Chapter 5: Getting Started with Java

Computer Fundamentals and Introduction to Programming

Programs | Tools | First program | Compiling | Errors

Sercan KÜLCÜ · Giresun University

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/computer/slides/Chapter_05_Getting_Started_with_Java.html

Learning Outcomes

Programs

Describe a program as data plus operations.

Java

Relate source code, bytecode and the JVM.

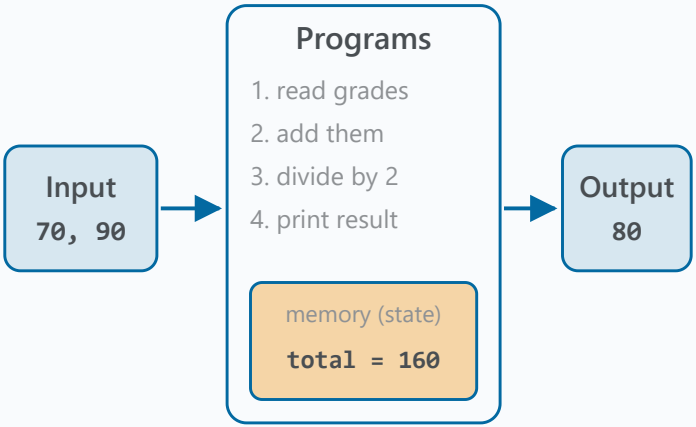
First program

Write, compile and run a Java file.

Errors

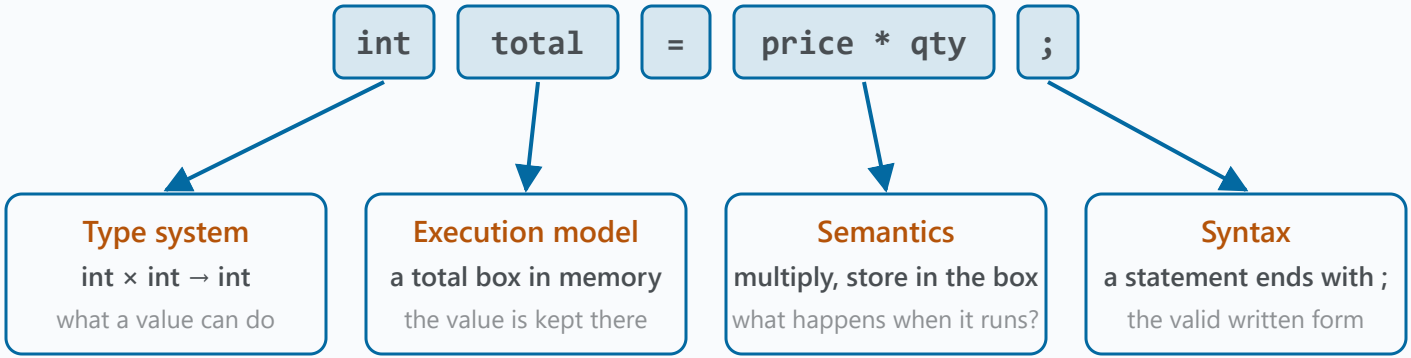
Tell syntax, runtime and logic errors apart.

What Is a Program?



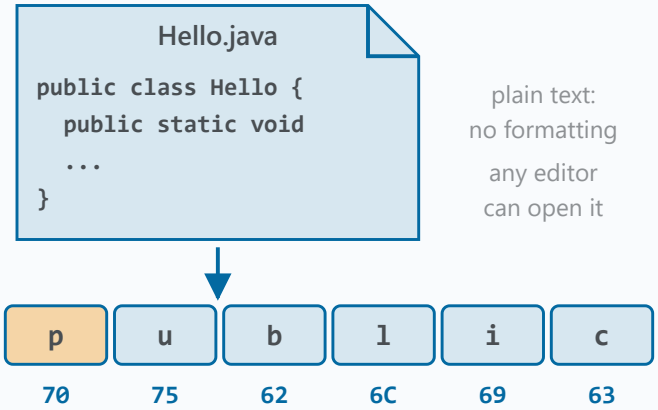
- A set of instructions the computer executes to carry out a task.
- It takes input, processes data, keeps **state** (values in memory) and produces output.
- Its behavior is set by the language's rules and the algorithm.
- Example: the average of two grades → input 70, 90; output 80.

Aspects of a Programming Language



- **Syntax** defines the valid form; **semantics** defines what happens when it runs.
- The **type system** restricts operations; the **execution model** defines how memory and instructions behave.
- The compiler checks syntax and types before the program runs.

A Plain-Text Source File



ASCII characters take 1 byte in UTF-8 (hex); letters like ç, ğ, ş take 2

- A .java file is plain text; Notepad, VS Code or an IDE can open it.
- Version control systems can track line-by-line changes.
- Character encoding (UTF-8), line endings and the file name must be consistent: non-ASCII characters break under the wrong encoding.

Programming Tools

Editor
Writing source code.

IDE
Projects, completion, compiling and debugging.

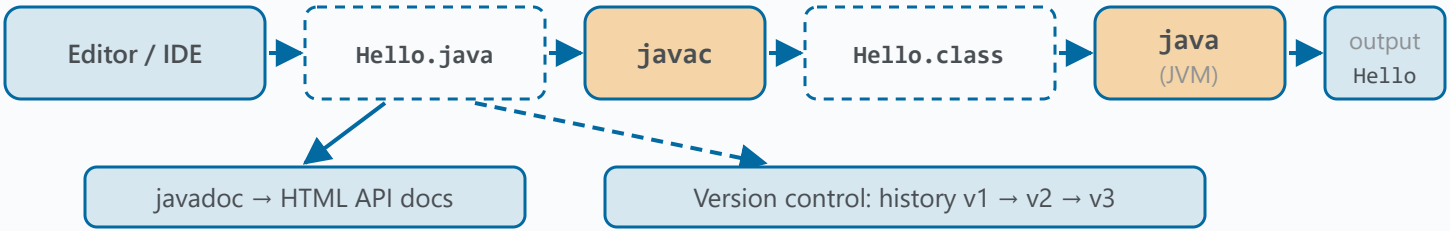
javac
Compiling source code to bytecode.

java
Running a class on the JVM.

Javadoc
Generating API documentation.

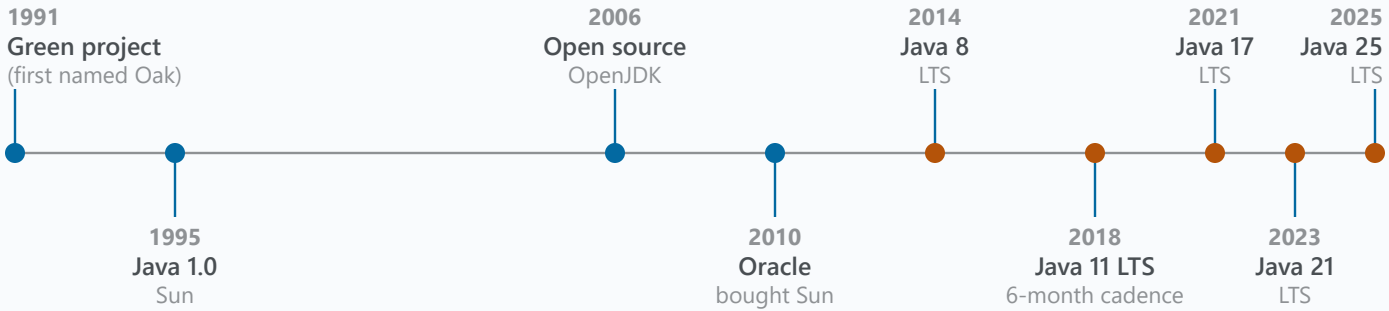
Version control
Change history.

The tools along a Java program's journey



JAVA

The Birth of Java and Its Versions



- James Gosling and his team began the work at Sun Microsystems in 1991; the language was released as Java in 1995. Oracle has developed it since 2010, and the open-source code lives in OpenJDK.
- The design goal: **write once, run anywhere**. The same bytecode runs on every platform's JVM.
- Other features: object-oriented, strong type checking, automatic memory management (garbage collection) and a large class library.
- Since 2018 a new release appears every six months; **LTS** (long-term support) releases get updates for years, so companies and courses prefer them.

Where Is Java Used?

Enterprise servers

The back end of banking, e-commerce and government systems; the Spring and Jakarta EE frameworks.

Android

Apps are written in Java or Kotlin; the bytecode is converted into a form for the phone's Android Runtime (ART).

Big data

Data processing tools such as Hadoop and Kafka run on the JVM.

Developer tools

IDEs such as IntelliJ IDEA, Eclipse and NetBeans are themselves written in Java.

Smart cards

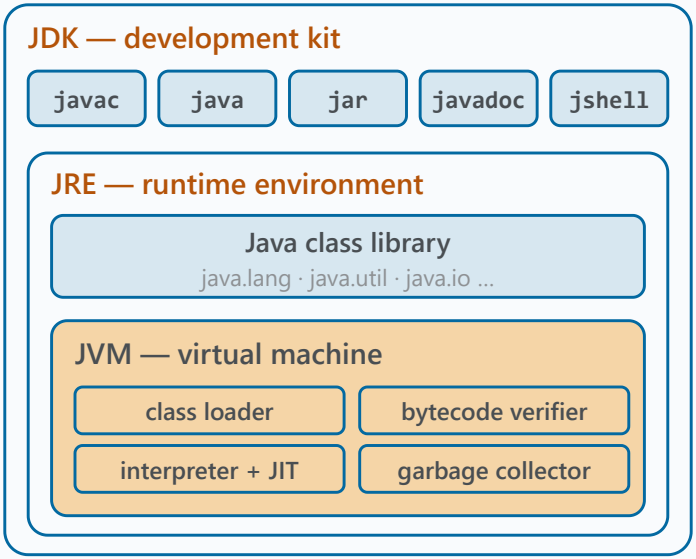
Small programs on SIM and bank cards run with Java Card.

Games

Minecraft: Java Edition; desktop games and simulations.

The JVM runs not only Java but other languages that compile to the same bytecode, such as Kotlin, Scala and Groovy.

JDK, JRE and JVM



- The **JVM** (Java Virtual Machine) is the virtual machine that executes bytecode.
- **JRE** = JVM + class library: what you need to run a program.
- **JDK** = JRE + development tools (javac, jar, javadoc…). Developers install the JDK.
- Since Java 11, Oracle ships no separate JRE; even someone who only runs programs installs a JDK (some vendors still offer a JRE).

