Lecture 2 Operators, Expressions

Named Constants

```
final datatype CONSTANTNAME = VALUE;

final double PI = 3.14159;
final int SIZE = 3;
```

Naming Conventions

- Choose meaningful and descriptive names.
- Variables and method names:
  - Use **lowercase**. If the name consists of several words, concatenate all in one, use lowercase for the first word, and capitalize the first letter of each subsequent word in the name. For example, the variables `radius` and `area`.

Naming Conventions, cont.

- Class names:
  - Capitalize the first letter of each word in the name. For example, the class name `ComputeArea`.
- Constants:
  - Capitalize all letters in constants, and use underscores to connect words. For example, the constant PI and MAX_VALUE

Numeric Operators

Name	Meaning	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 – 0.1	33.9
*	Multiplication	300 * 30	9000
/	Division	1.0 / 2.0	0.5
%	Remainder	20 % 3	2

PEMDAS

What is it? Precedence order in arithmetic expressions:

Parentheses
Exponentiation
Multiplication, Division
Addition, Subtraction

2 + 3 * 5 = ?

(2 + 3) * 5 = ?

Operators at the same level (*, /) and (+,-) go from left to right (left associative (+,-) and left associative (*,/))

4 - 3 + 5 = ?

12 / 3 * 6 = ?

Integer Division

+, -, *, /, and %

5 / 2 yields an integer 2.

5.0 / 2 yields a double value 2.5

5 % 2 yields 1 (the remainder of the (integer) division)

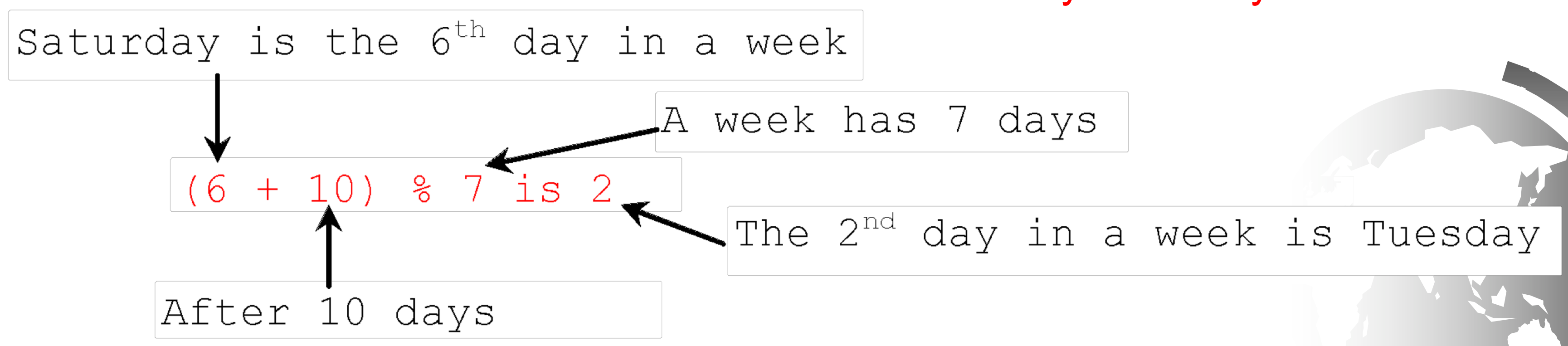
Modulo/Remainder Operator

Remainder is very useful in programming.

An even number % 2 is 0 and an odd number % 2 is 1.

Suppose today is Saturday and you and your friends are going to meet in 10 days. What day is in 10 days? Tuesday; use the following expression:

Which day is Sunday?



Saturday is the 6th day in a week

A week has 7 days

(6 + 10) % 7 is 2

The 2nd day in a week is Tuesday

After 10 days

Relationship of / and %

- For all integers p and q : **p = (p/q)*q + p%q**

- Terminology:
    - p: dividend
    - q: divisor
    - p/q: quotient
    - p%q: remainder

The remainder is negative only if the dividend is negative.

Integer division rounds towards 0: 3/2 = 1, -3/2 = -1

Exponent Operations

```
// these are (Math)library methods

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
```

Number Literals

A ***literal*** is a constant value that appears directly in the program. For example, 34, 1,000,000, and 5.0 are literals in the following statements:

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

Integer Literals

An **integer literal** can be assigned to an integer variable as long as it can fit into the variable.

```
byte b1 = 100;
byte b2 = 1000; // compiler error, WHY?
```

An integer literal is assumed to be of the **int** type, whose value is between -2^{31} (-2147483648) to $2^{31}–1$ (2147483647).

