Java Syntax & Data Types

Programming with Java: Session 2

Zirui Zhang

Tornado Engineering Club
Jinan Foreign Language School

September 20, 2025

Courtesy: https://liaoxuefeng.com/books/java/

Complete Java Program Example

```java
/**
 * Comments for documentation generation
 */
public class Hello {
    public static void main(String[] args) {
        // Output text to the screen:
        System.out.println("Hello, world!");
        /* Multi-line comment start
        Comment content
        Comment end */
    }
} // End of class definition
```

Class Structure in Java

- Java is object-oriented → basic unit is a `class`
- `class` is a keyword
- Class name follows specific naming conventions

```
1 public class Hello { // Class name is Hello
2     // Class body...
3 } // End of class definition
```

Class Naming Conventions

Good Class Names:

- `Hello`
- `NoteBook`
- `VRPlayer`

Bad Class Names:

- `hello` (should start with uppercase)
- `Good123` (meaningless numbers)
- `Note_Book` (avoid underscores)
- `_World` (should not start with underscore)

Access Modifiers

- `public` indicates the class is accessible everywhere
- If omitted: program compiles but cannot be executed from command line
- Classes can contain multiple methods

```
1 public class Hello {
2     public static void main(String[] args) {
3         // Method code...
4     }
5 }
```

Method Structure

- Methods define execution statements
- Code executes sequentially
- `main` method is the program entry point
- `void` indicates no return value
- `static` indicates a static method

```
1 public static void main(String[] args) {
2     System.out.println("Hello, world!");
3 }
```

Method Naming Conventions

Good Method Names:

- `main`
- `goodMorning`
- `playVR`

Bad Method Names:

- `Main` (should start with lowercase)
- `good123` (meaningless numbers)
- `good_morning` (avoid underscores)
- `_playVR` (should not start with underscore)

Statements and Syntax

- Statements are executable code
- Each statement must end with a semicolon
- Statements execute in sequence

```java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!"); // Statement
        int x = 5; // Another statement
    }
}
```

Comments in Java

Single-line Comments:

```
// This is a single-line comment
System.out.println("Hello"); // Comment after code
```

Multi-line Comments:

```
/*
This is a multi-line comment
that spans multiple lines
*/
```

Documentation Comments

- Special multi-line comments for documentation
- Start with /** and end with */
- Used to generate automatic documentation
- Place before class/method definitions

```
/**
 * Class description for documentation
 *
 * @author Developer Name
 * @version 1.0
 */
public class Hello {
    // Class implementation...
}
```

Code Formatting

- Java is flexible with whitespace
- Extra spaces/line breaks don't affect compilation
- Follow community coding conventions
- Use IDE formatting tools (Eclipse: `Ctrl+Shift+F`)
- Consistent formatting improves readability

IDE Settings:

`Java → Code Style` in Eclipse preferences

Summary

- Java programs are organized in classes
- `main` method is the entry point
- Follow naming conventions for classes and methods
- Use proper comments for documentation
- Maintain consistent code formatting
- Statements must end with semicolons

What is a Variable?

- A concept from algebra (e.g., ‘x‘, ‘y‘ in equations).
- In Java, variables are divided into two types:
  - Primitive type variables
  - Reference type variables

```
1 // Example equation in algebra:
2 y = x^2 + 1
```

Defining Variables

- Variables must be defined before use.
- Can assign an initial value during definition.
- If no initial value, assigned a default value (usually '0').

```java
int x = 1; // Define an int variable x with initial value 1

// Example: Define and print a variable
public class Main {
    public static void main(String[] args) {
        int x = 100;
        System.out.println(x); // Prints 100
    }
}
```

Reassigning Variables

- Variables can be reassigned new values.
- Do not specify the type again when reassigning.

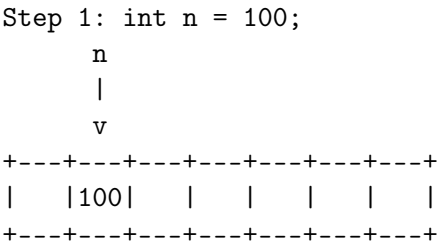
```java
public class Main {
    public static void main(String[] args) {
        int x = 100; // Define with initial value
        System.out.println(x); // Prints 100
        x = 200;     // Reassign (no 'int' here)
        System.out.println(x); // Prints 200
    }
}
```