JAVA

The First Java Program

```
public class Hello {

    public static void main(String[] args) {

        System.out.println("Hello");

    }

}
```

class declaration
name = file name: Hello.java

main method
the program's entry point

statement
print to the console; ends with ;

} end of main block

} end of class block

public
accessible from anywhere

static
called without creating an object

void
returns no value

String[] args
command-line arguments

JAVA

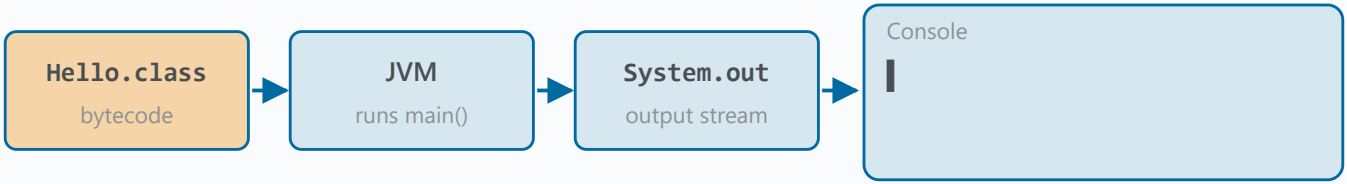
The Signature of `main`

Written (all compile)	Run with `java A`
`public static void main(String[] args)`	**runs**
`public static void Main(String[] args)`	`Error: Main method not found in class A, …`
`public void main(String[] args)`	`Error: Main method is not static in class A, …`
`public static int main(String[] args)`	`Error: Main method must return a value of type void in class A, …`
`static void main(String[] args)`	`Error: Main method not found in class A, …`

- To the compiler, each is an ordinary method, so the error shows up **at launch, not at compile time**. The JVM looks for the entry point by name, parameter and return type.
- The parameter name is free (`String[] a`); `String args[]` is also valid. `main` is lowercase: Java is case-sensitive.
- The messages come from JDK 21. From Java 25 on, a `main` that is not public or not static can also be launched; this course uses the classic signature.

JAVA · RUN THE CODE STEP BY STEP

Code Trace: `Hello`



The command `java Hello` starts the JVM, which loads the **Hello** class (Hello.class) into memory.

Step 1 / 6

```
▶ public class Hello {
2     public static void main(String[] args) {
3         System.out.println("Hello");
4     }
5 }
```

VARIABLES

—

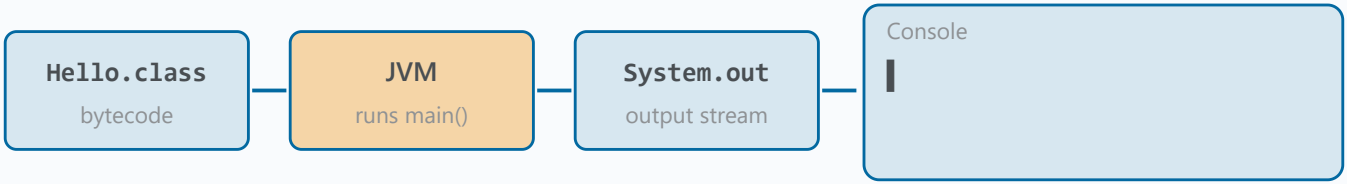
CALL STACK

empty

OUTPUT

JAVA · RUN THE CODE STEP BY STEP

Code Trace: `Hello`



The JVM calls the **main** method: a main frame is pushed on the stack. No command-line arguments were given, so args is empty.

Step 2 / 6

```
1 public class Hello {
▶     public static void main(String[] args) {
3         System.out.println("Hello");
4     }
5 }
```

VARIABLES

`args = empty (no arguments)`

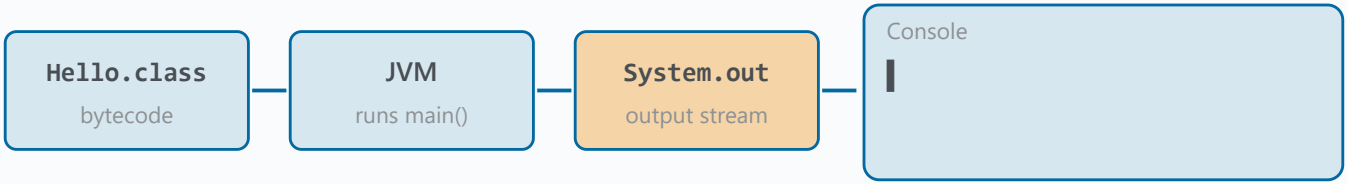
CALL STACK

`main(args)`

OUTPUT

JAVA · RUN THE CODE STEP BY STEP

Code Trace: `Hello`



The statement runs: the text `"Hello"` is passed to `System.out.println`.

Step 3 / 6

```
1 public class Hello {
2     public static void main(String[] args) {
▶         System.out.println("Hello");
4     }
5 }
```

VARIABLES

`args = empty (no arguments)`

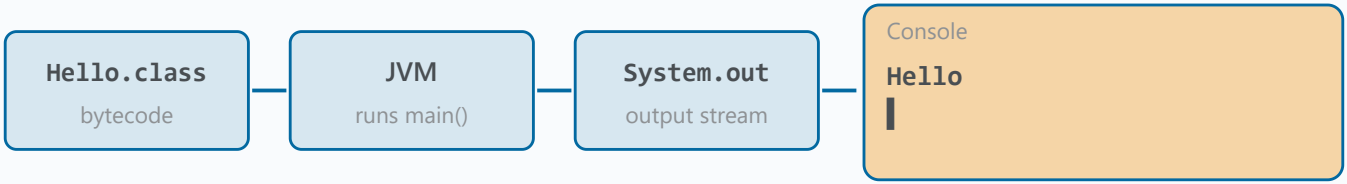
CALL STACK

`println("Hello")`

`main(args)`

OUTPUT

Code Trace: `Hello`



println prints the text and adds a **line break**: the cursor moves down a line. The method ends and control returns to main.

Step 4 / 6

```
1 public class Hello {
2     public static void main(String[] args) {
▶         System.out.println("Hello");
4     }
5 }
```

VARIABLES

`args = empty (no arguments)`

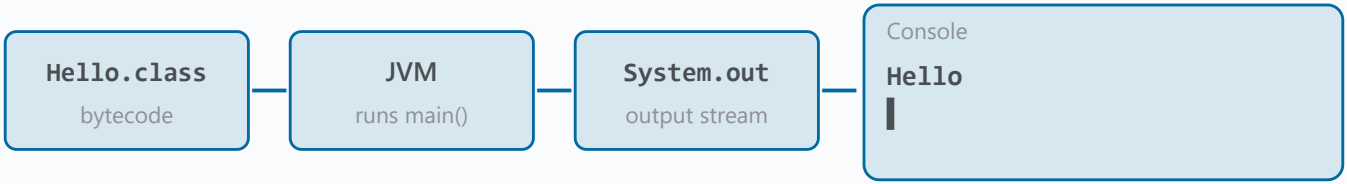
CALL STACK

`main(args)`

OUTPUT

```
Hello
```

Code Trace: `Hello`



We reach the `}` of the main block: main ends and its frame is popped.

Step 5 / 6

```
1 public class Hello {
2     public static void main(String[] args) {
3         System.out.println("Hello");
▶     }
5 }
```

VARIABLES

`args = empty (no arguments)`

CALL STACK

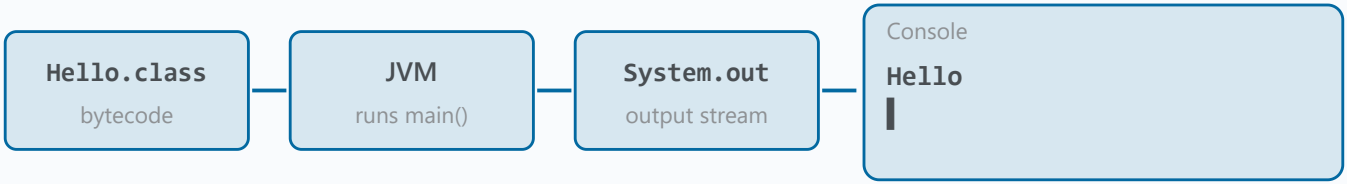
empty

OUTPUT

```
Hello
```

JAVA · RUN THE CODE STEP BY STEP

Code Trace: `Hello`



No code is left to run: the JVM shuts down and the program **terminates**. One statement ran and printed one line.

Step 6 / 6

```
1 public class Hello {
2     public static void main(String[] args) {
3         System.out.println("Hello");
4     }
▶ }
```

VARIABLES

—

CALL STACK

empty

OUTPUT

```
Hello
```

Code Trace: Command-Line Arguments

```
> java Greet Ali Veli
```



Words written after the class name are **command-line arguments**. The JVM puts them in a String array and passes it to **main**.

Step 1 / 8

```
1 public class Greet {
▶     public static void main(String[] args) {
3         System.out.println(args.length);
4         System.out.println("Hi " + args[0]);
5         System.out.println("Hi " + args[1]);
6     }
7 }
```

VARIABLES

`args = ["Ali", "Veli"]`

OUTPUT

Code Trace: Command-Line Arguments

```
> java Greet Ali Veli
```



args.length is the number of arguments: 2.

Step 2 / 8

```
1 public class Greet {
2     public static void main(String[] args) {
▶         System.out.println(args.length);
4         System.out.println("Hi " + args[0]);
5         System.out.println("Hi " + args[1]);
6     }
7 }
```

VARIABLES

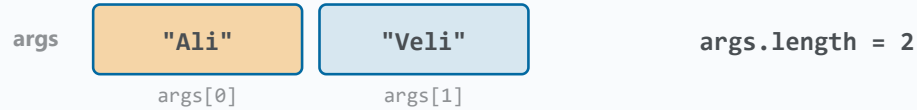
`args = ["Ali", "Veli"]`

OUTPUT

```
2
```

Code Trace: Command-Line Arguments

```
> java Greet Ali Veli
```



The first argument is in **args[0]** (indices start at 0): Hi Ali.

Step 3 / 8

```
1 public class Greet {
2     public static void main(String[] args) {
3         System.out.println(args.length);
▶         System.out.println("Hi " + args[0]);
5         System.out.println("Hi " + args[1]);
6     }
7 }
```

VARIABLES

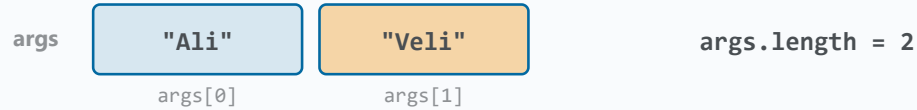
`args = ["Ali", "Veli"]`

OUTPUT

```
2
Hi Ali
```

Code Trace: Command-Line Arguments

```
> java Greet Ali Veli
```



args[1] → Veli. main ends and the program terminates.

Step 4 / 8

```
1 public class Greet {
2     public static void main(String[] args) {
3         System.out.println(args.length);
4         System.out.println("Hi " + args[0]);
▶         System.out.println("Hi " + args[1]);
6     }
7 }
```

VARIABLES

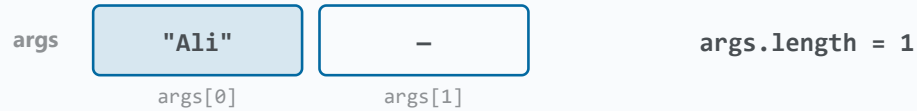
`args = ["Ali", "Veli"]`

OUTPUT

```
2
Hi Ali
Hi Veli
```

Code Trace: Command-Line Arguments

```
> java Greet Ali
```



Second run: the same .class, this time with one argument. The code did not change, so no recompiling is needed.