Floating-Point Literals

Floating-point literals are written with a decimal point. By default, a **decimal point literal** is treated as a double type value.

```
double d0 = 0.5;
double d1 = 100.2d;
double d2 = 100.1D;
float f1 = 100.2f;
float f2 = 100.3F;
```

double vs. float

The double type values are more accurate than the float type values. For example,

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
```

displays `1.0 / 3.0 is 0.`$\underbrace{\texttt{3333333333333333}}_{\text{16 digits}}$

```
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
```

displays `1.0F / 3.0F is 0.`$\underbrace{\texttt{33333334}}_{\text{7 digits}}$

Arithmetic Expressions

$$\boxed{\frac{3+4x}{5}-\frac{10(y-5)(a+b+c)}{x}+9(\frac{4}{x}+\frac{9+x}{y})}$$

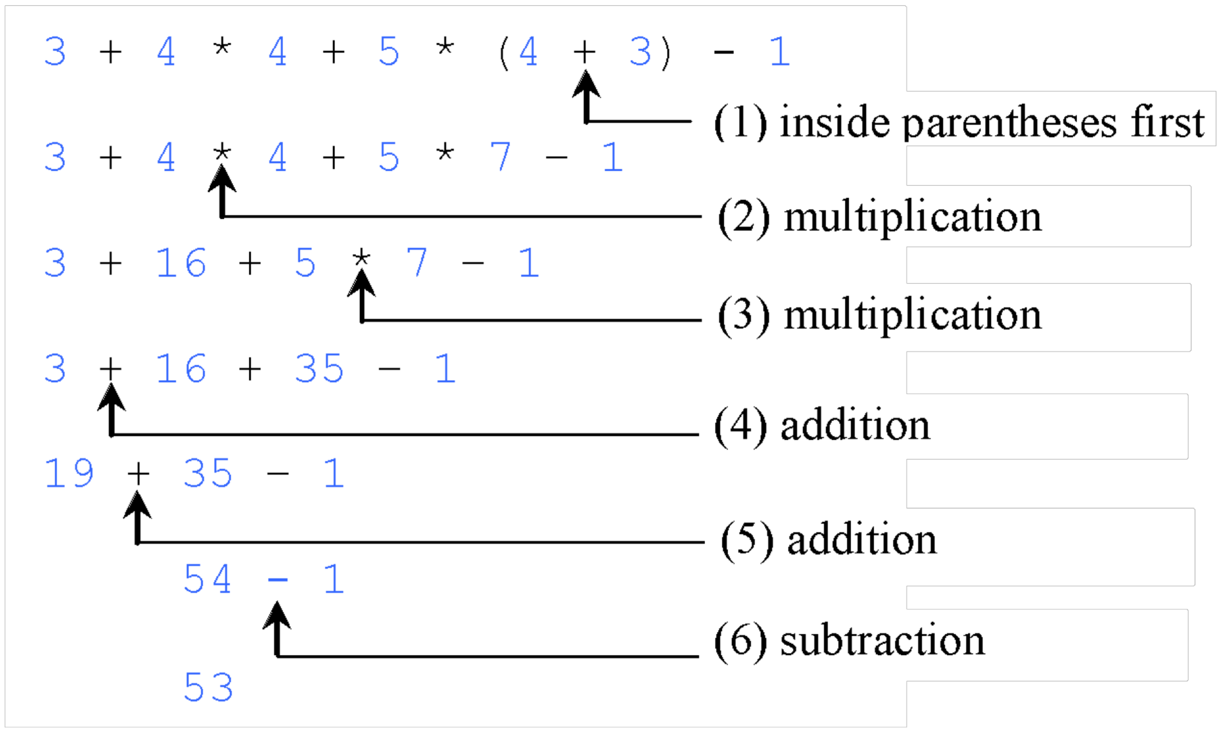
is translated to

(3+4*x)/5 – 10*(y-5)*(a+b+c)/x + 9*(4/x + (9+x)/y)

How to Evaluate an Expression

Apply the precedence and associativity rules discussed earlier To determine the order of evaluation.

Parentheses ( ) allow changing the order of evaluatioin

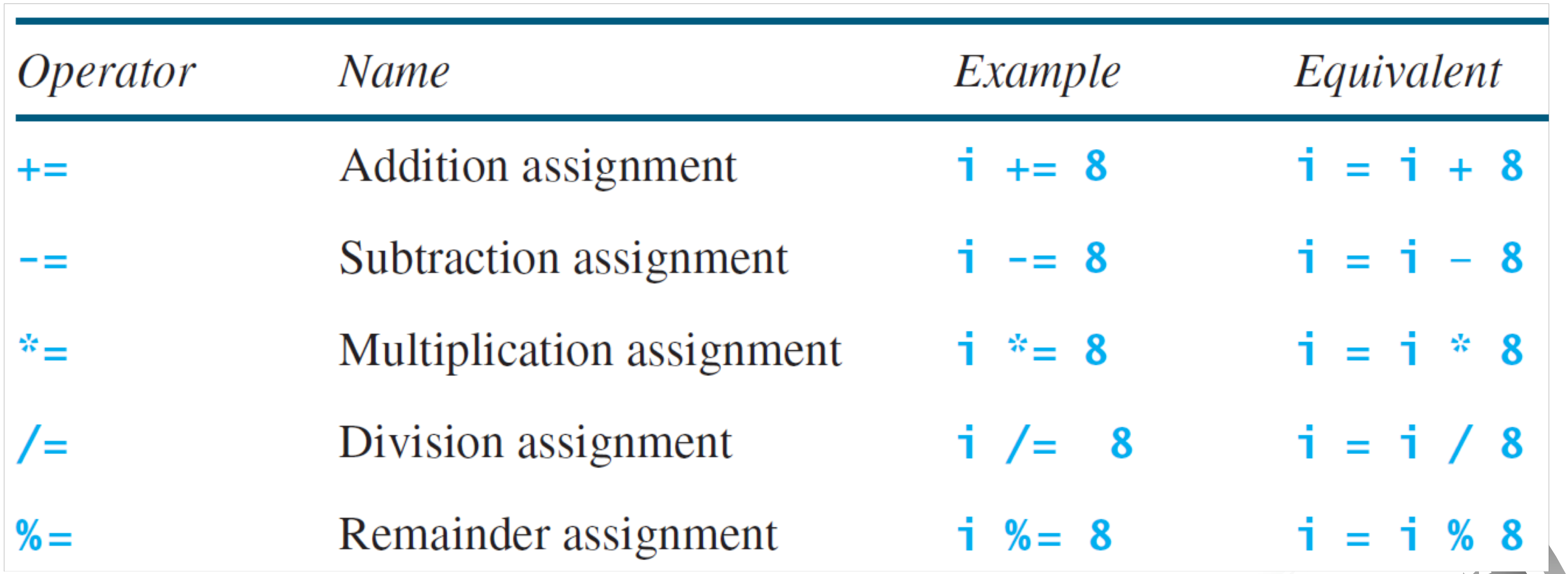


Full parenthesization

- In full parenthesization every sub expression is parenthesized. It explicitly defines the evaluation order in an expression:

5 * (3 – 2) - 3 + 4 → (((5 * (3-2)) – 3) + 4)

Augmented Assignment Operators



Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /=  8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`

Assignment Expressions and Assignment Statements

Prior to Java 2, all the expressions can be used as statements. Since Java 2, only the following types of expressions can be statements:

variable op= expression; // Where op is +, -, *, /, or %