Assigning Between Variables

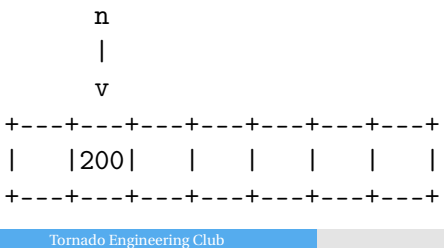
```java
public class Main {
    public static void main(String[] args) {
        int n = 100;
        System.out.println("n = " + n); // n = 100

        n = 200;
        System.out.println("n = " + n); // n = 200

        int x = n; // Assign n's value (200) to x
        System.out.println("x = " + x); // x = 200

        x = x + 100; // x becomes 300
        System.out.println("x = " + x); // x = 300
        System.out.println("n = " + n); // n = 200
    }
}
```

Memory Representation

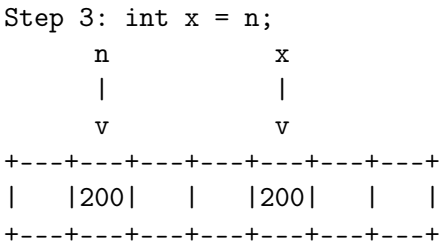
Step 1: int n = 100;



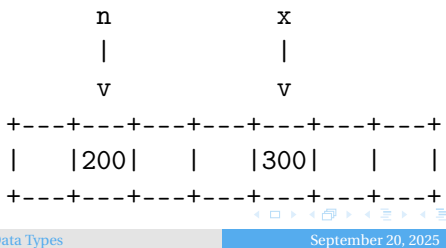
Step 2: n = 200;



Step 3: int x = n;



Step 4: x = x + 100;



Primitive Data Types

- Types the CPU can directly operate on.
- Java defines:
  - Integer types: ‘byte‘, ‘short‘, ‘int‘, ‘long‘
  - Floating-point types: ‘float‘, ‘double‘
  - Character type: ‘char‘
  - Boolean type: ‘boolean‘

Integer Types

- Ranges:
  - ‘byte‘: -128 127
  - ‘short‘: -32768 32767
  - ‘int‘: -2147483648 2147483647
  - ‘long‘: -9223372036854775808 9223372036854775807

```
int i = 2147483647;
int i2 = -2147483648;
int i3 = 2_000_000_000; // Underscores for readability
int i4 = 0xff0000;      // Hex: 16711680
int i5 = 0b1000000000;  // Binary: 512

long n1 = 9000000000000000000L; // Requires L suffix
long n2 = 900;          // OK, int 900 assigned to long
int i6 = 900L;          // Error: Cannot assign long to int
```

Floating-Point Types

- Numbers with a decimal point (or scientific notation).
- ‘float‘ requires an ‘f‘ suffix.

```
float f1 = 3.14f;
float f2 = 3.14e38f; // 3.14 x 10^38
float f3 = 1.0;      // Error: 1.0 is a double

double d = 1.79e308;
double d2 = -1.79e308;
double d3 = 4.9e-324; // 4.9 x 10^-324
```

Boolean Type

- Only two values: ‘true‘ and ‘false‘.
- Often the result of relational operations.
- Stored as 4-byte integers in the JVM.

```
boolean b1 = true;
boolean b2 = false;
boolean isGreater = 5 > 3; // true
int age = 12;
boolean isAdult = age >= 18; // false
```

Character Type

- Represents a single character.
- Uses single quotes ‘’.
- Can represent Unicode characters.

```
1 public class Main {
2     public static void main(String[] args) {
3         char a = 'A';
4         char zh = '\u4E2D';
5         System.out.println(a);    // A
6         System.out.println(zh);   // \u4E2D
7     }
8 }
```

Reference Types

- All non-primitive types are reference types.
- Store an address pointing to an object in memory.
- ‘String‘ is a common reference type.

```
1 String s = "hello"; // s is a reference type variable
```

Constants

- Defined with the ‘final‘ modifier.
- Cannot be reassigned after initialization.
- Use uppercase names by convention.
- Avoid "magic numbers" in code.

```
final double PI = 3.14; // Constant
double r = 5.0;
double area = PI * r * r;
PI = 300; // Compile error!
```

The ‘var‘ Keyword

- Lets the compiler infer the variable type.
- Makes code less verbose.

```
// Without var
StringBuilder sb = new StringBuilder();

// With var
var sb = new StringBuilder(); // Compiler infers StringBuilder
```