Step 5 / 8

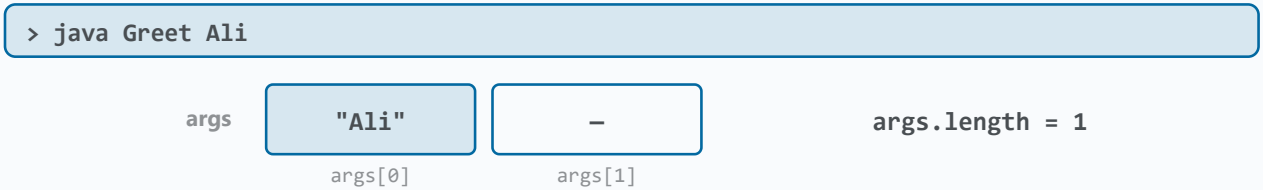
```
1 public class Greet {
▶     public static void main(String[] args) {
3         System.out.println(args.length);
4         System.out.println("Hi " + args[0]);
5         System.out.println("Hi " + args[1]);
6     }
7 }
```

VARIABLES

`args = ["Ali"]`

OUTPUT

Code Trace: Command-Line Arguments



args.length = 1.

Step 6 / 8

```
1 public class Greet {
2     public static void main(String[] args) {
▶         System.out.println(args.length);
4         System.out.println("Hi " + args[0]);
5         System.out.println("Hi " + args[1]);
6     }
7 }
```

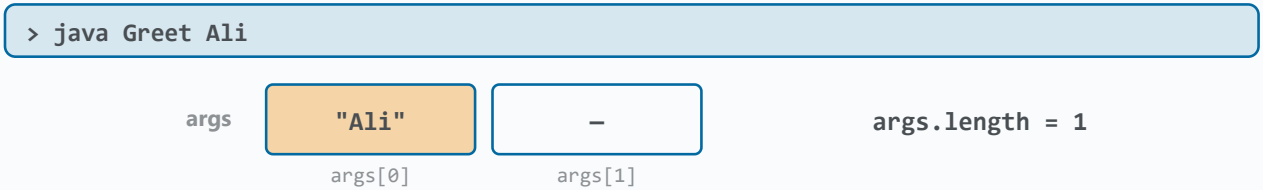
VARIABLES

`args = ["Ali"]`

OUTPUT

```
1
```

Code Trace: Command-Line Arguments



args[0] exists: Hi Ali.

Step 7 / 8

```
1 public class Greet {
2     public static void main(String[] args) {
3         System.out.println(args.length);
▶         System.out.println("Hi " + args[0]);
5         System.out.println("Hi " + args[1]);
6     }
7 }
```

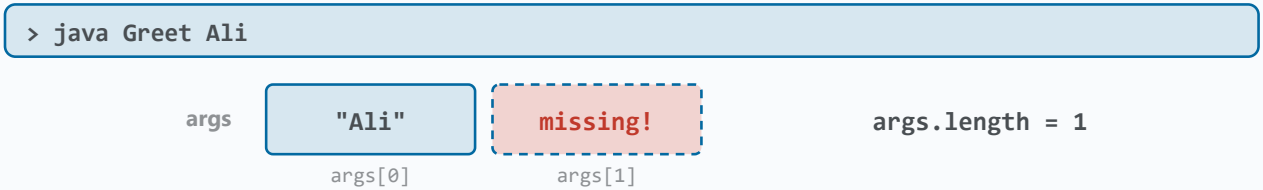
VARIABLES

`args = ["Ali"]`

OUTPUT

```
1
Hi Ali
```

Code Trace: Command-Line Arguments



There is no args[1]: the JVM throws **ArrayIndexOutOfBoundsException** and the program stops. The compiler could not have known; the error arose at run time and depends on the input.

Step 8 / 8

```
1 public class Greet {
2     public static void main(String[] args) {
3         System.out.println(args.length);
4         System.out.println("Hi " + args[0]);
▶         System.out.println("Hi " + args[1]);
6     }
7 }
```

VARIABLES

`args = ["Ali"]`

OUTPUT

```
1
Hi Ali
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
: Index 1 out of bounds for length 1
    at Greet.main(Greet.java:5)
```

JAVA · RUN THE CODE STEP BY STEP

Code Trace: `print` and `println`

Console

cursor: line 1, col 1

The JVM calls **main**; the console is still empty and the cursor sits at the start of line 1.

Step 1 / 5

```
1 public class Greeting {
▶     public static void main(String[] args) {
3         System.out.print("Hello ");
4         System.out.println("World");
5         System.out.println("Java");
6     }
7 }
```

CALL STACK

`main(args)`

OUTPUT

Code Trace: `print` and `println`

Console

```
Hello
```

cursor: line 1, col 7

print writes "Hello " but **adds no line break**: the cursor stays on the same line, right after the space.

Step 2 / 5

```
1 public class Greeting {
2     public static void main(String[] args) {
▶         System.out.print("Hello ");
4         System.out.println("World");
5         System.out.println("Java");
6     }
7 }
```

CALL STACK

`main(args)`

OUTPUT

```
Hello
```

Code Trace: `print` and `println`

Console

```
Hello World
▌
```

cursor: ▌
line 2, col 1

println writes "World" and adds a **line break**: the next output starts on the line below.

Step 3 / 5

```
1 public class Greeting {
2     public static void main(String[] args) {
3         System.out.print("Hello ");
▶         System.out.println("World");
5         System.out.println("Java");
6     }
7 }
```

CALL STACK

`main(args)`

OUTPUT

```
Hello World
```

Code Trace: `print` and `println`

Console

```
Hello World
Java
```

cursor: line 3, col 1

println writes "Java" and adds a **line break**: the next output starts on the line below.

Step 4 / 5

```
1 public class Greeting {
2     public static void main(String[] args) {
3         System.out.print("Hello ");
4         System.out.println("World");
▶         System.out.println("Java");
6     }
7 }
```

CALL STACK

`main(args)`

OUTPUT

```
Hello World
Java
```

JAVA · RUN THE CODE STEP BY STEP

Code Trace: `print` and `println`

Console

```
Hello World
Java
```

cursor: line 3, col 1

main ends; the console has **2 lines**: there were 3 calls, but only println ends a line.

Step 5 / 5

```
1 public class Greeting {
2     public static void main(String[] args) {
3         System.out.print("Hello ");
4         System.out.println("World");
5         System.out.println("Java");
▶     }
7 }
```

CALL STACK

empty

OUTPUT

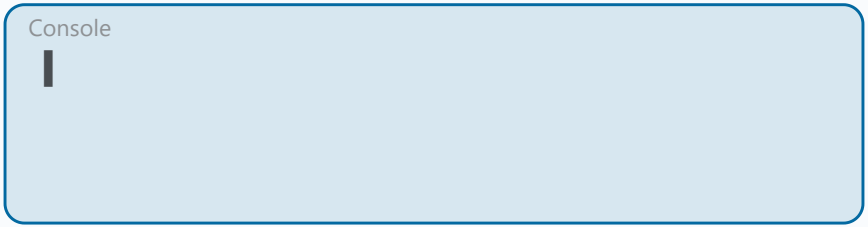
```
Hello World
Java
```

Escape Sequences

Sequence	Meaning	Example	Output
`\n`	new line	`"A\nB"`	A, then B on the next line
`\t`	tab	`"Name\tGrade"`	Name Grade
`\"`	double quote	`"\"Java\""`	"Java"
`\\`	backslash	`"C:\\docs"`	C:\docs
`\'`	single quote (inside a char)	`'\''`	'

- Characters that cannot be typed in a string, or that have a special meaning, are written as sequences starting with `\`.
- A `"` inside the quotes ends the text; if it belongs to the text, write `\"`.

Code Trace: Escape Sequences



main starts; the console is empty.

Step 1 / 5

```
1 public class Escape {
▶     public static void main(String[] args) {
3         System.out.println("Name\tGrade");
4         System.out.println("Ali\t85");
5         System.out.println("\"Java\" rocks");
6         System.out.print("Line 1\nLine 2\n");
7     }
8 }
```

CALL STACK

```
main(args)
```

OUTPUT

Code Trace: Escape Sequences

Console

```
Name    Grade
```

`\t` moves the cursor to the next **tab stop** (column 9).

Step 2 / 5

```
1 public class Escape {
2     public static void main(String[] args) {
▶         System.out.println("Name\tGrade");
4         System.out.println("Ali\t85");
5         System.out.println("\"Java\" rocks");
6         System.out.print("Line 1\nLine 2\n");
7     }
8 }
```

CALL STACK

`main(args)`

OUTPUT

```
Name    Grade
```

Code Trace: Escape Sequences

Console

```
Name    Grade
Ali     85
```

Thanks to the tab, 85 lines up under the Grade heading: a simple table.

Step 3 / 5

```
1 public class Escape {
2     public static void main(String[] args) {
3         System.out.println("Name\tGrade");
▶         System.out.println("Ali\t85");
5         System.out.println("\"Java\" rocks");
6         System.out.print("Line 1\nLine 2\n");
7     }
8 }
```

CALL STACK

```
main(args)
```

OUTPUT

```
Name    Grade
Ali     85
```

Code Trace: Escape Sequences

Console

```
Name    Grade
Ali     85
"Java" rocks
```

`\"` writes a double quote without ending the text.

Step 4 / 5

```
1 public class Escape {
2     public static void main(String[] args) {
3         System.out.println("Name\tGrade");
4         System.out.println("Ali\t85");
▶         System.out.println("\"Java\" rocks");
6         System.out.print("Line 1\nLine 2\n");
7     }
8 }
```

CALL STACK

`main(args)`

OUTPUT

```
Name    Grade
Ali     85
"Java" rocks
```

Code Trace: Escape Sequences

Console

```
"Java" rocks
Line 1
Line 2
```

One print produced two lines: `\n` is the line-break character; println adds a line break too (`\r\n` on Windows).

Step 5 / 5

```
1 public class Escape {
2     public static void main(String[] args) {
3         System.out.println("Name\tGrade");
4         System.out.println("Ali\t85");
5         System.out.println("\"Java\" rocks");
▶         System.out.print("Line 1\nLine 2\n");
7     }
8 }
```

CALL STACK

```
main(args)
```

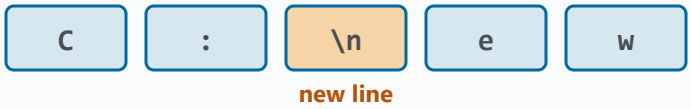
OUTPUT

```
Name    Grade
Ali     85
"Java" rocks
Line 1
Line 2
```

JAVA

Escape Sequence Pitfalls

`in source: "C:\new"`



on screen:
`C:`
`ew`

Statement	Result
`System.out.println("C:\data");`	compile error: `illegal escape character`; there is no `\d` escape sequence
`System.out.println("C:\new");`	compiles but prints two lines: `C:` and `ew`
`System.out.println("C:\\data");`	`C:\data`: the backslash is written twice
`System.out.println("Ali's grade");`	`Ali's grade`: inside a string, a single quote needs no escape
`System.out.println("Caf\u00E9");`	`Café`: `\u` plus four hex digits gives a Unicode character

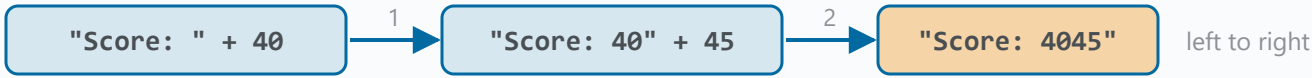
The most common mistake in Windows file paths is a single backslash; Java file operations also accept the form `"C:/data"`.

