

# Topic 3: Introduction to Java

1. Higher-level languages and software development
2. Java basics
3. documentation
4. Syntax and semantics

*Reading:*  
Ch. 1.2-1.3,  
2.1, 2.3-2.4

## Objectives

1. Use a programming environment to compile and test a Java program
2. Code screen output and keyboard input in Java
3. Recognize basic syntax rules of Java
4. Write appropriate comments to document code

## 1. Higher-level languages and software development

- Higher-level languages
  - support I/O, control structures, and modularity
  - shield programmer from hardware and operating-system details
  - are *portable* (compilable to any runtime environment)
  - are translated to machine language or byte code by compilers or interpreters
- *Examples:* COBOL, Fortran, Pascal, C, C++, Java

## Interpreting vs. compiling

- *Interpreted code* is executed one instruction at a time from input stream
- *Examples:* machine code, Java byte code, command line
- *Compilers* translate code from high-level languages like Java to low-level form that can be interpreted

## The Java virtual machine

- The Java compiler translates Java code to an assembler-like language called “byte code”
- The JVM is a program that interprets byte-code instructions
- The JVM simulates a real processor
- The *java* program at the command line, and any Internet browser, contain JVMs
- The *class loader* in *java* allows program statements from different *.class* files to use each other

## Integrated development environments

- *Editor* enables code entry and modification, with syntax highlighting
- *Compiler* translates Java to machine code or byte code and provides *warnings* and *error diagnostics*
- *Debugger* enables trace of variables
- *Help systems* provide reference
- *Examples*: NetBeans, BlueJ, Eclipse
- *Java Development Kit* (Sun) provides compiler, debugger

## Projects

- Most IDEs organize Java programs as *projects* consisting of multiple source-code files
- *Examples:* BlueJ, Eclipse, NetBeans
- Often programmers write one source file per Java class, compile source files separately, link compiled *.class* files
- If *.jar* file is produced, then it is executable alone if the Java runtime environment is on the computer

## 2. Java basics

```
// Hello.java: displays greeting
public class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello");
    }
}
```

- Defines a class *Hello* and a method *main*, calls method *println*, all to be explained later
- Example code on future slides will omit class and method definitions

## Learning Java

- To start learning Java, you need example programs
- To fully understand the code of the simplest example program, you must understand *methods* and *classes* (Topic 6)
- Hence to start learning Java, you must work with examples you don't fully understand
- For now it is enough to know that every Java program must have a *class* with the same name as the program, and a definition for the method *main* of that class

## A program to add numbers

1. Prompt for integers *input1*, *input2*
2.  $sum \leftarrow input1 + input2$
3. Display *sum*

```

System.out.print("Enter 2 integers: ");
Scanner in = new Scanner(System.in);
int x1, x2, sum;
x1 = in.nextInt();
x2 = in.nextInt();
sum = x1 + x2;
System.out.print("Sum is ");
System.out.println(sum);

```

*prompt*

*variable declarations*

*input statements*

*assignment*

*See section 7 for explanation of stream I/O*

## Java statements

- *Kinds:*
  - a.variable declarations
  - b.assignments
  - c.method call
  - d.branch or loop
  - e.compound statement
- (a) – (d) end with a semicolon (;)
- Compound statements are enclosed by braces

## Java tokens

*Tokens* are basic elements from which a program is built

- Reserved words (e.g., *int, double, char, void, public, class*)
- Identifiers (*main, System, out, println*)
- Operators (+, =, -, \*, /, %)
- Delimiters ({, }, (, ), ., ;, “)

A token cannot be subdivided

## Variables and assignments

- *Variable*: a named data storage location that
  - has an *address*
  - may store a *value*
  - has a *data type*, e.g., *int*
- Program must declare a variable before using it
- A variable gets its value by *initialization*, *assignment*, or *input*
- *Example*: `int x;`

[Examples]

## Assignments and initializations

- An *assignment expression* has a value:

```
int a,b;      Expression b = 2
a = b = 2;   has the value 2
           Assignment operator
```

- An *initialization statement* is a declaration, and the initialization operator (=) produces no expression:

```
int n = 4;
           Initialization operator
```

## Binary operators

Operator(s)	Associativity	Example	Value	Side effect
+, −	left	3 − 1 + 4	6	none
*, /, %	left	2 * 6 / 4	3	none
=	right	a = b = 1	1	assign

- A binary expression may have both a value and a side effect
- The value may be used in chaining
- Associativity affects result value in chaining

## Generations of programming languages

- *First generation*: Machine languages (binary)
- *2<sup>nd</sup> generation*: Assembler languages, processor specific
- *3<sup>rd</sup> generation*: Procedural, high-level, hardware independent (C, BASIC, JavaScript)
- *4<sup>th</sup> generation*: Nonprocedural query or report-generation languages (SQL, RPG)
- *Declarative* languages (Prolog)
- *Object-oriented* languages (Java)
- *Functional* languages (Lisp)

## Input and output

- Standard I/O devices are keyboard, screen
- To input an integer:  
`Scanner in = new Scanner(System.in);`  
`int x = in.nextInt();`
- To output a value:  
`System.out.println("Hello");`
- The *println* method adds a newline after its output, to omit newline, use *print*
- *Scanner*, *nextInt*, and *System* are discussed later.