Variable Scope

- Scope is defined by ‘ ... ‘ blocks.
- Variable is accessible from its point of definition to the end of its block.
- Minimize variable scope.
- Avoid reusing variable names in nested scopes.

```
{
    int i = 0;
    {
        int x = 1;
        {
            String s = "hello";
        } // s scope ends
        String s = "hi"; // OK, different variable
    } // x and s scope end
} // i scope ends
```

Summary

- Two variable types: **primitive** and **reference**.
- Primitive types: integers, floats, booleans, characters.
- Variables can be **reassigned**. ‘=‘ is assignment, not equality.
- **Constants** (‘final‘) cannot be reassigned.
- Use **‘var‘** for cleaner code when the type is obvious.
- Define variables in the **smallest possible scope**.

Integer Operations

- Follow arithmetic rules with nested parentheses
- Integer operations are always exact

```java
public class Main {
    public static void main(String[] args) {
        int i = (100 + 200) * (99 - 88); // 3300
        int n = 7 * (5 + (i - 9)); // 23072
        System.out.println(i);
        System.out.println(n);
    }
}
```

Division and Remainder

- Integer division yields integer part only
- Remainder operation uses ‘%‘

```
int x = 12345 / 67; // 184
int y = 12345 \% 67; // 17 (remainder)

// Division by zero causes runtime error
int z = 100 / 0; // ArithmeticException
```

Overflow

- Occurs when result exceeds integer range
- No error thrown, produces unexpected results

```java
public class Main {
    public static void main(String[] args) {
        int x = 2147483640;
        int y = 15;
        int sum = x + y;
        System.out.println(sum); // -2147483641
    }
}
```

Use `long` for larger range:

```java
long x = 2147483640;
long y = 15;
long sum = x + y; // 2147483655
```

Compound Assignment Operators

- Shorthand operators for common operations

```
int n = 3300;
n += 100; // n = n + 100; -> 3400
n -= 100; // n = n - 100; -> 3300
n *= 2;   // n = n * 2;   -> 6600
n /= 3;   // n = n / 3;   -> 2200
n %= 100; // n = n % 100; -> 0
```

Increment/Decrement Operators

- ++ increments, – decrements
- Position matters: prefix vs postfix

```
int n = 3300;
n++; // 3301 (postfix: use then increment)
++n; // 3302 (prefix: increment then use)

int a = n++; // a = 3302, n = 3303
int b = ++n; // n = 3304, b = 3304

// Avoid complex expressions with ++/--
```

Bitwise Shift Operations

- Left shift: <<, Right shift: >>
- Unsigned right shift: >>>

```
int n = 7;          // 00000111 = 7
int a = n << 1;     // 00001110 = 14
int b = n << 2;     // 00011100 = 28
int c = n >> 1;     // 00000011 = 3
int d = n >> 2;     // 00000001 = 1

int neg = -536870912;
int e = neg >> 1;   // preserves sign bit
int f = neg >>> 1; // fills with 0
```

Bitwise Operations

- AND: &, OR: |, NOT: ~, XOR: ^

```java
int i = 167776589; // 00001010 00000000 00010001 01001101
int n = 167776512; // 00001010 00000000 00010001 00000000

int and = i & n;    // 167776512
int or = i | n;     // 167776589
int xor = i ^ n;    // 77
int not = ~i;       // -167776590
```

Operator Precedence

- Highest: ()
- !, ~, ++, -
- *, /, %
- +, -
- «, », »>
- &
- |
- Lowest: +=, -=, *=, /=

```
// Use parentheses for clarity
int result = (a + b) * (c - d) / e;
```

Type Promotion and Casting

- Operations promote to larger type
- Explicit casting may cause data loss

```
short s = 1234;
int i = 123456;
int x = s + i; // promoted to int
// short y = s + i; // compilation error

short y = (short)(s + i); // explicit cast
// Risk of incorrect results if value too large
```

Casting Risks

- Casting large values produces unexpected results

```java
public class Main {
    public static void main(String[] args) {
        int i1 = 1234567;
        short s1 = (short) i1; // -10617
        System.out.println(s1);

        int i2 = 12345678;
        short s2 = (short) i2; // 24910
        System.out.println(s2);
    }
}
```

Exercise: Sum of First N Numbers

Calculate sum using formula: $\frac{(1+N)\times N}{2}$

```
public class Main {
    public static void main(String[] args) {
        int n = /*TODO*/;
        int sum = /*TODO*/;
        System.out.println(sum); // 5050
        System.out.println(sum == 5050 ?
            "Test passed" : "Test failed");
    }
}
```

Exercise: Sum of First N Numbers (Cont.)