JAVA

String Concatenation: `+` Does Two Jobs

- Between two numbers, `+` **adds**; if either operand is a String, it **concatenates**: the number is converted to text and appended.
- Operators of equal precedence go left to right; `*` and `/` always come before `+`.

```
System.out.println("Score: " + 40 + 45);
```



Expression	Output	Why?
`40 + 45 + " points"`	`85 points`	the two ints are added first, then the text is appended
`"Score: " + 40 + 45`	`Score: 4045`	the leftmost is a String; both numbers are appended as text
`"Score: " + (40 + 45)`	`Score: 85`	the parentheses are evaluated first
`"Avg: " + 85 / 2`	`Avg: 42`	`/` first; dividing two ints drops the fraction

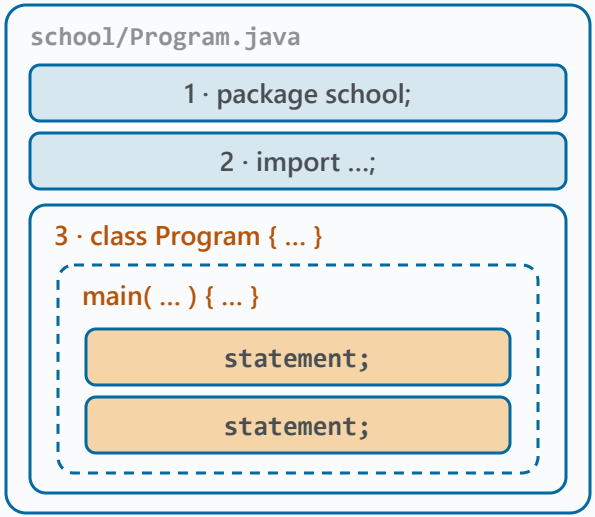
SOURCE CODE

Structure of a Java File

```java
package school;

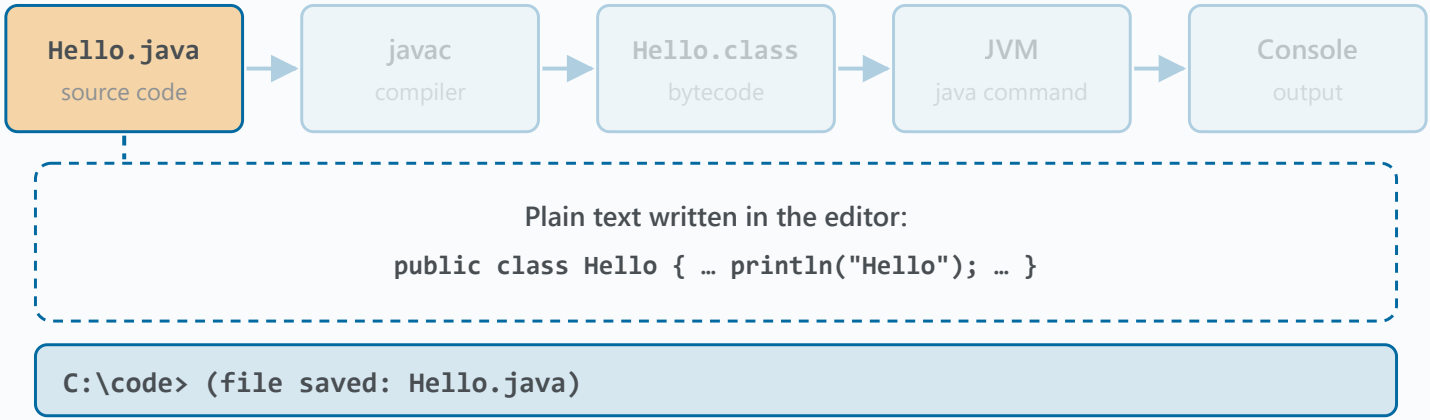
import java.util.Scanner;

public class Program {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```



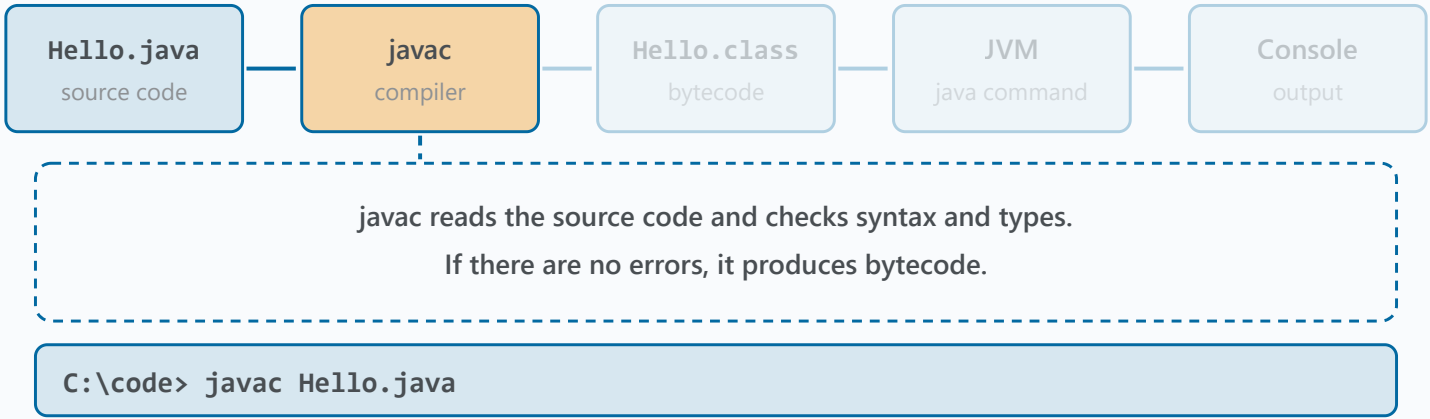
- package and import declarations come before the class; the public class name matches the file name.
- `package school;` → the file lives in the `school/` folder; run it with `java school.Program`.

The Build Pipeline Step by Step



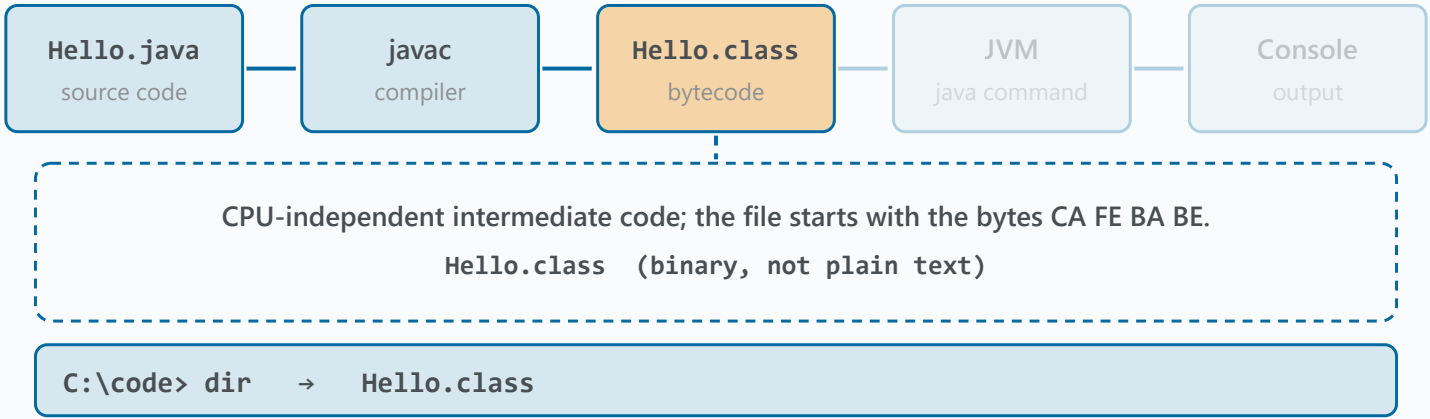
1. Write: save the source code as `Hello.java`. The file name must match the public class name.

The Build Pipeline Step by Step



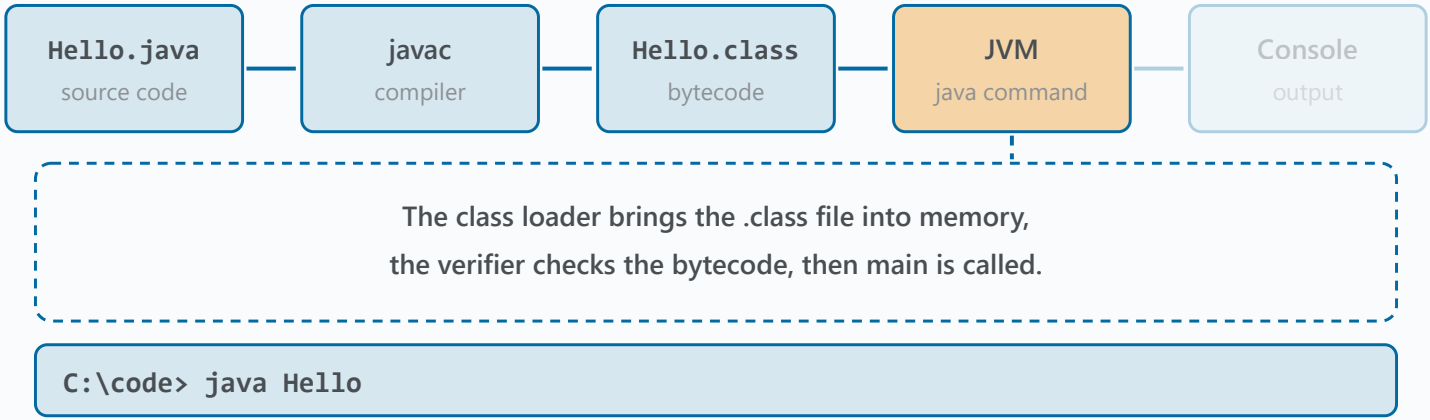
2. Compile: `javac` (the compiler in the JDK) checks the whole file before anything runs.

The Build Pipeline Step by Step



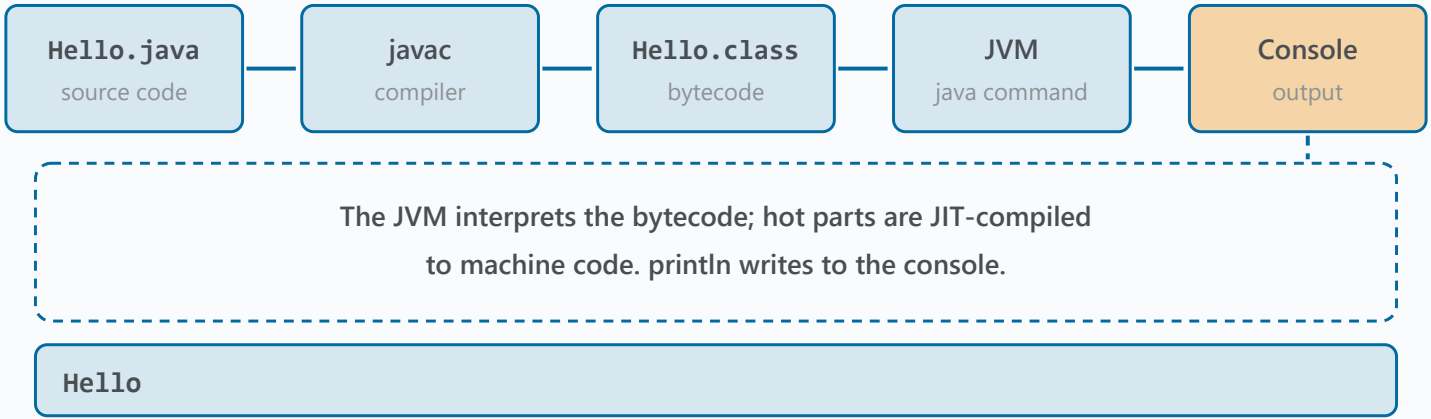
3. Bytecode: if compilation succeeds, `Hello.class` appears in the same folder. This file is identical on Windows, Linux and macOS.

The Build Pipeline Step by Step



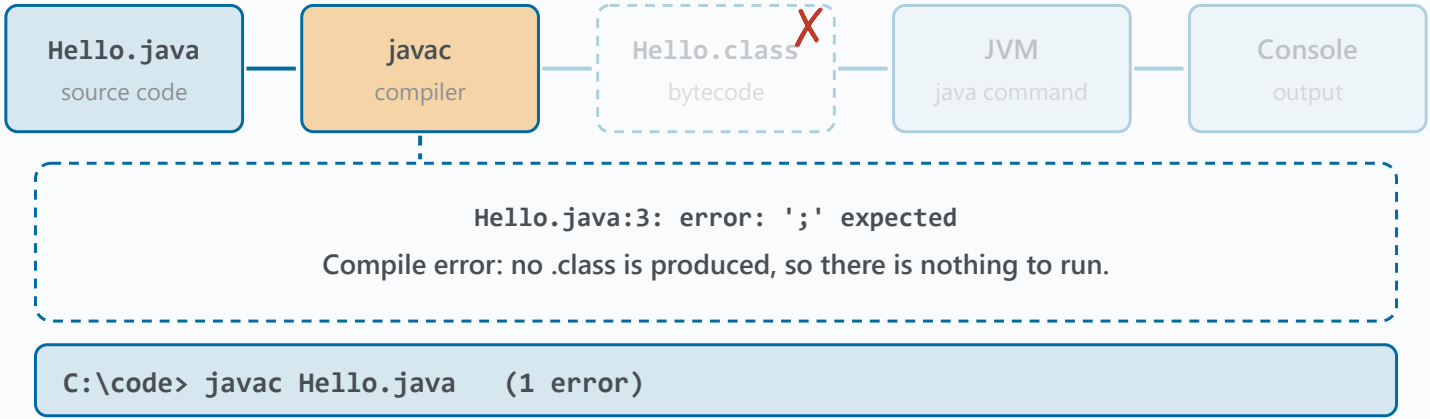
4. Run: `java Hello` starts the JVM. Give the class name; **do not write the .class extension**.

The Build Pipeline Step by Step



5. Output: `Hello` appears on the console. After any change, you must **compile** again.

The Build Pipeline Step by Step



If there is an error: javac reports it with a line number and **produces no .class**. Fix compile errors first.

Installing the JDK and Commands

Command	What does it do?
`java -version`	shows the installed Java version
`javac -version`	shows the compiler version (**javac** comes only with the JDK)
`javac Hello.java`	compiles: produces `Hello.class`
`java Hello`	runs it (no extension)
`java Hello.java`	Java 11+: compile and run in one step, leaving no .class

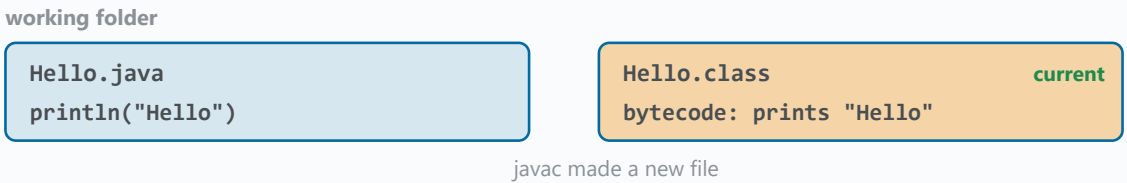
PATH and JAVA_HOME

If the commands are not found, add the JDK's `bin` folder to the PATH environment variable. `JAVA_HOME` points to the JDK folder; tools rely on it.

Non-ASCII text on the Windows console

Compile with `javac -encoding UTF-8 Hello.java`. If letters come out garbled, run `chcp 65001` in the console first.

Console Session: Compile, Edit, Recompile



When compilation succeeds, javac **prints nothing**; Hello.class quietly appears.

Step 1 / 7

