Evaluating the first operand results in the *initial* value of var and as a side-effect it increments var. So the comparison var ++ < LIMIT compares the *initial* value of var and LIMIT. Stated differently, the expression var ++ returns the *current* value of var and *then* increments var. Note that here the post-increment is not used in isolation and works different as before: if ⟨stuff⟩ does not change the value of var then inside the for statement var will have values ranging from START + 1 to LIMIT (including LIMIT). This is different from the previous example. Make sure you understand this.

The post-decrement operator is used as follows: ⟨lvalue⟩ --. It works similar to post-increment but it *decrements* ⟨lvalue⟩.

The pre-increment operator is used as follows: ++ ⟨lvalue⟩. It works similar to post-increment, but it first increments ⟨lvalue⟩, and then returns the (resulting) value of ⟨lvalue⟩. The pre-decrement operator, which is written -- ⟨lvalue⟩, works in a similar way: it first decrements ⟨lvalue⟩ and then returns the resulting value of ⟨lvalue⟩.

The increment and decrement operations should be used with caution. For example, you can write 'var = var ++'. Effectively, this does "nothing". We first evaluate the rhs of the assignment. This gives

us a value, which we assign. To evaluate the rhs we apply the post-increment operator. This returns the initial value of `var`. Next this increments `var`. So the result of evaluating the rhs is the *initial* value of `var`. Assigning it to `var` effectively does "nothing".

6 Partial Precedence Table

The purpose of this section is to present a partial operator precedence table for the operators which are studied in these notes. The table is listed in Table 1. The higer an operator in the table, the higher its precedence. The expression '⟨lvalue⟩' in the table means an expression which can be assigned a value.

Description	Operator	Associativity
post-*crement	⟨`lvalue`⟩`++` and ⟨`lvalue`⟩`--`	left
pre-*crement, unary	`++`⟨`lvalue`⟩, `--`⟨`lvalue`⟩, `+`⟨`expr`⟩, `-`⟨`expr`⟩, and `!`⟨`expr`⟩	right
object creation	`new`	right
multiplicative	`*`, `/`, and `%`	left
additive	`+` and `-`	right
relational	`<`, `>`, `<=`, and `>=`	left
equality	`==` and `!=`	left
logical and	`&&`	left
logical or	`||`	left
ternary	⟨`condition`⟩ `?` ⟨`expr`⟩ `:` ⟨`expr`⟩	right
assignment	`=`, `+=`, `-=`, `*=`, `/=`, and `%=`	right

Table 1: Partial operator precedence table.

The '`l`' and the '`value`' in '`lvalue`' probably come from the fact that an ⟨lvalue⟩ may be regarded as a "*value*" which may occur at the *l*eft hand side of an assignment.

7 Invariants

This section is not in the book. The following `Java` code is supposed to all all integers from `0` to `100`.

```Java
int i, sum;

i = 0;
sum = 0;
while (i < 100) {
  i = i + 1;
  sum = sum + i;
} // sum == 1 + 2 + ... + 100
```

In the remainder of this section we shall prove the previous code actually does add all non-negative integers that are less than or equal to 100. To do this, we shall need the notion of an *invariant*.

An *invariant* is a meaningful "comment" that relates the current values of the variables in the program. Invariants are useful for several reasons.

Concretize: They make the relationships explicit. This helps when writing the program.

Correctness: They may help you prove the program is correct.

Maintenance: By putting in the invariants, you remind yourself and colleagues about the purpose of the variables and how they control program execution. This makes it easier to understand the code and easier to modify the code, thereby improving program maintainability.

Good programmers add comments to their programs which state the invariants.

The following shows some useless comments which your lecturer has seen over the past years. These comments are useless because any Java programmer should know they are true. Adding such comments obfuscates the rest of the program.

Don't Try this at Home

```
// variable declaration.
int x;

// assign zero to x.
x = 0;

// add two to x.
x = x + 2;

// increment x.
x ++;
```

The following is a useful relationship of the `if` statement. Regardless of the actual value of ⟨`condition`⟩, we know that ⟨`condition`⟩ should be `false` at the start of the `else` clause. Here it is assumed that ⟨`condition`⟩ is side-effect free.

Java

```
if (  condition  ) {
    //  condition
    ⋮
}
// !  condition
```

⟨ ⟩

Likewise, the following also holds.

