

# Elementary Programming



EECS2030 B & G: Advanced  
Object Oriented Programming  
Fall 2025

CHEN-WEI WANG

# Learning Objectives

- Learn *ingredients* of programming:
  - development process: compile time (syntax, types) vs. runtime
  - literal values
  - data types [numbers, characters, strings]
  - named constants vs. variables
  - variable (re-)assignments
  - operations, operators, operands [arithmetic, relational]
  - expressions, type correctness
  - coercion vs. casting
  - input and output
  - escape sequences
- Given a problem:
  - First, plan how you would solve it mathematically.
  - Then, *Implement* your solution by writing a Java program.

# Extra Practice?

- Java project archive:  
`Lecture_01_Elementary_Programming.zip`  
contains some of the lecture examples.  
Expanded it: *remaining examples* and *your own examples* !
- *Optional (but recommended)*: **Videos 1 – 7** from W19 Java tutorial:  
[https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java\\_from\\_scratch](https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java_from_scratch)

# Entry Point of Execution: the “main” Method

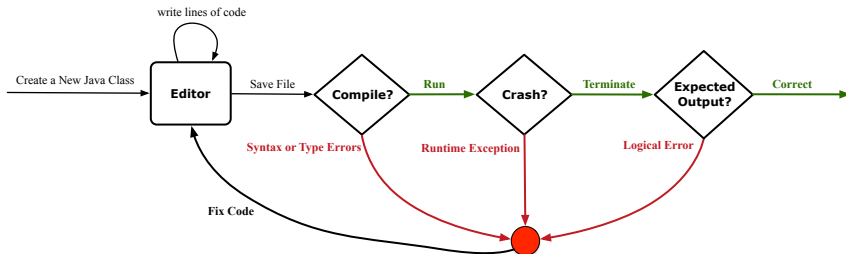
Most of your lecture examples or exercises will be defined within the context of the *main* method (for some *console application*).

```
public class MyClass {  
    public static void main(String[] args) {  
        /* Your programming solution is defined here. */  
    }  
}
```

The *main* method is treated by Java as the **starting point** of executing your program.

- **Sequential** Execution:
  - The execution starts with the first line in the *main* method, proceed line by line, from top to bottom, until there are no more lines to execute, then it **terminates**.
  - The *main* method may call methods from other classes (e.g., `model package`).

# Development Process



# Compile Time vs. Run Time

- These terms refer to two stages of developing your program.
- **Compile time**: when editing programs in Eclipse.
  - There are two kinds of **compile-time errors**:
    - **Syntax errors**: your program does not conform to Java's grammar.
      - e.g., missing the semicolon, curly braces, or round parentheses
      - Java syntax is defined in the Java language specification.
    - **Type errors**: your program manipulates data in an inconsistent way  
e.g., `"York" * 23` [  $\because$  multiplication is only for numbers ]
- **Run time** is when executing/running the **main** method.
  - **Exceptions**: your program crashes and terminates **abnormally**
    - e.g., `ArithmeticException` (e.g., `10 / 0`),  
`ArrayIndexOutOfBoundsException`, `NullPointerException`.
  - **Logical errors**: your program terminates **normally** but does not behave as expected  
e.g., calculating area of a circle with radius  $r$  using  $2 \cdot \pi \cdot r$

# Compile Time Errors vs. Run Time Errors

At the end of a programming test, if your submitted program:

- Cannot compile
- ⇒ Your program cannot even be run
- ⇒ **Zero!**

## ***What you should do :***

Practice writing as many programs as possible.

- Compiles, but run with exceptions or unexpected outputs.
- ⇒ Not necessarily zero, but likely **low marks!**

## ***What you should do :***

Truly understand the logic/rationale beyond programs.

# Always Document Your Code

- Each important design or implementation **decision** should be carefully **documented** at the right place of your code.
- Single-Lined Comments:** [Eclipse: Ctrl + /]

```
// This is Comment 1.  
... // Some code  
// This is Comment 2.
```

- Multiple-Lined Comments:** [Eclipse: Ctrl + /]

```
/* This is Line 1 of Comment 1.  
*/  
... // Some code  
/* This is Line 1 of Comment 2.  
* This is Line 2 of Comment 2.  
* This is Line 3 of Comment 2.  
*/
```

- Comments **do not affect** the runtime behaviour of programs.
- Comments are **only interpreted by** human developers.  
⇒ Useful for **revision** and **extension**.