Calculate sum using formula: $\frac{(1+N)\times N}{2}$

```java
public class Main {
    public static void main(String[] args) {
        int n = 100;
        int sum = (1 + n) * n / 2;
        System.out.println(sum); // 5050
        System.out.println(sum == 5050 ?
            "Test passed" : "Test failed");
    }
}
```

Summary

- Integer operations are precise but watch for overflow
- Use appropriate types (`int` vs `long`)
- Compound assignment operators simplify code
- Be careful with casting and type promotion
- Use parentheses for complex expressions
- Avoid `byte`/`short` for arithmetic unless necessary

Floating Point Characteristics

- Limited operations: +, -, *, /
- Cannot perform bitwise or shift operations
- Often cannot be represented precisely

```
// Precision issues
double x = 1.0 / 10;     // 0.1 (approximately)
double y = 1 - 9.0 / 10; // 0.1 (approximately)
System.out.println(x);   // 0.1
System.out.println(y);   // 0.09999999999999998
```

Comparing Floating Point Numbers

- Never use == for floating point comparison
- Compare absolute difference against threshold

```
1 double x = 1.0 / 10;
2 double y = 1 - 9.0 / 10;
3
4 double diff = Math.abs(x - y);
5 if (diff < 0.00001) {
6     System.out.println("Equal");
7 } else {
8     System.out.println("Not equal");
9 }
```

Type Promotion

- Integers automatically promoted to floating point
- Watch for integer division in mixed expressions

```
int n = 5;
double d1 = 1.2 + 24.0 / n; // 6.0
double d2 = 1.2 + 24 / n;   // 5.2

System.out.println(d1);
System.out.println(d2);
```

Special Values

- Division by zero returns special values
- No exception thrown

```
1 double d1 = 0.0 / 0;    // NaN
2 double d2 = 1.0 / 0;    // Infinity
3 double d3 = -1.0 / 0;   // -Infinity
4
5 System.out.println(d1); // NaN
6 System.out.println(d2); // Infinity
7 System.out.println(d3); // -Infinity
```

Explicit Casting

- Fractional part discarded during casting
- Values beyond range return max integer
- Add 0.5 for rounding

```
int n1 = (int) 12.3;    // 12
int n2 = (int) 12.7;    // 12
int n3 = (int) -12.7;   // -12
int n4 = (int) 1.2e20; // 2147483647

// Rounding
double d = 2.6;
int rounded = (int) (d + 0.5); // 3
```

Exercise: Quadratic Equation

Solve $ax^2 + bx + c = 0$ using quadratic formula:

```java
public class Main {
    public static void main(String[] args) {
        double a = 1.0, b = 3.0, c = -4.0;
        // TODO: Calculate roots
        double r1 = 0;
        double r2 = 0;

        System.out.println(r1); // 1.0
        System.out.println(r2); // -4.0
    }
}
```

Exercise: Quadratic Equation (Cont.)

Solve $ax^2 + bx + c = 0$ using quadratic formula:

```java
public class Main {
    public static void main(String[] args) {
        double a = 1.0, b = 3.0, c = -4.0;
        double discriminant = b * b - 4 * a * c;
        double r1 = (-b + Math.sqrt(discriminant)) / (2 * a);
        double r2 = (-b - Math.sqrt(discriminant)) / (2 * a);

        System.out.println(r1); // 1.0
        System.out.println(r2); // -4.0
    }
}
```

Summary

- Floating point numbers often cannot be represented precisely, and their arithmetic results may have errors.
- To compare two floating point numbers, compare the absolute value of their difference against a specific threshold.
- When integers and floating point numbers are involved in arithmetic, integers are automatically promoted to floating point numbers.
- Floating point numbers can be explicitly cast to integers, but values exceeding the range will always return the maximum integer value.