⟨ ⟩

```
if ( ⟨condition⟩ ) {
    // ⟨condition⟩
    ⋮
} else {
    // ! ⟨condition⟩
    ⋮
}
```
Java

The following shows that invariants also help when you're using the `while` statement. The idea is that the termination condition of the `while` statement together with a proper condition before and at the end of body of the `while` statement let you derive a condition which should hold immediately after the `while` statement. Again it is assumed that all conditions are side-effect free. In addition it is assumed that there is no explicit `break` from the `while` loop. The following demonstrates the idea: the invariant at the end follows from the previous invariants and the termination conditions.

```
// ⟨condition⟩1
while ( ⟨condition⟩2 ) {
    ⋮
    // ⟨condition⟩1
}
    // ⟨condition⟩1 && ! ⟨condition⟩2
```
Java

- It is easy to see why $\langle$`condition`$\rangle_2$ should be `false` after the `while` statement: the condition of the `while` loop failed.
- To see why $\langle$`condition`$\rangle_1$ should also hold immediately after the `while` statement, notice that $\langle$`condition`$\rangle_1$ was true just before the condition of the `while` loop was evaluated. Regardless of how many iterations there are, it *has* to be true (since both conditions are side-effect free).

This relationship is frequently used to prove things about the `while` statement.

```
int i, sum;

i = 0;
sum = 0;           // i <= 100 && sum == 0 + 1 + … + i
while (i < 100) { // i <  100 && sum == 0 + 1 + … + i
  i = i + 1;       // i <= 100 && sum == 0 + 1 + … + i-1
  sum = sum + i;  // i <= 100 && sum == 0 + 1 + … + i
}     // i >= 100 && i <= 100 && sum == 0 + 1 + … + i
                   // i == 100 && sum == 0 + 1 + … + i
                                  // sum == 0 + 1 + … + 100
```
Java

Good programmers are so used to using these invariants that they automatically put in the two key invariants:

- The condition `i < 100 && sum == 0 + 1 + … + i` at the start of the body of the `while` loop.
- The condition `sum == 0 + 1 + … + 100` after the `while` loop.

The remaining invariants usually follow from these two.

8 Puzzlers

This section presents two problems which should help you understand division and remainder and appreciate some pitfalls when implementing loops. Both puzzles are based on [Bloch and Gafter, 2005].

8.1 Test for Oddness

Before you start, think about how you'd implement a method `isOdd( )` that takes an `int` and returns `true` if and only if its argument is odd. If you don't know how to write methods then think about how you'd detect whether a given `int` is odd.

Now that you've thought about the implementation, you're ready for the first problem [Bloch and Gafter, 2005, Puzzle 1]. The problem is as follows: what is wrong with the implementation? *Hint: the method returns the wrong result in about 1 out of every 4 cases.*

Don't Try this at Home

```
public static boolean isOdd( int number ) {
    return number % 2 == 1;
}
```

The problem with the previous implementation is that it returns the wrong result if `number` is negative and odd. If you remember the rules for integer division and remainder the sign of the result for division and remainder is the same as that of the first operand, provided the result is non-zero. For example, `-1 / 2 == 0` and `-1 % 2 == -1`, `-2 / 2 == -1` and `-2 % 2 == 0`, `-3 / 2 == -1` and `-3 % 2 == -1`, `-4 / 2 == -2` and `-4 % 2 == 0`, and so on. The obvious solution is the following.

Java

```
public static boolean isOdd( int number ) {
    return number % 2 != 0;
}
```

8.2 Loop de Loop

The next puzzle is based on [Bloch and Gafter, 2005, Puzzle 26]. The following program counts the iterations in the `for` loop. What does it print? (Remember that `Integer.MAX_VALUE` is the largest possible `int` value.)

Don't Try this at Home

```
public class LoopDeLoop {
    public static void main( String[] args ) {
        final int LAST_INDEX = Integer.MAX_VALUE;
        final int FIRST_INDEX = Integer.MAX_VALUE - 10;
        int count = 0;
        for (int index = FIRST_INDEX; index <= LAST_INDEX; index++) {
            count ++;
        }
        System.out.println( count );
    }
}
```

The correct answer is that the program doesn't print anything: it starts an infinite loop. The key to understanding this is that the test '`index <= LAST_INDEX`' is always true, regardless of the value of `index`. Add to this that `Integer.MAX_VALUE + 1 == Integer.MIN_VALUE` (because of overflow) *et voila*.

9 Acknowledgements

The partial operator precedence table is based on `http://download.oracle.com/javase/tutorial/java/nutsandbolts/operators.html`. The puzzlers are based on [Bloch and Gafter, 2005, Puzzles 1 and 26].

References

[Bloch and Gafter, 2005] Joshua Bloch and Neal Gafter. *Java Puzzlers* Traps, Pitfalls, and Corner Cases. Addison-Wesley, 2005.