# Literals (1)

A *literal* is a *constant value* that appears directly in a program.

## 1. *Character* Literals

- A single character enclosed within a pair of single quotes
- e.g., `'a'`, `'1'`, `'*'`, `'('`, `' '`
- It is invalid to write an empty character: `''`

## 2. *String* Literals

- A (possibly empty) sequence of characters enclosed within a pair of double quotes
- e.g., `''''`, `''a''`, `''York''`, `''*#@$''`, `'' ''`

## 3. *Integer* Literals

- A non-empty sequence of numerical digits
- e.g., 0, -123, 123, 23943

## 4. *Floating-Point* Literals

- Specified using a combination of an integral part and a fractional part, separated by a decimal point, or using the scientific notation
- e.g., 0.3334, 12.0, 34.298, 1.23456E+2 (for  $1.23456 \times 10^2$ ), 1.23456E-2 (for  $1.23456 \times 10^{-2}$ )

# Operations

An *operation* refers to the process of applying an *operator* to its *operand(s)*.

## 1. Numerical Operations

[results are numbers]

e.g.,  $1.1 + 0.34$

e.g.,  $13 / 4$

[ quotient: 3 ]

e.g.,  $13.0 / 4$

[ precision: 3.25 ]

e.g.,  $13 \% 4$

[ remainder: 1 ]

e.g.,  $-45$

e.g.,  $-1 * 45$

## 2. Relational Operations

[results are true or false]

e.g.,  $3 \leq 4$

[ *true* ]

e.g.,  $5 < 3$

[ *false* ]

e.g.,  $56 == 34$

[ *false* ]

## 3. String Concatenations

[results are strings]

e.g., ```York`` + `` `` + ``University``` is equivalent to  
```York University```

# Java Data Types

A (data) type denotes a set of related *runtime values*.

## 1. Integer Type

|       |         |                                           |
|-------|---------|-------------------------------------------|
| byte  | 8 bits  | $[-128, \dots, -1, 0, 1, \dots, 2^7 - 1]$ |
| short | 16 bits | $[-2^{15}, 2^{15} - 1]$                   |
| int   | 32 bits | $[-2^{31}, 2^{31} - 1]$                   |
| long  | 64 bits | $[-2^{63}, 2^{63} - 1]$                   |

## 2. Floating-Point Number Type

|        |         |
|--------|---------|
| float  | 32 bits |
| double | 64 bits |

## 3. Character Type

char: the set of single characters

## 4. String Type

String: the set of all possible character sequences

*Declaring a variable  $v$  to be of type  $T$  constrains  $v$  to store **only** those values defined in  $T$ .*

# Named Constants vs. Variables

A *named constant* or a *variable*:

- Is an identifier that refers to a *placeholder*
- Must be declared with its *type* (of stored value) before use:

```
final double PI = 3.14159; /* a named constant */  
double radius; /* an uninitialized variable */
```

- Can only store a value that is *compatible with its declared type*

However, a *named constant* and a *variable* are different in that:

- A named constant must be *initialized*, and cannot change its stored value.
- A variable may change its stored value as needed.

# Assignments

An **assignment** designates a value for a variable, or initializes a *named constant*.

That is, an assignment replaces the *old value* stored in a placeholder with a *new value*.

An **assignment** is done using the assignment operator (=).

An **assignment operator** has two operands:

- The *left* operand is called the *assignment target* which must be a variable name
- The *right* operand is called the *assignment source* which must be an expression whose type is **compatible** with the declared type of *assignment target*

e.g., This is a *valid* assignment:

```
String name1 = ``Heeyeon``;
```

e.g., This is an *invalid* assignment:

```
String name1 = (1 + 2) * (23 % 5);
```

# Multiple Executions of Same Print Statement

Executing *the same print statement* multiple times *may or may not* output different messages to the console.

e.g., Print statements involving literals or named constants only:

