05 Augmented Assignments and Type Casts

Instructor: Ke Wei（柯韋）

➽ A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP112/18/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

September 10, 2018

Outline

1. Floating-Point Numbers
2. Number Literals
3. Arithmetic Expressions
4. Numeric Type Conversion and Type Casting
5. Reading Homework

Floating-Point Numbers

- Just like writing very large or very small numbers on paper with limited space, we divide a number into two parts: *significant digits* and *scale factors*, such as

$$1.1023\times 10^{120} \qquad -7.3000\times 10^{-302}.$$

- We use a fixed number of bits to represent real numbers of very large range with a fixed precision.

IEEE 754 32-bit single precision (`float`) $\pm 1.M\times 2^{E'-127}$

S	E'	M
sign bit	8-bit excess-127 exponent	23-bit mantissa fraction

IEEE 754 64-bit double precision (`double`) $\pm 1.M\times 2^{E'-1023}$

S	E'	M
sign bit	11-bit excess-1023 exponent	52-bit mantissa fraction

Floating-Point numbers Are Not Accurate

- Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy.
- For example,

```
System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);
```

  displays 0.5000000000000001, not 0.5, and

```
System.out.println(1.0 - 0.9);
```

  displays 0.09999999999999998, not 0.1.
- Integers are stored precisely. Therefore, calculations with integers yield precise integer results.

Exponent Operation

There is no exponent operator in Java language. However, we can call the $Math.pow(x,\ y)$ method provided by the built-in *Math* class to compute x^y on two floating-point numbers.

```
System.out.println(Math.pow(2, 3));       // displays 8.0
System.out.println(Math.pow(4, 0.5));     // displays 2.0
System.out.println(Math.pow(2.5, 2));     // displays 6.25
System.out.println(Math.pow(2.5, -2));    // displays 0.16

double x = 4.0, y = 8.0;
System.out.println(Math.pow(x, y));       // displays 65536.0
```

Integer Literals

- A literal is a constant value that appears directly in the program. For example, 34, 1000000 and 5.0 are literals in the following statements:

  ```
  int i = 34;   long x = 1000000;   double d = 5.0;
  ```

- An integer literal can be assigned to an integer variable as long as it can fit into the variable. A compilation error would occur if the literal were too large for the variable to hold. For example, the statement

  ```
  byte b = 1000;
  ```

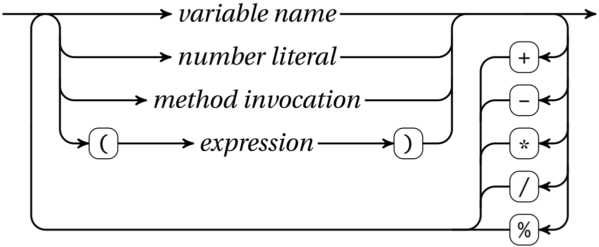
  would cause a compilation error, because 1000 cannot be stored in a variable of the byte type.
- An integer literal is assumed to be of the `int` type, whose value is between -2^{31} (-2147483648) to $2^{31}-1$ (2147483647). To denote an integer literal of the long type, append it with the letter `L` or `l`. `L` is preferred because `l` (lowercase `L`) can easily be confused with 1 (the digit one).

Floating-Point Literals

- Floating-point literals are written with a decimal point.
- By default, a floating-point literal is treated as a `double` type value. For example, 5.0 is considered a `double` value, not a `float` value.
- You can make a number a `float` by appending the letter `f` or `F`, and make a number a double by appending the letter `d` or `D`. For example, you can use `100.2f` or `100.2F` for a `float` number, and `100.2d` or `100.2D` for a `double` number.
- **Scientific notation**. Floating-point literals can also be specified in scientific notation, for example, `1.23456e+2`, same as `1.23456e2`, is equivalent to 123.456, and `1.23456e-2` is equivalent to 0.0123456. `E` (or `e`) represents an exponent and it can be either in lowercase or uppercase.