```
▶ > javac Hello.java
2 > java Hello
3   (in the editor "Hello" → "Hi", save)
4 > java Hello
5 > javac Hello.java
6 > java Hello
7 > java Hello.class
```

VARIABLES

`Hello.java = Hello`

`Hello.class = Hello`

OUTPUT

```
> javac Hello.java
```

Console Session: Compile, Edit, Recompile

working folder

```
Hello.java
println("Hello")
```

```
Hello.class                    current
bytecode: prints "Hello"
```

The JVM loads Hello.class and runs it.

Step 2 / 7

```
1 > javac Hello.java
▶ > java Hello
3   (in the editor "Hello" → "Hi", save)
4 > java Hello
5 > javac Hello.java
6 > java Hello
7 > java Hello.class
```

VARIABLES

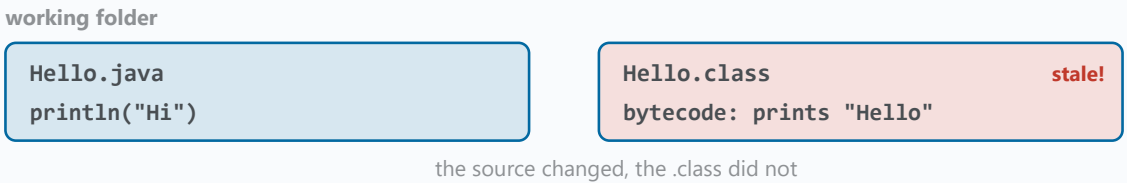
`Hello.java = Hello`

`Hello.class = Hello`

OUTPUT

```
> java Hello
Hello
```

Console Session: Compile, Edit, Recompile



The source was edited and saved, but the .class file does **not** update itself.

Step 3 / 7

```
1 > javac Hello.java
2 > java Hello
  (in the editor "Hello" → "Hi", save)
4 > java Hello
5 > javac Hello.java
6 > java Hello
7 > java Hello.class
```

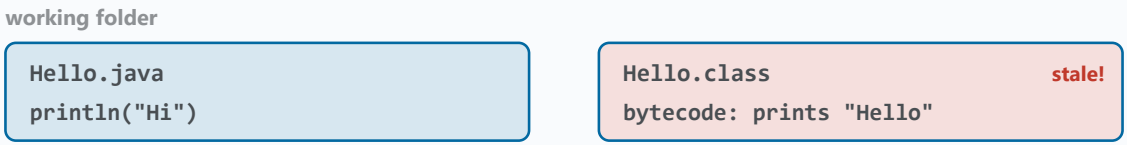
VARIABLES

`Hello.java = Hi`

`Hello.class = Hello (old)`

OUTPUT

Console Session: Compile, Edit, Recompile



The output is still **Hello**: the java command ignores the source and runs only the bytecode in the .class.


```
1 > javac Hello.java
2 > java Hello
3   (in the editor "Hello" → "Hi", save)
▶ > java Hello
5 > javac Hello.java
6 > java Hello
7 > java Hello.class
```

VARIABLES

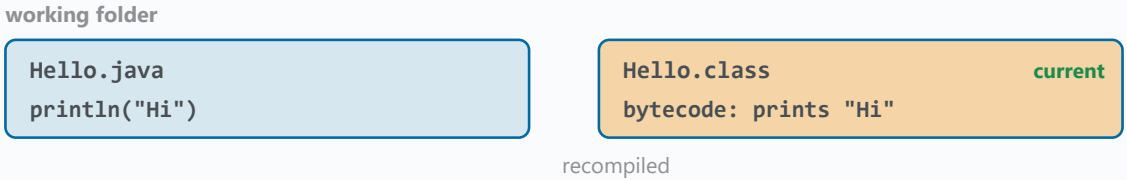
`Hello.java = Hi`

`Hello.class = Hello (old)`

OUTPUT

```
> java Hello
Hello
```

Console Session: Compile, Edit, Recompile



Recompiling refreshes the .class file.

Step 5 / 7

```
1 > javac Hello.java
2 > java Hello
3   (in the editor "Hello" → "Hi", save)
4 > java Hello
▶ > javac Hello.java
6 > java Hello
7 > java Hello.class
```

VARIABLES

`Hello.java = Hi` `Hello.class = Hi`

OUTPUT

```
> javac Hello.java
```

Console Session: Compile, Edit, Recompile

working folder

```
Hello.java
println("Hi")
```

```
Hello.class                    current
bytecode: prints "Hi"
```

The change is now visible: **Hi**.

Step 6 / 7

```
1 > javac Hello.java
2 > java Hello
3   (in the editor "Hello" → "Hi", save)
4 > java Hello
5 > javac Hello.java
▶ > java Hello
7 > java Hello.class
```

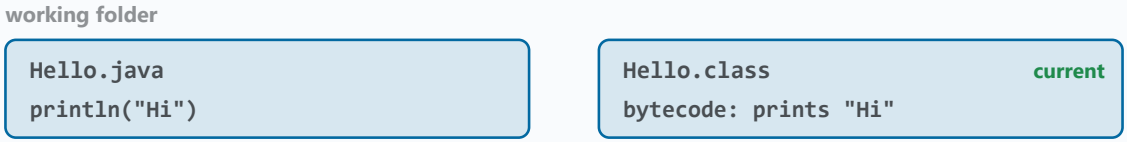
VARIABLES

`Hello.java = Hi` `Hello.class = Hi`

OUTPUT

```
> java Hello
Hi
```

Console Session: Compile, Edit, Recompile



java expects a class name, not a file name

java was given a **file name**: the JVM looks for a class named "Hello.class" and cannot find it. Leave out the extension.

Step 7 / 7

