# COMP-248
## Object Oriented Programming I

## Lecture 05:
## Arithmetic and Assignment Operators

**Emad Shihab, PhD**

Parts of the slides are taken from Prof. L. Kosseim

# In this chapter, we will see…

1. Comments
2. Identifiers
3. Indentation
4. Primitive Types
5. Variables
6. Output & Input
7. Assignment
8. **(more) Arithmetic Expressions**
9. (now) More Assignment Operators
10. (now) Assignment Compatibility
11. Strings

## 8- Arithmetic Expressions

- An *expression* is a combination of one or more operands and their operators
- *Arithmetic* operators:

| | |
|---|---|
| Addition | + |
| Subtraction | - |
| Multiplication | * |
| Division | / |
| Remainder | % |

# Operator Precedence (p 24)

- Operators can be combined into complex expressions

```
result  =  total + count / max - offset;
```

- precedence determines the order of evaluation
  - 1st: expressions in parenthesis
  - 2nd: unary + and -
  - 3rd: multiplication, division, and remainder
  - 4th: addition, subtraction, and string concatenation
  - 5th: assignment operator

## Operator Associativity

- Unary operators of equal precedence are grouped right-to-left

  `+-+rate` is evaluated as `+(-(+rate))`

- Binary operators of equal precedence are grouped left-to-right

  `base + rate + hours` is evaluated as

  `(base + rate) + hours`

- **Exception**: A string of assignment operators is grouped right-to-left

  `n1 = n2 = n3;` is evaluated as `n1 = (n2 = n3);`

## Example

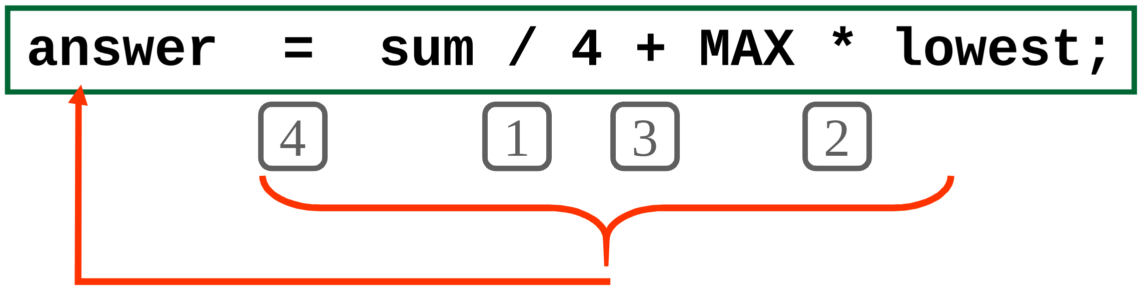
- What is the order of evaluation in the following expressions?

```
a + b + c + d + e

a + b * c - d / e              (a + (b * c)) – (d / e)

a / (b + c) - d % e            (a / (b + c)) – (d % e)

a / (b * (c + (d - e)))        a / (b * (c + (d - e)))
```

## Assignment Revisited

- The assignment operator has a lower precedence than the arithmetic operators

First the expression on the RHS is evaluated



```
answer  =  sum / 4 + MAX * lowest;
```

Then the result is stored in the variable on the LHS

# Just checking...

- What is stored in the integer variable **num1** after this statement?

```
    num1 = 2 + 3 * 5 - 5 * 2 / 5 + 10 ;
```

A. 0
B. 18
C. 25
D. 10

## In this chapter, we will see…

1. Comments
2. Identifiers
3. Indentation
4. Primitive Types
5. Variables
6. Output & Input
7. Assignment
8. (more) Arithmetic Expressions
9. **(now) More Assignment Operators**
10. (now) Assignment Compatibility
11. Strings

## 9- More assignment operators

- in addition to =, often we perform an operation on a variable, and then store the result back into that variable

- Java has shortcut assignment operators:

```
variable = variable operator expression;
variable operator= expression;
```

| <u>Operator</u> | <u>Example</u> | <u>Equivalent To</u> |
|---|---|---|
| `+=` | `x += y` | `x = x + y` |
| `-=` | `x -= y` | `x = x - y` |
| `*=` | `x *= y` | `x = x * y` |
| `/=` | `x /= y` | `x = x / y` |
| `%=` | `x %= y` | `x = x % y` |

## Shorthand Assignment Statements

| Example: | Equivalent To: |
|---|---|
| | |
| `sum -= discount;` | `sum = sum – discount;` |
| | |
| `time /= rushFactor;` | `time = time / rushFactor;` |
| | |
| `amount *= count1 + count2;` | `amount = amount * (count1 + count2);` |

## Assignment operators

- The behavior of some assignment operators depends on the types of the operands
- ex: the +=
  - If the operands are strings, += performs string concatenation
  - The behavior of += is consistent with the behavior of the "regular" +

## Example

```
int amount = 10;
amount += 5;
System.out.println(amount);

double temp = 10.0;
temp *= 10;
System.out.println(temp);

String word = "hello ";
word += "bye";
System.out.println(word);
word *= "bye";   // ???
```

```
15
100.0
hellobye
```

Output

## Increment and Decrement

- In Java, we often add-one or subtract-one to a variable…
- 2 shortcut operators:
  - The *increment operator* (++) adds one to its operand
  - The *decrement operator* (- -) subtracts one from its operand
- The statement: **`count++;`**

  is functionally equivalent to: **`count = count+1;`**
- The statement: **`count--;`**

  is functionally equivalent to: **`count = count-1;`**

## Increment and Decrement

- The increment and decrement operators can be used in expressions in two forms:

1. **in prefix form: `++count;`**
   1. the variable is incremented/decremented by 1
   2. the value of the entire expression is the **new** value of the variable (**after** the incrementation/decrementation)

2. **in postfix form: `count++;`**
   1. the variable is incremented/decremented by 1
   2. the value of the entire expression is the **old** value of the variable (**before** the incrementation/decrementation)

## Example

```
int nb = 50;
++nb;
```

51
value of nb

```
int nb = 50;
nb++;
```

51
value of nb

```
int nb = 50;
int x;
x = ++nb;
```

nb =51
x = 51
value of nb & x

```
int nb = 50;
int x;
x = nb++;
```

nb =51
x = 50
value of nb & x

```
int nb = 50;
int x;
x = nb++ + 10;
```

nb =51
x = 60
value of nb & x

## Just checking...

What is stored in the integer variables **num1, num2** and **num3** after the following statements?

```
num1 = 1;
num2 = 0;
num3 = 2 * num1++  + --num2 * 5;
```

A. `num1 = 1, num2 = 0, num3 = 2`
B. `num1 = 1, num2 = 0, num3 = -1`
C. `num1 = 2, num2 = -1, num3 = 2`
D. `num1 = 2, num2 = -1, num3 = -1`
E. `num1 = 2, num2 = -1, num3 = -3`

# Summary of ++ and --

| Expression | Operation | Value Used in Expression |
|---|---|---|
| `count++` | add 1 | old value |
| `++count` | add 1 | new value |
| `count--` | subtract 1 | old value |
| `--count` | subtract 1 | new value |

## In this chapter, we will see…

1. Comments
2. Identifiers
3. Indentation
4. Primitive Types
5. Variables
6. Output & Input
7. Assignment
8. (more) Arithmetic Expressions
9. (now) More Assignment Operators
10. **(now) Assignment Compatibility**
11. Strings

## 10 - Assignment Compatibility

- In general, the value of one type cannot be stored in a variable of another type

```
int intVariable = 2.99; //Illegal
```

- However, there are exceptions to this

```
double doubleVariable = 2;
```

  - For example, an `int` value can be stored in a `double` type

# Assignment Compatibility

- an expression has a value and a type

```
2 / 4   (value = 0, type = int)
2 / 4.0 (value = 0.5, type = double)
```

- the type of the expression depends on the type of its operands
- In Java, type conversions can occur in 3 ways:
  - arithmetic promotion
  - assignment conversion
  - casting

# Arithmetic promotion

- happens automatically, if the operands of an expression are of different types

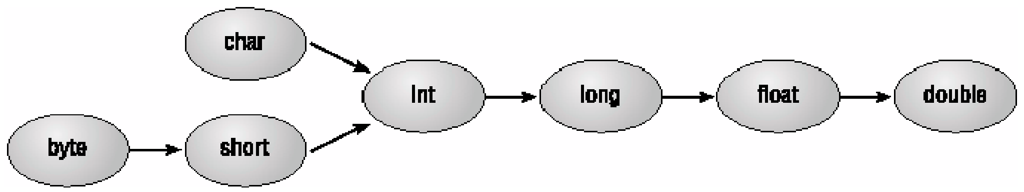
```
aLong + anInt * aDouble
```

- operands are promoted so that they have the same type

- promotion rules:
  - if 1 operand is of type… the others are promoted to…

| if 1 operand is of type… | the others are promoted to… |
|---|---|
| **`double`** | **`double`** |
| **`float`** | **`float`** |
| **`long`** | **`long`** |

  - **`short`**, **`byte`** and **`char`** are always converted to **`int`**



## Examples

- value and type of these expressions?

```
2 / 4
int / int      2/4
int            0
```

```
2 / 4 * 1.0
```

int/int * double	0*1.0
double	0.0

```
1.0 * 2 / 4
```

double * int/int	2.0/4
double	0.5

## Assignment conversions

- occurs when an expression of one type is assigned to a variable of another type

```
var = expression;
```

- *widening* conversion
  - if the variable has a wider type than the expression
  - then, the expression is widened automatically

```
long aVar;
aVar = 5+5;
```

```
byte aByte;
int anInt;
anInt = aByte;
```

```
double aDouble;
int anInt = 10;
aDouble = anInt;
```

  - int & floating point types are compatible
  - **`boolean`** are not compatible with any type

# Assignment conversions

- *narrowing* conversion
  - if the variable has a smaller type than the expression
  - then, compilation error, because possible loss of information

```
int aVar;
aVar = 3.7; ok?
```

```
int aVar;
aVar = 10/4; ok?
```

```
int aVar;
aVar = 10.0/4; ok?
```

## Casting

- the programmer can explicitly force a type conversion
- syntax: **`(desired_type)  expression_to_convert`**

```
int aVar;
aVar = (int)3.7;
(aVar is 3… not 4!)
```

```
byte aByte;
int anInt = 75;
aByte = anInt; // ok?
aByte = (byte)anInt;  // ok?
```

```
double d;
d = 2/4;  // d is 0
d = (double)2/4;  // d is 0.5  (2.0 / 4)
d = (double)(2/4); // d is 0.0
```

*Casting* can be dangerous! you better know what you're doing…

```
byte aByte;
int anInt = 75000;
aByte = (byte)anInt;  // ok
System.out.print(aByte);  // -8!
```

## Question

# In this chapter, we have seen…

- ✓ Comments
- ✓ Identifiers
- ✓ Indentation
- ✓ Primitive Types
- ✓ Variables
- ✓ Output
- ✓ Assignment
- ✓ Arithmetic Expressions
- ✓ More Assignment Operators
- ✓ Assignment Compatibility
- ✓ Strings