3. Java basics

1. Higher-level languages and compilation
2. Java programs and statements
3. Program documentation
4. Program syntax

Inquiry

• What is a more convenient way to program a computer than assembler language?
• What programming languages have you used?
3. Write, document, and test a Java program

**Topic objective**

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

**Subtopic outcomes**

3.1 Describe the Java development environment*

3.2a Compile and test a Java program*

3.2b Code screen output and keyboard input in Java*

3.3a Explain code documentation*

3.3b Write appropriate comments to document code*

3.4 Identify basic syntax rules of Java*
1. Higher-level languages and compilation

- What is Java?
- How do Java and HTML compare?
- Have you programming in other languages?
- Have you heard of *compilers*?

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
Kinds of programming languages

- *Procedural* (C, C++, Java)
- *Declarative* (Prolog)
- *Object-oriented* (C++ Java)
- *Functional* (Lisp)
- *Programming* languages differ from HTML in that HTML is a *formatting* language
- XML uses tags to label items for meaning, for use in data storage

Generations of programming languages

- *First generation*: Machine languages (binary)
- *2nd generation*: Assembler languages, processor specific
- *3rd generation*: Procedural, high-level, hardware independent (COBOL, Fortran, JavaScript, Java)
- *4th generation*: Nonprocedural query or report-generation languages (SQL, RPG)
Interpreting vs. compiling

- Interpreted code is executed one instruction at a time from input stream (machine code, Java byte code, command line)

  [Diagram: one instruction fetched → Interpreter → one instruction executed]

- Compilers translate code from high-level languages like Java to low-level form that can be interpreted

  [Diagram: Java program → Java compiler → byte-code program]

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, simulating 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 invoke each other
Integrated development environments

- *Editor* enables code entry and modification, with syntax highlighting
- *Compiler* translates Java to machine code or byte code; provides *warnings*; *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 developers create 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
Separate compilation and projects

- **Project**: a set of source files ready to link together after compilation
- Exactly one source file has definition of `main`
- Other source files (e.g., libraries) may define functions called from `main`, etc.
- Programmer selects files to comprise the project
- **Build** command automatically recompiles any source files modified since last compile and links them into an `.exe` file

Projects and object files

- **Project**: a set of source files ready to link together after compilation
- Programmer selects files to include based on class and function dependencies
- Compilation produces intermediate linkable object files (.obj or .o), which may or may not define `main`; some function addresses may not be resolved
- **Build** command automatically compiles any source files modified since last compile
The linker finishes the compiler’s work

- The linker resolves unresolved function calls (i.e., jump addresses to subroutines corresponding to function definitions)

Subtopic outcome

3.1 Describe the Java development environment*

---

* Subtopic outcome
2. Java programs and statements

• What code does every Java program contain?
• How does Java compare with pseudocode?
• What are the simplest elements of a Java program?
• How does Java support operations and input/output?

A minimal program

// 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, both to be explained later
• Example code on future slides will omit class and method definitions
Learning Java

• Learning Java can start with example programs
• To understand well the code of any program, you’d have to understand *methods* and *classes* (Topic 6)
• Hence to start learning Java, you may work with examples you don’t fully understand
• *In short:* Every Java program must define 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 ← input1 + input2*
3. Display *sum*

```java
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);
```

See topic 4 for explanation of stream I/O
Input and output

- Standard I/O devices are keyboard, screen
- To input an integer:
  ```java
  Scanner in = new Scanner(System.in);
  int x = in.nextInt();
  ```
- To output a value:
  ```java
  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.

Variables and assignments

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

- An assignment expression has a value:
  ```java
  int a, b;
  Expression b = 2
  a = b = 2; has the value 2
  Assignment operator
  ```

- An initialization statement is also a declaration

- The initialization operator (=) produces no expression:
  ```java
  int n = 4;
  ```

Ways to give a value to a variable

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Who chooses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td><code>int n = 3;</code></td>
<td>programmer</td>
</tr>
<tr>
<td>Assignment</td>
<td><code>n = 3;</code></td>
<td>programmer</td>
</tr>
<tr>
<td>Input</td>
<td><code>n = in.nextInt();</code></td>
<td>user</td>
</tr>
</tbody>
</table>

**Giving a value to a constant**

- Named constants
  
  *Example:* `final float price = 3.95;`

- Literals cannot take new value
  
  *Invalid:* `3.95 = price;`
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

```java
import javax.swing;
...
int a = Integer.parseInt(JOptionPane.showInputDialog(null,"Enter an integer"));
int b = Integer.parseInt(JOptionPane.showMessageDialog(null,"Enter another int"));
int sum = a + b;
JOptionPane.showMessageDialog(null,
a + " + " b + " = " + sum);
```

Java packages

- **Package**: A set of related class definitions
- A package is compiled together into a single class file
- A package is made available to a source file using the `import` directive
- Classes and methods in packages not imported in this way may be used by listing their names before method calls
- The **standard library** package defines the `System` class
Subtopic outcomes

3.2a Use a programming environment to compile and test a Java program*
3.2b Code screen output and keyboard input in Java*

3. Program documentation

• When you write a program, is it always clear why you wrote a certain line?
• When you debug a program, do you need to know the original intention?
• What elements of a Java code does the compiler ignore?
• Why do professional developers care about these elements?
Why document code?

- Code is often hard to understand
- Code is often used by persons other than the original coder
- Programmers need *clarity* about code in order to *use* and *maintain* it

Documentation guidelines

- State *purpose* of every program and every component of large programs at top
- Give meaningful names to *variables*
- Use well-named *constants*
- Use *comments* to clarify intention
- *Format* source code for clarity
- To be debugged or maintained, a program must be understandable
Well-named constants

```java
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 help make programs readable and understandable
• `//` starts a comment that lasts until the end of the source line
• `/*` and `*/` delimit a (possibly multi-line) comment
• `Guideline`: write a comment wherever necessary to make your intention clear
• Professional programs 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:
  ```java
  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
- Readability is a major factor in effective programming
Subtopic outcomes

3.3a Explain code documentation*
3.3b Write appropriate comments to document code*

4. Program syntax

- Does the compiler always translate source code into object code?
- What errors does the compiler flag?
- What is syntax?
- What is semantics?
- What is English grammar like? Java grammar?
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.
Java lexical tokens

- *Tokens* are the most 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 ({, }, (, ), ., ;, “)
  - Constants (12, “Hello”)

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 (//…, /*…*/
Some syntax rules

- **Case sensitivity:**
  - `Total` is a different identifier from `total`
  - `Int` is not a keyword

- White space: identifiers and operators may not contain spaces,

- Identifiers are different from quoted literals

- Quotation marks:
  - “+” is not same as +
  - “input” is not same as the 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.:
  
  ```java
  statement-list:
  nothing
  statement statement-list
  ```

- **Diagram;** e.g., `sign`
  
  may be diagrammed as at right
Java statements

- **Kinds:**
  - variable declarations
  - assignments
  - method call
  - branch or loop
  - compound statement
- All but compound statement end with a semicolon (;)
- Compound statements are enclosed by braces

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

Programming-language semantics

• Coders become familiar with semantics in order to avoid logic errors
• To avoid unintentional errors, be aware that
  – the values of uninitialized variables are unpredictable
  – int variables cannot store fractional values
• The compiler cannot detect semantics-related errors
Subtopic outcome

3.4 Identify basic syntax rules of Java

References