

# 19

## *Variables Types and Expressions*

## Self-review Questions

**19.1** Which of the following are illegal variable names and why?

*aName, 2D, WIDTH, Position, HelgHt, this, x&y,  
whiletrue, name\$, current\_position, Float*

The name 2D is not allowed as it starts with a numeric digit. x&y is not allowed as it contains a symbol not permitted in variable names. Float is not allowed since it is the name of a built-in type.

**19.2** What is seen if the following strings are displayed on the computer screen?

```
"Hello\nWorld\n!"
"\t\tHello"
"\\\\\\nHello"
"1.23e+9f"
"Hello\b\bWorld"
"\Hello\"
```

**19.3** Are any of the following expressions using ++ and – legal? If not, why not?

```
int j = 5 ;
j++ ++ ;
j(++);
(j++)++;
j+-- ;
(j)++ ;
```

j++ ++ is not allowed since the result of j++ is a value not a variable and the ++ operator can only be applied to a variable. (j++)++ is not allowed for exactly the same reason. Similarly j+--, -- must be applied to a variable not a value.

(j)++ is allowed since the result of the expression (j) is the variable j.

**19.4** What value is assigned to n in each of the following?

```
int n = 0 ;
int x = 1 ;
n = x++ + x++ ;
n = n++ - x++ ;
n = x-- + -x++ ;
```

The value assigned for the three assignment statements are:

1. 3
2. 0
3. 1

which can be checked by executing a small program:

```
public class S_19_4 {
    private void run () {
        int n = 0 ;
        int x = 1 ;
        n = x++ + x++ ;
        System.out.println ( n ) ;
        n = n++ - x++ ;
        System.out.println ( n ) ;
        n = x-- + -x++ ;
        System.out.println ( n ) ;
    }
    public static void main ( final String[] args ) {
        ( new S_19_4 ( ) ).run ( ) ;
    }
}
```

**19.5** Why can't the largest integer be negated in Java?

**19.6** What does the expression:

```
i < 0 ? 0 : i > 10 ? 1 : i > 20 ? 2 : 3
```

evaluate to for values of *i* between -5 and 50?

For negative values of *i* the value is 0.

For values of *i* such that  $0 \leq i \leq 10$ , the value is 3.

For all values of *i* > 10 the value is 1.

This result can be verified by writing, compiling and executing a small program:

```
public class S_19_6 {
    private void run () {
        for ( int i = -5 ; i < 51 ; ++i ) {
            int n = i < 0 ? 0 : i > 10 ? 1 : i > 20 ? 2 : 3 ;
            System.out.println ( i + " : " + n ) ;
        }
    }
    public static void main ( final String[] args ) {
```

```
        ( new S_19_6 ( ) ).run ( ) ;  
    }  
}
```

When executed this gives:

```
-5: 0  
-4: 0  
-3: 0  
-2: 0  
-1: 0  
0: 3  
1: 3  
2: 3  
3: 3  
4: 3  
5: 3  
6: 3  
7: 3  
8: 3  
9: 3  
10: 3  
11: 1  
12: 1  
13: 1  
14: 1  
15: 1  
16: 1  
17: 1  
18: 1  
19: 1  
20: 1  
21: 1  
22: 1  
23: 1  
24: 1  
25: 1  
26: 1  
27: 1  
28: 1  
29: 1  
30: 1  
31: 1  
32: 1  
33: 1  
34: 1  
35: 1  
36: 1  
37: 1  
38: 1  
39: 1  
40: 1  
41: 1  
42: 1  
43: 1  
44: 1  
45: 1  
46: 1  
47: 1  
48: 1  
49: 1  
50: 1
```

## Programming Exercises

**19.1** Write a program to use each kind of cast between primitive types. Which casts lose information and what is lost?

```
public class E_19_1 {
    private byte b = (byte) 5 ;
    private short s = (short) 5 ;
    private int i = 5 ;
    private long l = 5 ;
    private float f = 5.0f ;
    private double d = 5.0 ;
    private void checkByte () {
        byte v ;
        v = (byte) s ;
        v = (byte) i ;
        v = (byte) l ;
        v = (byte) f ;
        v = (byte) d ;
    }
    private void checkShort () {
        short v ;
        v = (short) b ;
        v = (short) i ;
        v = (short) l ;
        v = (short) f ;
        v = (short) d ;
    }
    private void checkInt () {
        int v ;
        v = (int) b ;
        v = (int) s ;
        v = (int) l ;
        v = (int) f ;
        v = (int) d ;
    }
    private void checkLong () {
        long v ;
        v = (long) b ;
        v = (long) s ;
        v = (long) i ;
        v = (long) f ;
        v = (long) d ;
    }
    private void checkFloat () {
        float v ;
        v = (float) b ;
        v = (float) s ;
        v = (float) i ;
        v = (float) l ;
        v = (float) d ;
    }
    private void checkDouble () {
```

```

    double v ;
    v = (double) b ;
    v = (double) s ;
    v = (double) i ;
    v = (double) l ;
    v = (double) f ;
}
private void run () {
    checkByte () ;
    checkShort () ;
    checkInt () ;
    checkLong () ;
    checkFloat () ;
    checkDouble () ;
}
public static void main ( final String[] args ) {
    ( new E_19_1 () ).run () ;
}
}

```

**19.2** Write a program to investigate the effects of overflow. What happens to the result when an overflow occurs when multiplying two integers or doubles?

**19.3** Write a program to illustrate how each primitive type is represented as a string. What happens to the value null?

The program:

```

public class E_19_3 {
    private byte b = (byte) 5 ;
    private short s = (short) 5 ;
    private int i = 5 ;
    private long l = 5 ;
    private float f = 5.0f ;
    private double d = 5.0 ;
    private void run () {
        System.out.println ( b ) ;
        System.out.println ( s ) ;
        System.out.println ( i ) ;
        System.out.println ( l ) ;
        System.out.println ( f ) ;
        System.out.println ( d ) ;
        System.out.println ( (String) null ) ;
    }
    public static void main ( final String[] args ) {
        ( new E_19_3 () ).run () ;
    }
}

```

results in the output:

```

5
5
5
5
5.0
5.0
null

```

The reason for explicitly casting the **null** literal is because without it there is ambiguity: There are two overloads of the `System.println` method that could apply, `String` or `char[]`, so the cast is needed to disambiguate.

**19.4** Write a program that demonstrates the effects of applying the shift operators.

**19.5** Write a program that inputs an integer value and uses the bitwise operators to print out the binary representation of the integer.

The algorithm we use here is 'peel off' the least significant bit of a number and append a character ('0' or '1') to a `StringBuilder` depending on whether the bit was 0 or 1 respectively. We loop for 32 bits (the size of an `int`) or until the value is zero, i.e. all bits left are zero and so don't need printing. This results in a string representation which is in the reverse order to the way we need to print it, so we reverse it prior to printing:

```

public class E_19_5 {
    private void run () {
        final Input input = new Input ();
        System.out.print ( "Enter and integer: " );
        int number = input.nextInt ();
        StringBuilder sb = new StringBuilder ();
        for ( int i = 0 ; i < 32 ; ++i ) {
            if ( ( number & 1 ) == 1 ) { sb.append ( '1' ); }
            else { sb.append ( '0' ); }
            number >>= 1 ;
            if ( number == 0 ) { break ; }
        }
        System.out.println ( sb.reverse () );
    }
    public static void main ( final String[] args ) {
        ( new E_19_5 () ).run ();
    }
}

```