```
final double PI = 3.14; /* a named double constant */  
System.out.println("Pi is " + PI); /* str. lit. and num. const. */  
System.out.println("Pi is " + PI);
```

e.g., Print statements involving variables:

```
String msg = "Counter value is "; /* a string variable */  
int counter = 1; /* an integer variable */  
System.out.println(msg + counter);  
System.out.println(msg + counter);  
counter = 2; /* re-assignment changes variable's stored value */  
System.out.println(msg + counter);
```

# Identifiers & Naming Conventions

- Identifiers are *names* for identifying Java elements: *classes*, *methods*, *constants*, and *variables*.
- An identifier:
  - Is an arbitrarily long sequence of characters: letters, digits, underscores (\_), and dollar signs (\$).
  - Must start with a letter, an underscore, or a dollar sign.
  - Must not start with a digit.
  - Cannot clash with reserved words (e.g., class, if, for, int).
- *Valid* ids: \$2, Welcome, name, \_name, YORK\_University
- *Invalid* ids: 2name, +YORK, Toronto@Canada
- More conventions:
  - Class names are compound words, all capitalized:  
e.g., Tester, HelloWorld, TicTacToe, MagicCardGame
  - Variable and method names are like class names, except 1st word is all lower cases: e.g, main, firstName, averageOfClass
  - Constant names are underscore-separated upper cases:  
e.g., PI, USD\_IN\_WON

# Variables: Common Mistakes (1)

**Mistake:** The same variable is declared more than once.

```
int counter = 1;  
int counter = 2;
```

**Fix 1:** Assign the new value to the same variable.

```
int counter = 1;  
counter = 2;
```

**Fix 2:** Declare a new variable (with a different name).

```
int counter = 1;  
int counter2 = 2;
```

Which fix to adopt depends on what you need!

## Variables: Common Mistakes (2)

**Mistake:** A variable is used before it is declared.

```
System.out.println("Counter value is " + counter);  
int counter = 1;  
counter = 2;  
System.out.println("Counter value is " + counter);
```

**Fix:** Move a variable's declaration before its very first usage.

```
int counter = 1;  
System.out.println("Counter value is " + counter);  
counter = 2;  
System.out.println("Counter value is " + counter);
```

Remember, Java programs are always executed, line by line,  
*from top to bottom*.

# Expressions (1)

An *expression* is a composition of *operations*.

An expression may be:

- *Type Correct*: for each constituent operation, types of the *operands* are compatible with the corresponding *operator*.

e.g., `(1 + 2) * (23 % 5)`

e.g., ```Hello `` + ``world```

- *Not Type Correct*

e.g., ```46`` % ``4```

e.g., `(``YORK `` + ``University``) * (46 % 4)`

- ```YORK``` and ```University``` are both strings  
∴ LHS of `*` is *type correct* and is of type `String`
- `46` and `4` are both integers  
∴ RHS of `%` is *type correct* and is of type `int`
- Types of LHS and RHS of `*` are not compatible  
∴ Overall the expression (i.e., a multiplication) is *not type correct*

# Numerical Type Conversion: Coercion

- *Implicit* and **automatic** type conversion
- Java *automatically* converts an integer value to a real number when necessary (which adds a fractional part).

```
double value1 = 3 * 4.5;  /* 3 coerced to 3.0 */  
double value2 = 7 + 2;    /* result of + coerced to 9.0 */
```

- However, does the following work?

```
int value1 = 3 * 4.5;
```

- RHS evaluates to 13.5 due to coercion.
- LHS declares a variable for storing integers (with no fractional parts).

∴ Not compatible [ *compile-time error* ]

⇒ Need a way to “truncate” the fractional part!