Java Arithmetic Expressions

A Java arithmetic expression can be a variable name, a number literal, a method call (invocation), or sub-expressions connected by arithmetic operators and grouped by parentheses.



variable name

number literal

method invocation

(*expression*)

+

−

*

/

%

For example, $\frac{3+4x}{5} - \frac{10(y-5)\sqrt{a+b+c}}{x} + 9\left(\frac{4}{x}+\frac{9+x}{y}\right)$ is translated to Java expression:

```
(3+4*x)/5 - 10*(y-5)*Math.sqrt(a+b+c)/x + 9*(4/x+(9+x)/y)
```

How to Evaluate an Expression

- Though Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression are the same.
- Therefore, you can safely apply the arithmetic rule for evaluating a Java expression.
- Particularly, method invocations are evaluated first.

$$
\begin{aligned}
& 3+4\ast\boxed{Math.pow(2,2)}+5\ast(4+3)-1 \\
\Rightarrow & 3+4\ast4+5\ast\boxed{(4+3)}-1 \\
\Rightarrow & 3+\boxed{4\ast4}+5\ast7-1 \\
\Rightarrow & 3+16+\boxed{5\ast7}-1 \\
\Rightarrow & \boxed{3+16}+35-1 \Rightarrow \boxed{19+35}-1 \Rightarrow \boxed{54-1} \Rightarrow 53.
\end{aligned}
$$

Problem: Converting Temperatures

✍ Write a program *FahrenheightToCelsius* that converts a Fahrenheit degree to Celsius using the formula:

$$celsius = \frac{5}{9}(fahrenheit - 32).$$

Note: you must write (why?)

```
    celsius = (5.0 / 9) * (fahrenheit - 32)
```

Augmented Assignment Operators and Self-Increment/Decrement

- Augmented assignment operators.

Operator	Example	Equivalent to
+=	*i* += 8	*i* = *i* + 8
-=	*f* -= 8.0	*f* = *f* - 8.0

Operator	Example	Equivalent to
*=	*i* *= 8	*i* = *i* * 8
/=	*i* /= 8	*i* = *i* / 8
%=	*i* %= 8	*i* = *i* % 8

- Increment and Decrement Operators.

Operator	Name	Description
++*var*	preincrement	The expression (++*var*) increments *var* by 1 and evaluates to the new value in *var* after the increment.
var++	postincrement	The expression (*var*++) evaluates to the original value in *var* and increments *var* by 1.
--*var*	predecrement	The expression (--*var*) decrements *var* by 1 and evaluates to the new value in *var* after the decrement.
var--	postdecrement	The expression (*var*--) evaluates to the original value in *var* and decrements *var* by 1.

Increment and Decrement Operators Explained

- Given the variable declaration below:

```
int i = 10;
```

`int newNum = 10 * i++;` has the same effect as `int newNum = 10*i; i = i+1;`

`int newNum = 10 * (++i);` has the same effect as `i = i+1; int newNum = 10*i;`

- Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read.
- Avoid using these operators in expressions that modify a variable *multiple* times, or use a variable multiple times while modifying it, such as:

```
k = ++i + i;
i = ++i + i--;
```

STOP

Numeric Type Conversion

- Consider the following statements:

```
byte i = 100;
long k = i * 3 + 4;
double d = i * 3.1 + k / 2;
```

- **Conversion Rules**. When performing a binary operation involving two operands of different types, Java automatically converts the operand based on the following rules:
  1. If one of the operands is `double`, the other is converted into `double`.
  2. Otherwise, if one of the operands is `float`, the other is converted into `float`.
  3. Otherwise, if one of the operands is `long`, the other is converted into `long`.
  4. Otherwise, both operands are converted into `int`.

Type Casting

- Implicit casting:

```
double d = 3;    // type widening
```

- Explicit casting:

```
int i = (int)3.0;    // type narrowing
int i = (int)3.9;    // fraction part is truncated
```

✍ What is wrong? `int x = 5 / 2.0;`

- The range of a data type increases in the following order:

  `byte, short, int, long, float, double.`

- **Casting in an augmented assignment**. In Java, an augmented assignment of the form $x \oplus = y$ is implemented as $x = (T)(x \oplus y)$, where T is the type for x, and $\oplus$ stands for an arithmetic operator. Therefore, the following code is correct:

```
int sum = 0;
sum += 4.5; // sum becomes 4 after this statement.
```

equivalent to:

```
sum = (int)(sum + 4.5)
```

Reading Homework

Textbook

- Section 2.10 – 2.15.
- Check Point 2.12 – 2.31.

Internet

- Floating point (`http://en.wikipedia.org/wiki/Floating_point`).

Self-test

- 2.31 – 2.55 (`http://tiger.armstrong.edu/selftest/selftest9e?chapter=2`).