```
1 > javac Hello.java
2 > java Hello
3   (in the editor "Hello" → "Hi", save)
4 > java Hello
5 > javac Hello.java
6 > java Hello
▶ > java Hello.class
```

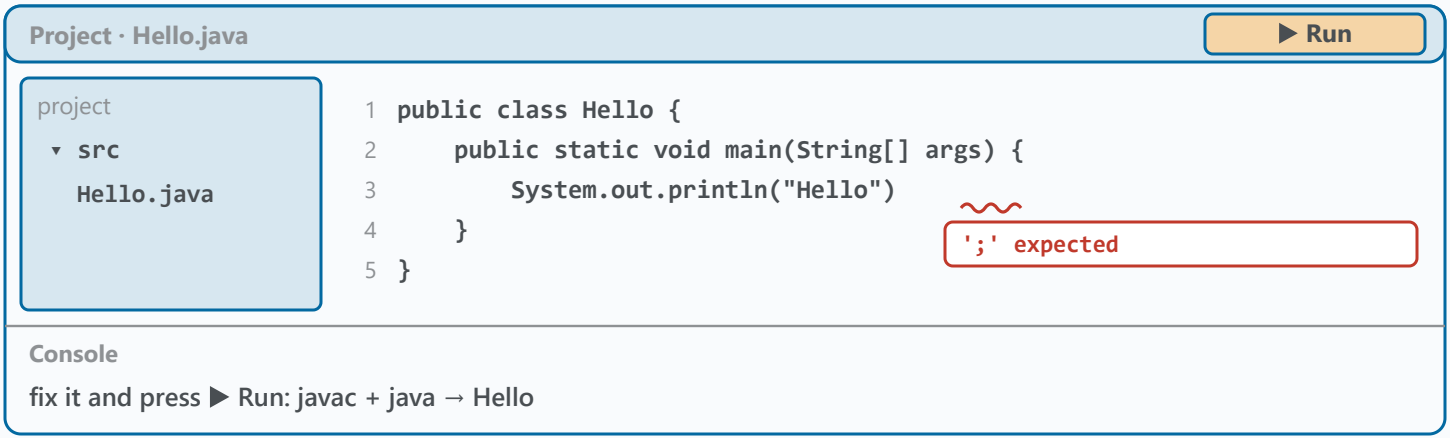
VARIABLES

`Hello.java = Hi` `Hello.class = Hi`

OUTPUT

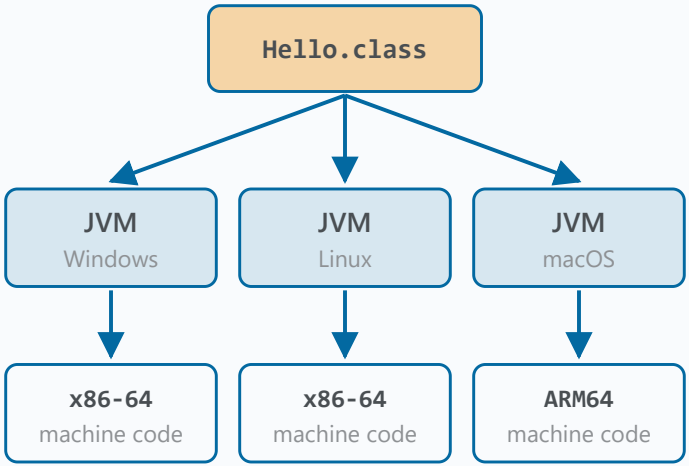
```
> java Hello.class
Error: Could not find or load main
class Hello.class
Caused by:
java.lang.ClassNotFoundException:
Hello.class
```

Working in an IDE



- The ▶ **Run** button does the compile and run steps behind the scenes; a changed file is recompiled first, so there is no stale .class problem.
- Compile errors are underlined in red as you type; hovering shows the compiler's message.
- IDEs offer code completion, auto-indentation, renaming and a debugger with breakpoints. Common Java IDEs: IntelliJ IDEA, Eclipse, NetBeans, and VS Code with Java extensions.
- An IDE makes life easier, but knowing what happens behind the scenes lets you understand error messages where there is no IDE (servers, remote sessions).

Bytecode and the JVM



compile once, run wherever there is a JVM

- The compiler produces **bytecode** instead of CPU-specific machine code.
- Each platform's own JVM interprets the bytecode and JIT-compiles hot parts into that CPU's machine code.
- Class loading, verification and memory management (garbage collection) are part of the runtime.

Inside Bytecode: `javap`

```
> javap -c Hello
...
public static void main(java.lang.String[]);
  Code:
     0: getstatic     // System.out
     3: ldc           // "Hello"
     5: invokevirtual // println
     8: return
```

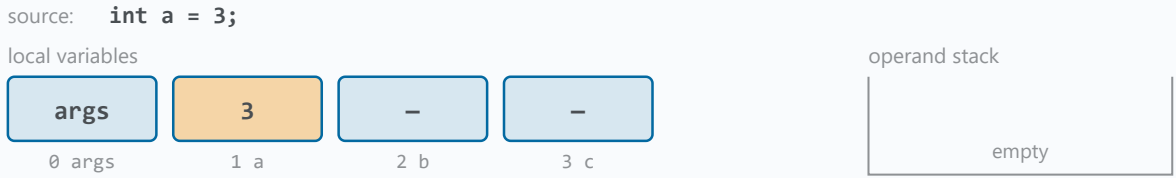
javap also prints constant-pool references; they are trimmed here.

- `javap -c` ships with the JDK; it lists the instructions in a .class file in readable form.
- **getstatic** pushes the System.out object and **ldc** pushes the constant "Hello" onto the **operand stack**.
- **invokevirtual** pops both and calls println; **return** ends main.
- Each instruction's opcode is one byte, hence the name "bytecode". The numbers on the left are byte offsets.

The JVM is a **stack machine**: instead of registers like a CPU, it puts values on an operand stack and operates on the values at the top.

RUNNING · RUN THE CODE STEP BY STEP

Code Trace: The JVM Executes Bytecode



The four source lines compiled to the 12 instructions javap -c shows. **iconst_3** pushes 3; **istore_1** pops it into local variable 1 (a).

Step 1 / 10

```
▶  0: iconst_3
▶  1: istore_1
   2: iconst_4
   3: istore_2
   4: iload_1
   5: iload_2
   6: iadd
   7: istore_3
   8: getstatic     // System.out
  11: iload_3
  12: invokevirtual // println(int)
  15: return
```

VARIABLES

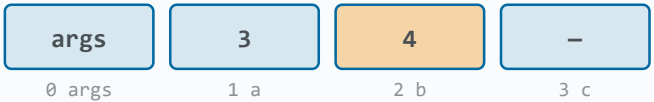
a = 3

OUTPUT

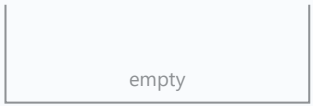
Code Trace: The JVM Executes Bytecode

source: `int b = 4;`

local variables



operand stack



The same pair for b: 4 onto the stack, then into slot 2.

Step 2 / 10

```
 1  0: iconst_3
 2  1: istore_1
 ▶  2: iconst_4
 ▶  3: istore_2
 5  4: iload_1
 6  5: iload_2
 7  6: iadd
 8  7: istore_3
 9  8: getstatic     // System.out
10 11: iload_3
11 12: invokevirtual // println(int)
12 15: return
```

VARIABLES

`a = 3` `b = 4`

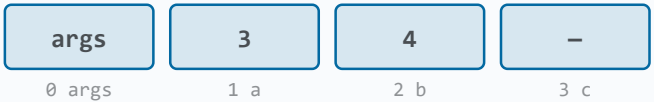
OUTPUT

RUNNING · RUN THE CODE STEP BY STEP

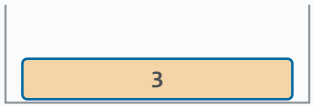
Code Trace: The JVM Executes Bytecode

source: `int c = a + b;`

local variables



operand stack



iload_1 copies a's value onto the stack: [3].

Step 3 / 10

```
 1  0: iconst_3
 2  1: istore_1
 3  2: iconst_4
 4  3: istore_2
 ▶  4: iload_1
 6  5: iload_2
 7  6: iadd
 8  7: istore_3
 9  8: getstatic     // System.out
10 11: iload_3
11 12: invokevirtual // println(int)
12 15: return
```

VARIABLES

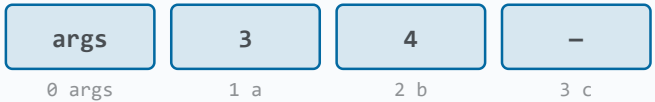
a = 3 b = 4

OUTPUT

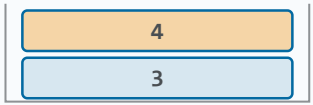
Code Trace: The JVM Executes Bytecode

source: `int c = a + b;`

local variables



operand stack



iload_2: [3, 4]. The two values to add are on top of the stack.

Step 4 / 10

```
1  0: iconst_3
2  1: istore_1
3  2: iconst_4
4  3: istore_2
5  4: iload_1
▶  5: iload_2
7  6: iadd
8  7: istore_3
9  8: getstatic     // System.out
10 11: iload_3
11 12: invokevirtual // println(int)
12 15: return
```

VARIABLES

`a = 3` `b = 4`

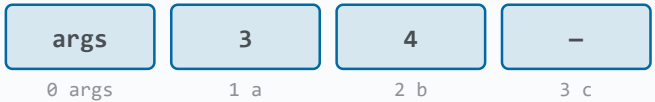
OUTPUT

RUNNING · RUN THE CODE STEP BY STEP

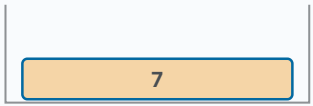
Code Trace: The JVM Executes Bytecode

source: `int c = a + b;`

local variables



operand stack



iadd pops the top two values, adds them and pushes the result: [7].

Step 5 / 10

```
1  0: iconst_3
2  1: istore_1
3  2: iconst_4
4  3: istore_2
5  4: iload_1
6  5: iload_2
▶  6: iadd
8  7: istore_3
9  8: getstatic     // System.out
10 11: iload_3
11 12: invokevirtual // println(int)
12 15: return
```

VARIABLES

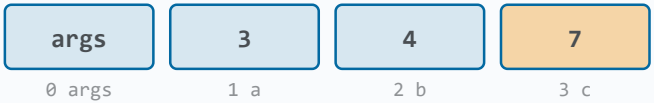
a = 3 b = 4

OUTPUT

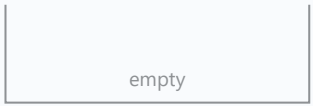
Code Trace: The JVM Executes Bytecode

source: `int c = a + b;`

local variables



operand stack



istore_3: 7 is popped and stored in c. The stack is empty again.

Step 6 / 10

```
 1  0: iconst_3
 2  1: istore_1
 3  2: iconst_4
 4  3: istore_2
 5  4: iload_1
 6  5: iload_2
 7  6: iadd
 ▶  7: istore_3
 9  8: getstatic     // System.out
10 11: iload_3
11 12: invokevirtual // println(int)
12 15: return
```

VARIABLES

a = 3 b = 4 c = 7

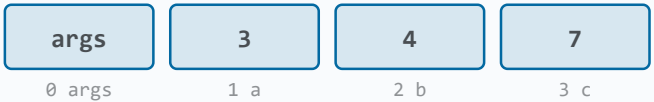
OUTPUT

RUNNING · RUN THE CODE STEP BY STEP

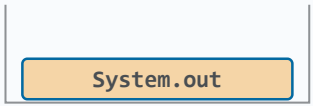
Code Trace: The JVM Executes Bytecode

source: `System.out.println(c);`

local variables



operand stack



getstatic pushes a reference to the System.out object.

Step 7 / 10