# Numerical Type Conversion: Casting

- *Explicit* and *manual* type conversion
- **Usage 1:** To assign a real number to an integer variable, you need to use explicit *casting* (which throws off the fractional part).

```
int value3 = (int) 3.1415926;
```

- **Usage 2:** You may also use explicit *casting* to force precision.

```
System.out.println(1 / 2); /* 0 */
```

∴ When both operands are integers, division evaluates to quotient.

```
System.out.println( ((double) 1) / 2 ); /* 0.5 */
System.out.println( 1 / ((double) 2) ); /* 0.5 */
System.out.println( ((double) 1) / ((double) 2) ); /* 0.5 */
```

∴ Either or both of the integers operands are cast to double type

```
System.out.println((double) 1 / 2); /* 0.5 */
```

∴ Casting has *higher precedence* than arithmetic operation.

```
System.out.println((double) (1 / 2)); /* 0.0 */
```

∴ Order of evaluating division is forced, via parentheses, to occur first.

# Numerical Type Conversion: Exercise

Consider the following Java code:

```
1 double d1 = 3.1415926;  
2 System.out.println("d1 is " + d1);  
3 double d2 = d1;  
4 System.out.println("d2 is " + d2);  
5 int i1 = (int) d1;  
6 System.out.println("i1 is " + i1);  
7 d2 = i1 * 5;  
8 System.out.println("d2 is " + d2);
```

Write the **exact** output to the console.

```
d1 is 3.1415926  
d2 is 3.1415926  
i1 is 3  
d2 is 15.0
```

## Expressions (2.1)

Consider the following Java code, is each line type-correct?  
Why and Why Not?

```
1 double d1 = 23;  
2 int i1 = 23.6;  
3 String s1 = ' ';  
4 char c1 = " ";
```

- L1: YES [coercion]
- L2: No [cast assignment source, i.e., (int) 23.6]
- L3: No [cannot assign char to string]
- L4: No [cannot assign string to char]

## Expressions (2.2)

Consider the following Java code, is each line type-correct?  
Why and Why Not?

```
1 int i1 = (int) 23.6;  
2 double d1 = i1 * 3;  
3 String s1 = "La ";  
4 String s2 = s1 + "La Land";  
5 i1 = (s2 * d1) + (i1 + d1);
```

- **L1: YES** [proper cast]
- **L2: YES** [coercion]
- **L3: YES** [string literal assigned to string var.]
- **L4: YES** [type-correct string concat. assigned to string var.]
- **L5: No** [string  $\times$  number is undefined]

# Augmented Assignments

- You very often want to increment or decrement the value of a variable by some amount.

```
balance = balance + deposit;  
balance = balance - withdraw;
```

- Java supports special operators for these:

```
balance += deposit;  
balance -= withdraw;
```

- Java supports operators for incrementing or decrementing by 1:

```
i ++; j --;
```

- Confusingly**, these increment/decrement assignment operators can be used in assignments:

```
int i = 0; int j = 0; int k = 0;  
k = i ++; /* k is assigned to i's old value */  
k = ++ j; /* k is assigned to j's new value */
```

# Escape Sequences

An *escape sequence* denotes a single character.

- Specified as a *backslash* (\) followed by a *single character*
  - e.g., \t, \n, \', \", \\
- *Does not mean literally*, but means specially to Java compiler
  - \t means a tab
  - \n means a new line
  - \\ means a back slash
  - \' means a single quote
  - \" means a double quote
- May use an *escape sequence* in a character or string literal:
  - ' ' [INVALID; need to escape ']
  - \' ' [VALID]
  - " " [VALID; no need to escape "]
  - "" " [INVALID; need to escape "]
  - \" " [VALID]
  - "' " [VALID; no need to escape ']
  - "\n\t\" " [VALID]

# Case Study 1: Compute the Area of a Circle

**Problem:** declare two variables `radius` and `area`, initialize `radius` as 20, compute the value of `area` accordingly, and print out the value of `area`.

```
public class ComputeArea {  
    public static void main(String[] args) {  
        double radius; /* Declare radius */  
        double area; /* Declare area */  
        /* Assign a radius */  
        radius = 20; /* assign value to radius */  
        /* Compute area */  
        area = radius * radius * 3.14159;  
        /* Display results */  
        System.out.print("The area of circle with radius ");  
        System.out.println(radius + " is " + area);  
    }  
}
```

It would be more flexible if we can let the user specify the inputs via keyboard!