```
++variable;
variable++;
--variable;
variable--;
```

`(because they are implicit assignment statements)`

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;

$$\xrightarrow{\text{range increases}}$$

byte, short, int, long, float, double

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.

Casting in an Augmented Expression

In Java, an augmented expression of the form **x1 op= x2** is implemented as **x1 = (T)(x1 op x2)**, where **T** is the type for **x1**. Therefore, the following code is correct.

int sum = **0**;

sum += **4.5**; // sum becomes 4 after this statement

// is equivalent to **sum = (int)(sum + 4.5)**.

Common Errors and Pitfalls

- Common Error 1: Undeclared/Uninitialized Variables and Unused Variables
- Common Error 2: Integer Overflow
- Common Error 3: Unintended Integer Division

Common Error 1: Undeclared/Uninitialized Variables and Unused Variables

double interestRate = **0.05**;

double interest = interestrate * **45**;

Common Error 2: Integer Overflow

int value = **2147483647** + **1**;

Common Error 3: Unintended Integer Division

```java
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2;
System.out.println(average);
```

(a)

```java
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2.0;
System.out.println(average);
```

(b)

Common Error 3: Round-off Errors

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

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

Let's go look what happens . . .

NOTE

Calculations involving floating-point numbers are **approximate** 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 a precise integer result.

Your Turn!

Write a program that stores the remainder of dividing the integer variable *i* by integer variable *j* in an integer variable named *k*. Use a Scanner object to get those variables from a user at the keyboard. Print *k* to the console.

```
Enter i:
Enter j:
Result k:
```