```
 1  0: iconst_3
 2  1: istore_1
 3  2: iconst_4
 4  3: istore_2
 5  4: iload_1
 6  5: iload_2
 7  6: iadd
 8  7: istore_3
 ▶  8: getstatic     // System.out
10 11: iload_3
11 12: invokevirtual // println(int)
12 15: return
```

VARIABLES

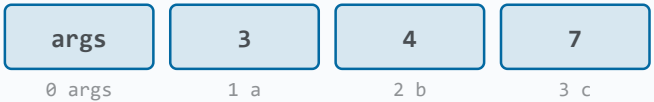
a = 3 b = 4 c = 7

OUTPUT

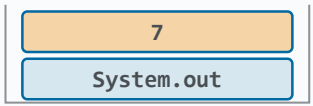
Code Trace: The JVM Executes Bytecode

source: `System.out.println(c);`

local variables



operand stack



iload_3: the value to print (7) is on the stack too.

Step 8 / 10

```
 1  0: iconst_3
 2  1: istore_1
 3  2: iconst_4
 4  3: istore_2
 5  4: iload_1
 6  5: iload_2
 7  6: iadd
 8  7: istore_3
 9  8: getstatic     // System.out
▶ 11: iload_3
11 12: invokevirtual // println(int)
12 15: return
```

VARIABLES

a = 3 b = 4 c = 7

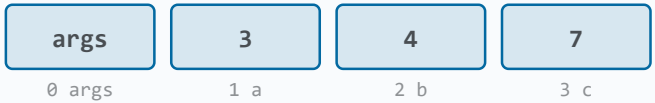
OUTPUT

RUNNING · RUN THE CODE STEP BY STEP

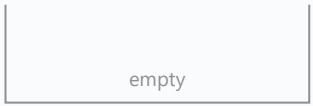
Code Trace: The JVM Executes Bytecode

source: `System.out.println(c);`

local variables



operand stack



invokevirtual pops both values and calls println(int): 7 is printed on the console.

Step 9 / 10

```
 0: iconst_3
 1: istore_1
 2: iconst_4
 3: istore_2
 4: iload_1
 5: iload_2
 6: iadd
 7: istore_3
 8: getstatic     // System.out
11: iload_3
12: invokevirtual // println(int)
15: return
```

VARIABLES

a = 3 b = 4 c = 7

OUTPUT

```
7
```

RUNNING · RUN THE CODE STEP BY STEP

Code Trace: The JVM Executes Bytecode

source: }

local variables

args	3	4	7
0 args	1 a	2 b	3 c

operand stack

empty

return: main ends and its frame is removed. The local variables and the operand stack disappear with the frame.

Step 10 / 10

```
 0: iconst_3
 1: istore_1
 2: iconst_4
 3: istore_2
 4: iload_1
 5: iload_2
 6: iadd
 7: istore_3
 8: getstatic     // System.out
11: iload_3
12: invokevirtual // println(int)
15: return
```

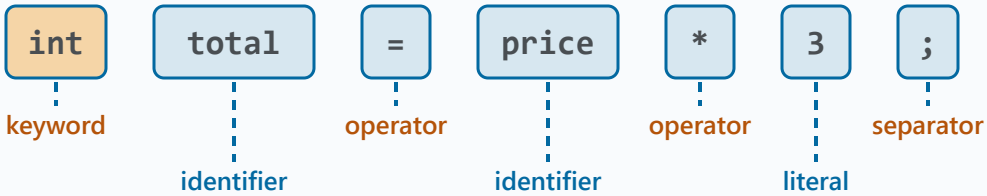
VARIABLES

a = 3 b = 4 c = 7

OUTPUT

```
7
```

Basic Programming Elements



- The compiler first splits the code into **tokens**: keywords, identifiers, literals, operators, separators.
- Java is case-sensitive: `num` and `Num` are different names.
- Most statements end with a semicolon; blocks are enclosed in braces.

LANGUAGE STRUCTURE

Whitespace, Indentation and Braces

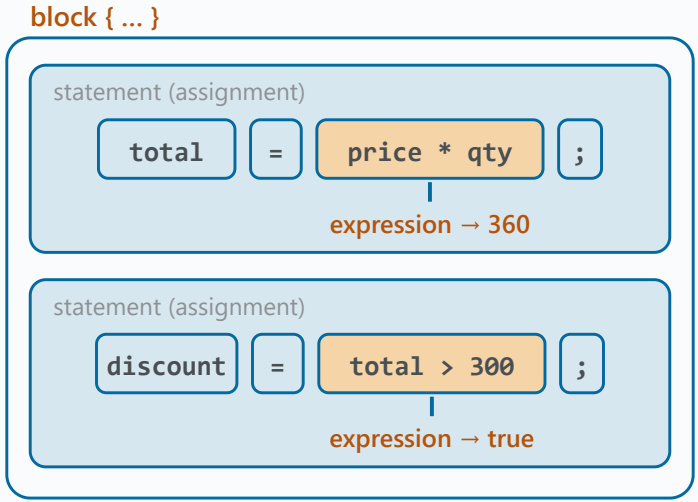
```
public class One { public static void main(String[] args) { System.out.println("Hello"); } }
```

```
public class One {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

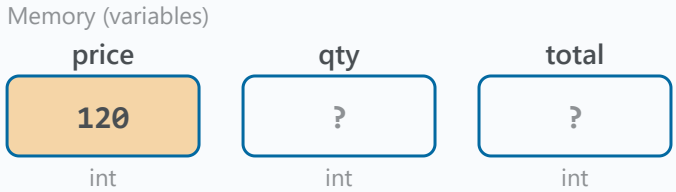
- Both programs compile to **the same bytecode**: to the compiler, spaces, tabs and line breaks only separate tokens. `x=1;` and `x = 1;` are the same, but `publicclass` is a single word.
- **Indentation** shows the reader how blocks nest: each new block starts 4 spaces in, and the closing `}` lines up with the line that opened it.
- By Java convention the opening `{` goes at the end of the line. Spaces inside a string are part of the text: `"Hello "` and `"Hello"` differ.

Expressions and Statements

- An **expression** produces a value: `2 + 3` → 5, `num > 0` → true.
- A **statement** performs an action: `total = 5;` or `System.out.println(total);`
- A **block** groups the statements inside braces into one unit.
- A statement can contain one or more expressions.



Expression, Statement and Block Step by Step



An int box is set aside for **price** and 120 is stored in it.

Step 1 / 7

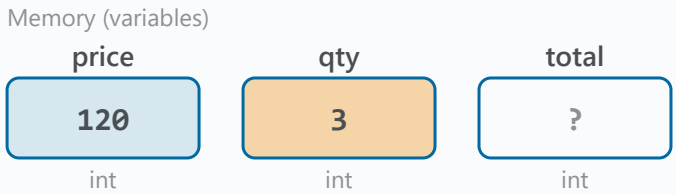
```
▶ int price = 120;            // declaration
2 int qty = 3;                // declaration
3 int total = price * qty;    // expression +
  assignment
4 if (total > 300) {          // condition +
  block
5     System.out.println("Discount");
6 }
```

VARIABLES

`price = 120`

OUTPUT

Expression, Statement and Block Step by Step



The **qty** box is created: 3.

Step 2 / 7

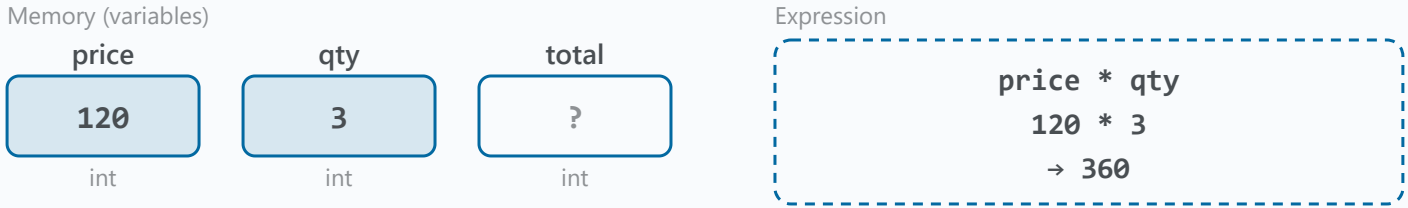
```
1 int price = 120;            // declaration
▶ int qty = 3;                // declaration
3 int total = price * qty;    // expression +
  assignment
4 if (total > 300) {          // condition +
  block
5     System.out.println("Discount");
6 }
```

VARIABLES

`price = 120` `qty = 3`

OUTPUT

Expression, Statement and Block Step by Step



The **expression** on the right is evaluated first: price * qty → 120 * 3 → **360**.

Step 3 / 7

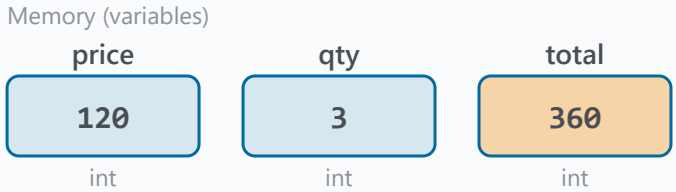
```
1 int price = 120;            // declaration
2 int qty = 3;                // declaration
▶ int total = price * qty;    // expression + assignment
4 if (total > 300) {          // condition + block
5     System.out.println("Discount");
6 }
```

VARIABLES

`price = 120` `qty = 3`

OUTPUT

Expression, Statement and Block Step by Step



The value is assigned to the **total** box: total = 360.

Step 4 / 7

```
1 int price = 120;            // declaration
2 int qty = 3;                // declaration
▶ int total = price * qty;    // expression + assignment
4 if (total > 300) {          // condition + block
5     System.out.println("Discount");
6 }
```

VARIABLES

`price = 120` `qty = 3` `total = 360`

OUTPUT

Expression, Statement and Block Step by Step

Memory (variables)

price	qty	total
120	3	360
int	int	int

Expression

```
total > 300
360 > 300
→ true
```

Condition expression: total > 300 → 360 > 300 → **true**. Since it is true, we enter the block.

Step 5 / 7

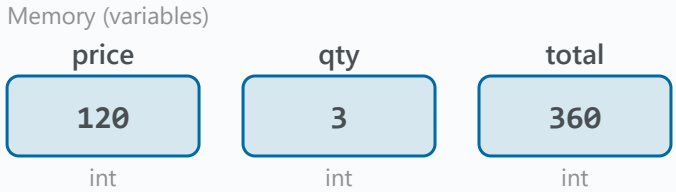
```
1 int price = 120;            // declaration
2 int qty = 3;                // declaration
3 int total = price * qty;    // expression + assignment
▶ if (total > 300) {          // condition + block
5     System.out.println("Discount");
6 }
```

VARIABLES

price = 120 | qty = 3 | total = 360

OUTPUT

Expression, Statement and Block Step by Step



The statement in the block runs: **Discount** is printed.