# Input and Output

*Reading* input from the console enables *user interaction*.

```
import java.util.Scanner;
public class ComputeAreaWithConsoleInput {
    public static void main(String[] args) {
        /* Create a Scanner object */
        Scanner input = new Scanner(System.in);
        /* Prompt the user to enter a radius */
        System.out.print("Enter a number for radius: ");
        double radius = input.nextDouble();
        /* Compute area */
        final double PI = 3.14169; /* a named constant for  $\pi$  */
        double area = PI * radius * radius; /*  $area = \pi r^2$  */
        /* Display result */
        System.out.println(
            "Area for circle of radius " + radius + " is " + area);
    }
}
```

# Useful Methods for Scanner

---

- *nextInt()* which reads an integer value from the keyboard
- *nextDouble()* which reads a double value from the keyboard
- *nextLine()* which reads a string value from the keyboard

## Case Study 2: Display Time

**Problem:** prompt the user for an integer value of seconds, divide that value into minutes and remaining seconds, and print the results. For example, given an input 200, output “200 seconds is 3 minutes and 20 seconds”.

```
import java.util.Scanner;
public class DisplayTime {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        /* Prompt the user for input */
        System.out.print("Enter an integer for seconds: ");
        int seconds = input.nextInt();
        int minutes = seconds / 60; /* minutes */
        int remainingSeconds = seconds % 60; /* seconds */
        System.out.print(seconds + " seconds is ");
        System.out.print(" minutes and ");
        System.out.println(remainingSeconds + " seconds");
    }
}
```

# Where May Assignment Sources Come From?

In `tar = src`, the *assignment source* `src` may come from:

- A literal

```
int i = 23;
```

- A variable

```
int i = 23;  
int j = i;
```

- An expression involving literals and variables

```
int i = 23;  
int j = i * 2;
```

- An input from the user

```
Scanner input = new Scanner(System.in);  
int i = input.nextInt();  
int j = i * 2;
```

## Literals (2)

Q. Outputs of `System.out.println('a')` versus `System.out.println("`a`")`? [SAME]

Q. Result of comparison ``a` == 'a'``? [TYPE ERROR]

- Literal ``a`` is a string (i.e., *character sequence*) that consists of a single character.
- Literal `'a'` is a single *character*.

∴ You cannot compare a character sequence with a character.

## print VS. println

Executing `System.out.println(someString)` is the same as executing `System.out.print(someString + "\n")`.

- e.g.,

```
System.out.print("Hello");  
System.out.print("World");
```

```
HelloWorld
```

- e.g.,

```
System.out.println("Hello");  
System.out.println("World");
```

```
Hello  
World
```

## Beyond this lecture...

- Create a *console tester* in Eclipse.  
Try out the examples given in the slides.
- *Optional (but recommended)*: **Videos 1 – 7** from W19 Java tutorial:  
`https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java\_from\_scratch`
- See `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html` for more information about data types in Java.

# Index (1)

---

**Learning Objectives**

**Extra Practice?**

**Entry Point of Execution: the “main” Method**

**Development Process**

**Compile Time vs. Run Time**

**Compile Time Errors vs. Run Time Errors**

**Always Document Your Code**

**Literals (1)**

**Operations**

**Java Data Types**

**Named Constants vs. Variables**

# **Index (2)**

---

## **Assignments**

### **Multiple Executions of Same Print Statement**

### **Identifiers and Naming Conventions in Java**

### **Variables: Common Mistakes (1)**

### **Variables: Common Mistakes (2)**

### **Expressions (1)**

### **Numerical Type Conversion: Coercion**

### **Numerical Type Conversion: Casting**

### **Numerical Type Conversion: Exercise**

### **Expressions (2.1)**

### **Expressions (2.2)**

## **Index (3)**

---

**Augmented Assignments**

**Escape Sequence**

**Case Study 1: Compute the Area of a Circle**

**Input and Output**

**Useful Methods for Scanner**

**Case Study 2: Display Time**

**Where May Assignment Sources Come From?**

**Literals (2)**

**`print` VS. `println`**

**Beyond this lecture...**