## Input/output with GUIs

- *JOptionPane* is a standard class in the *javax.swing* package that enables I/O via dialog boxes
- *gui\_add.java* adds two numbers using GUI I/O
- `import javax.swing;`
- ...
- `int a = Integer.parseInt(JOptionPane.showInputDialog(null, "Enter an integer"));`
- `int b = Integer.parseInt(JOptionPane.showInputDialog(null, "Enter another integer"));`
- `int sum = a + b;`
- `JOptionPane.showMessageDialog(null, a + " + " b + " = " + sum);`

## Ways to give a value to a variable

<i>Name</i>	<i>Example</i>	<i>Who chooses</i>
Initialization	<code>int n = 3;</code>	programmer
Assignment	<code>n = 3;</code>	programmer
Input	<code>n = in.nextInt();</code>	user

## Giving a value to a constant

- Named constants  
*Example:* `final float price = 3.95;`
- Literals cannot take new value  
*Invalid:* ~~`3.95 = price;`~~

## 3. Documentation

### *Guidelines:*

- State purpose of every program and every component of large programs at top
- Give meaningful names to variables
- Use named constants
- Use comments to clarify intention
- Format source code for clarity

## Named constants

```
final int MO_PER_YR = 12;  
  
int per_month = 26; // monthly cost  
System.out.print("Yearly rate: $");  
System.out.print(MO_PER_YR * per_month);
```

- Variables tagged *final* cannot get new values
- Named constants are reusable, updatable
- Constants predefined in the Java *Math* class:  
*Math.E*, *Math.PI*

## Comments

- Comments document a program to help make it readable and understandable
- To be debugged or maintained, a program must be understandable
- `//` starts a comment until the end of the source line
- `/*` and `*/` delimit a (possibly multi-line) comment
- *Guideline*: write a comment wherever necessary to make intention clear
- Every program must have a comment at the top, with file name, purpose of program, name of programmer, date

## Writing clear comments

- Not clear:  
`// Computes result from input.`
- Not clear:  
`/* Displays the absolute value  
of the difference. */`
- Clear:  
`/* This program prompts for 2 integers  
and displays the absolute value of  
the difference between them. */`
- A comment is usually a narrative that tells the story of what a program does, or is an assertion about values present as the program runs.

## Formatting source code

- *Example:*  

```
void main()  
{  
    System.out.println("Hello");  
}
```
- Leave an empty line before a method definition such as *main*
- Align pairs of braces vertically
- Indent statements 2-3 spaces
- Comments and readability are a major factor in effective programming

## 4. Syntax and semantics

- *Syntax* is the set of grammar rules that define a language formally
- *Semantics* is the set of meanings of each of the syntax elements
- The compiler handles a syntax error by halting and displaying a message (often misleading)
- The compiler follows semantics by generating the appropriate machine code for statements

## Specifying grammar rules

- A language is a set of strings, e.g., the set of all possible Java programs
- A grammar is a set of rules for what is permitted in a language
- Java *tokens* are formed by simple rules; e.g., an integer literal is a series of digits
- Higher-level (*nonterminal*) components (*program, statement, expression, etc.*) are built from tokens or other nonterminals

## Kinds of tokens (lexical elements)

- keyword (*void, main, int, ...*)
- identifier (letter or ‘\_’ followed by a series of letters, digits, ‘\_’s)
- constant literal (numeral, double-quoted string, single quoted character)
- operator (=, +, \*, -)
- punctuator (semicolon, comma, paren, brace)

### *Not tokens (Ignored by compiler):*

- white space (space characters, tabs, newlines)
- comments (*//..., /\*...\*/*)

## The compiler is case sensitive

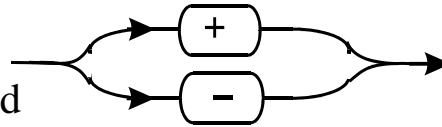
- **Total** is a different identifier from **total**
- **Int** is not a keyword

## Identifiers and operators

- May not contain spaces,
- Are different from literals:
  - “ + ” is not same as +
  - ”input” is not same as ID **input**
  - ”2” is not same as numeral 2

## Ways to specify syntax

- *Plain English* (e.g., “A compound statement is a series of statements, in braces”)
- *List of alternatives*; e.g.:  
*statement-list:*  
*nothing*  
*statement statement-list*
- *Diagram*;  
e.g., *sign*  
may be diagrammed  
as at right



## Syntax rules and diagrams

*compound-statement:*

{ *statement-list* }

*statement-list:*

*nothing*

*statement statement-list*

*statement:*

*declaration*

*assignment*

*IO-statement*

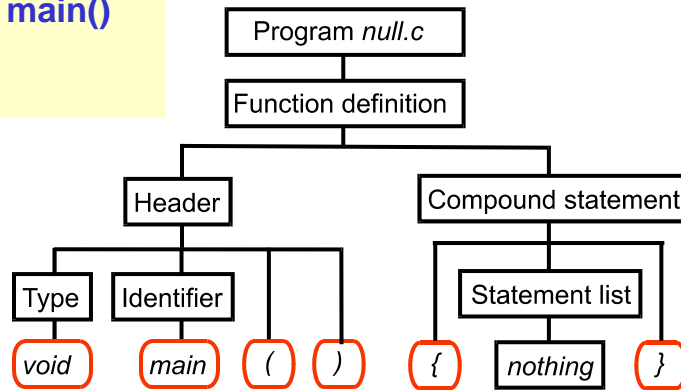
*compound-statement*

Diagram for  
*statement-list*



## The parser generates a parse tree

```
void main()
{
}
```



- Each syntax rule is applied by putting a defined element's components under the name of the element

## References

Cay Horstmann. *Big Java*, 3<sup>rd</sup> ed. Wiley, 2008, Chs. 1-2, 4.