Boolean Operators

- Comparison: >, >=, <, <=, ==, !=
- Logical: && (AND), || (OR), ! (NOT)

```
int age = 12;
boolean isGreater = 5 > 3;          // true
boolean isZero = age == 0;          // false
boolean isNonZero = !isZero;        // true
boolean isAdult = age >= 18;        // false
boolean isTeen = age > 6 && age < 18; // true
```

Short-Circuit Evaluation

- Stops evaluating when result is determined
- Prevents unnecessary computations

```
boolean b = 5 < 3; // false
// Division by zero avoided due to short-circuit
boolean result = b && (5 / 0 > 0);
System.out.println(result); // false

// This would throw ArithmeticException
// boolean result2 = true && (5 / 0 > 0);
```

Ternary Operator

- Syntax: condition ? expr1 : expr2
- Returns expr1 if true, expr2 if false

```
int n = -100;
int x = n >= 0 ? n : -n; // Absolute value
System.out.println(x); // 100

// Types must match
String result = n > 0 ? "positive" : "negative";
```

Exercise: Primary Student Check

Check if age is 6-12 years:

```
public class Main {
    public static void main(String[] args) {
        int age = 7;
        boolean isPrimaryStudent = ???;
        System.out.println(isPrimaryStudent ? "Yes" : "No");
    }
}
```

Exercise: Primary Student Check (Cont.)

Check if age is 6-12 years:

```
public class Main {
    public static void main(String[] args) {
        int age = 7;
        boolean isPrimaryStudent = age >= 6 && age <= 12;
        System.out.println(isPrimaryStudent ? "Yes" : "No");
    }
}
```

Summary

- AND and OR operations are short-circuit operations.
- The ternary operation `b ? x : y` requires the types of `x` and `y` to be the same.
- The ternary operation is also a "short-circuit operation," evaluating only `x` or `y`.

Character Type (char)

- Primitive type, holds single Unicode character
- 2 bytes per character

```
char c1 = 'A';
char c2 = '\u4e2d'; // Chinese character

int n1 = 'A';  // 65
int n2 = '\u4e2d'; // 20013

System.out.println(c1); // A
System.out.println(n2); // 20013
```

String Type (String)

- Reference type, uses double quotes
- Escape sequences for special characters

```
String s1 = ""; // Empty string
String s2 = "A"; // 1 character
String s3 = "ABC"; // 3 characters

// Escape sequences
String s4 = "abc\"xyz"; // Contains: a, b, c, ", x, y, z
String s5 = "abc\\xyz"; // Contains: a, b, c, \, x, y, z
String s6 = "Line1\nLine2"; // Newline
```

String Concatenation

- + operator automatically converts types
- Other types converted to strings

```
String s1 = "Hello";
String s2 = "world";
String s = s1 + " " + s2 + "!"; // Hello world!

int age = 25;
String info = "Age: " + age; // Age: 25

double pi = 3.14159;
String msg = "PI = " + pi; // PI = 3.14159
```

Multi-line Strings (Text Blocks)

- Java 13+ feature using """..."""
- Common leading spaces removed

```
String query = """
               SELECT * FROM users
               WHERE age > 18
               ORDER BY name
               """;

// Equivalent to:
String oldStyle = "SELECT * FROM users\n" +
                  "WHERE age > 18\n" +
                  "ORDER BY name";
```

String Immutability

- String content cannot be changed
- Variables can reference different strings

```
String s = "hello";
String t = s;        // t references "hello"
s = "world";         // s now references "world"

System.out.println(t); // hello (unchanged)
System.out.println(s); // world
```

Null vs Empty String

- null: no object reference
- "": empty string object

```
String s1 = null;     // No string object
String s2 = "";       // Empty string object
String s3 = s1;       // Also null

System.out.println(s1 == null); // true
System.out.println(s2.isEmpty()); // true
// System.out.println(s1.isEmpty()); // NullPointerException
```

Exercise: Unicode to String

Convert Unicode codes to string:

```java
public class Main {
    public static void main(String[] args) {
        int a = 72;     // 'H'
        int b = 105;    // 'i'
        int c = 65281; // '!'
        // FIXME:
        String s = a + b + c;
        System.out.println(s); // Hi!
    }
}
```

Exercise: Unicode to String (Cont.)

Convert Unicode codes to string:

```java
public class Main {
    public static void main(String[] args) {
        int a = 72;     // 'H'
        int b = 105;    // 'i'
        int c = 65281; // '!'

        String s = "" + (char)a + (char)b + (char)c;
        System.out.println(s); // Hi!
    }
}
```

Summary

- Java's character type ‘char‘ is a primitive type, while the string type ‘String‘ is a reference type.
- Primitive type variables "hold" a value, while reference type variables "point to" an object.
- Reference type variables can be null.
- Distinguish between the null value ‘null‘ and the empty string ‘""‘.