Step 6 / 7

```
1 int price = 120;            // declaration
2 int qty = 3;                // declaration
3 int total = price * qty;    // expression +
  assignment
4 if (total > 300) {          // condition +
  block
      System.out.println("Discount");
6 }
```

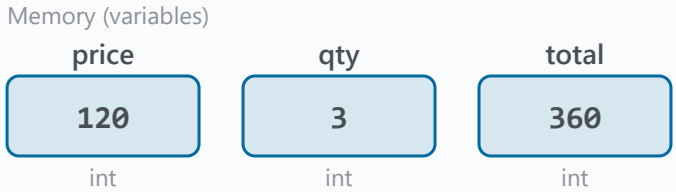
VARIABLES

`price = 120` `qty = 3` `total = 360`

OUTPUT

```
Discount
```

Expression, Statement and Block Step by Step



The block ends. With qty = 2, total would be 240, the condition false, and the block would never run.

Step 7 / 7

```
1 int price = 120;            // declaration
2 int qty = 3;                // declaration
3 int total = price * qty;    // expression +
  assignment
4 if (total > 300) {          // condition +
  block
5     System.out.println("Discount");
▶ }
```

VARIABLES

`price = 120` `qty = 3` `total = 360`

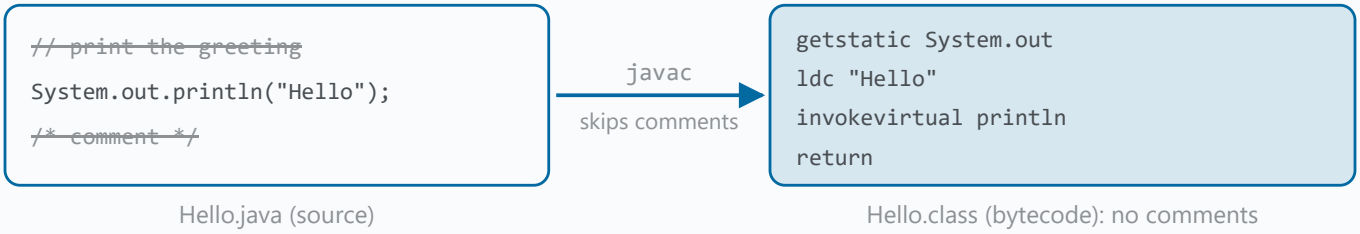
OUTPUT

```
Discount
```

Comments

- `//` starts a comment that runs to the end of the line.
- `/* ... */` spans several lines.
- `/** ... */` is a Javadoc (API documentation) comment.
- The compiler skips comments; they never reach the bytecode.
- A comment should explain **why**, not repeat what the code does.

```
/** The Hello program (Javadoc). */
public class Hello {
    /* A comment spanning
       several lines */
    public static void main(String[] args) {
        // print the greeting
        System.out.println("Hello");
    }
}
```



Hello.java (source)

Hello.class (bytecode): no comments

SOURCE CODE

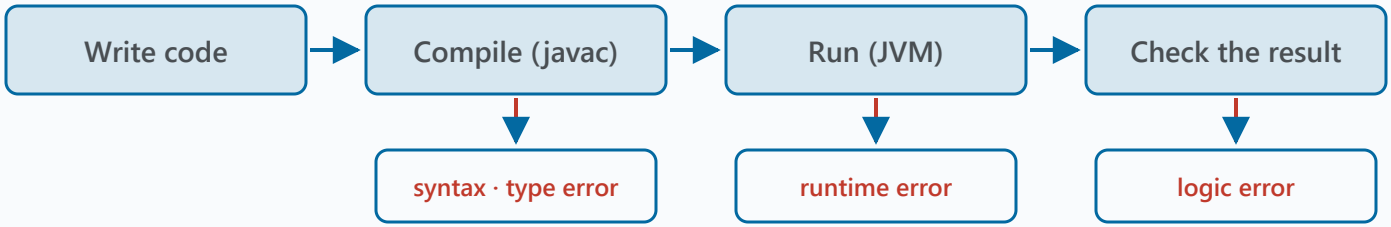
Comment Pitfalls

Code	What happens?
`/* outer /* inner */ rest */`	compile error: `illegal start of expression`. The comment ends at the **first** `*/`, so `rest */` is read as code.
`System.out.println("a // b");`	`a // b` is printed: a `//` inside quotes is part of the text, not a comment.
`// System.out.println(x);`	the line is temporarily disabled; in IDEs, Ctrl+/ comments out the selected lines this way.
`double price = net * 1.20; // VAT 18%`	compiles, but the comment contradicts the code: an outdated comment misleads the reader.

- To disable a section that contains `/* */`, commenting out each line with `//` is safer.
- When a comment and the code disagree, the compiler runs the code; the code is always the truth.

BASICS

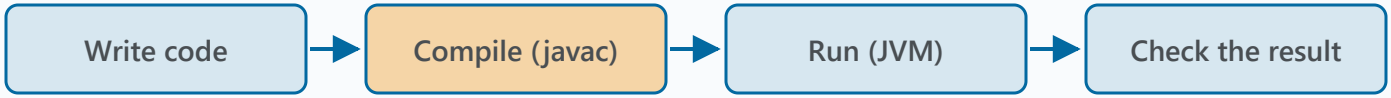
Kinds of Errors



Error	When?	Example
Syntax	At compile time	Missing semicolon
Type	At compile time	Assigning text to a number
Runtime	While the program runs	Division by zero
Logic	The program runs but the result is wrong	Wrong condition

Kinds of Errors by Example

Syntax error



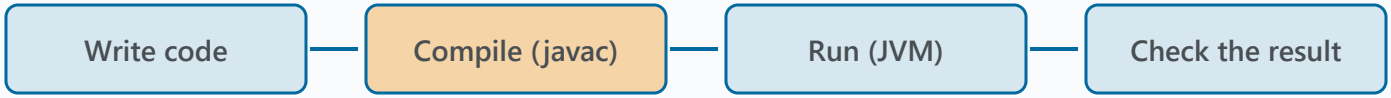

```java
public class Bug1 {
    public static void main(String[] args) {
        System.out.println("Hello")
    }
}
```

```
Bug1.java:3: error: ';' expected
        System.out.println("Hello")
                                   ^
1 error
```

The compiler points to where the rule was broken, with a line number; **no .class is produced**. Fix: add `;` at the end of the line.

Kinds of Errors by Example

Type error



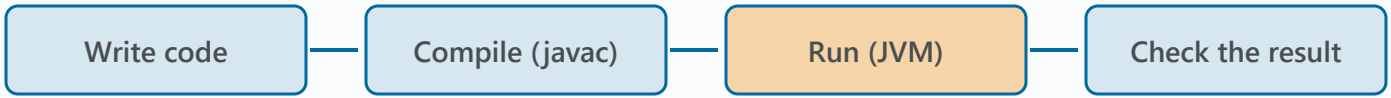

```java
public class Bug2 {
    public static void main(String[] args) {
        int num = "42";
    }
}
```

```
Bug2.java:3: error: incompatible types: S
        int num = "42";
                  ^
1 error
```

A String cannot go into an int box. Types are checked at compile time. Fix: `int num = 42;`

Kinds of Errors by Example

Runtime error




```java
public class Bug3 {
    public static void main(String[] args) {
        int a = 10, b = 0;
        System.out.println(a / b);
    }
}
```

```
Exception in thread "main" java.lang.Arit
        at Bug3.main(Bug3.java:4)
```

The code compiles, but at run time an integer is divided by zero: the JVM throws an **exception** and the program stops.

Kinds of Errors by Example

Logic error




```java
public class Bug4 {
    public static void main(String[] args) {
        int a = 70, b = 90;
        System.out.println(a + b / 2);
    }
}
```

```
115
```

No error message, but the result is wrong: 80 was expected. `/` goes first: 70 + 45. Fix: `(a + b) / 2`.

Common Compile Errors

Error	Actual message	Fix
Missing semicolon	`Hello.java:3: error: ';' expected`	add `;` at the end of the previous statement

Misspelled method (`System.out.println`)	`error: cannot find symbol`
	`symbol: method println(String)`
	`location: variable out of type PrintStream`
		rename it to `println`

Lowercase (`system.out`)	`error: package system does not exist`	`System` starts with a capital S
Name and file don't match	`error: class Greeting is public, should be declared in a file named Greeting.java`	file name = public class name
Wrong launch (`java Greeting`)	`Error: Could not find or load main class Greeting`	no `.class`, or wrong name/location; don't type the extension

The messages come from JDK 21; depending on the version, extra lines may appear (e.g. Caused by: ClassNotFoundException). Always start with the first error.

Reading an Error Message

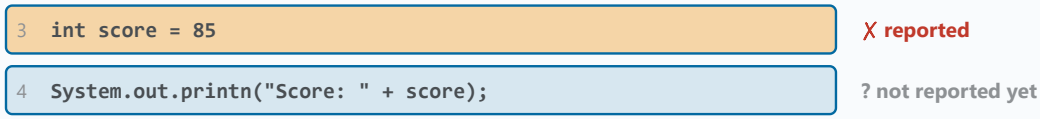
```
Score.java:3: error: ';' expected
        int score = 85
                    ^   ← ^ where the error was noticed
1 error
```

file line kind description

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Bug3.main(Bug3.java:4)          ← stack trace: which method, which file and line
```

- A javac message has the form **file:line: error: description**; below it come the offending source line and a **^** marking the column where the error was noticed.
- The ^ marks not the error itself but where the compiler **noticed** it: a missing `;` is reported at the end of the line, a missing `}` at the end of the file (`reached end of file while parsing`).
- For a runtime error, the first line gives the exception's type and detail; the **stack trace** lines starting with **at** give the method and line where it happened.

Code Trace: Fixing Errors in Order



First compile: only **one** error is reported. While the syntax is broken, javac does not move on to semantic checks; the mistake on line 4 is not visible yet.

Step 1 / 5

```
1 public class Score {
2     public static void main(String[] args) {
▶         int score = 85
4         System.out.printn("Score: " + score);
5     }
6 }
```

VARIABLES

`compile = failed` `errors = 1`

OUTPUT

```
> javac Score.java
Score.java:3: error: ';' expected
        int score = 85
                    ^
1 error
```

Code Trace: Fixing Errors in Order

3	`int score = 85;`	✓ fixed
4	`System.out.printn("Score: " + score);`	? not reported yet

Score.class: none

Fix the spot the message points to: add `;` at the end of the line and save the file.

Step 2 / 5

```
1 public class Score {
2     public static void main(String[] args) {
▶         int score = 85
4         System.out.printn("Score: " + score);
5     }
6 }
```

VARIABLES

`compile = not run` `errors = ?`

OUTPUT

```
> javac Score.java
Score.java:3: error: ';' expected
        int score = 85
                    ^
1 error
```

Code Trace: Fixing Errors in Order

```
3  int score = 85;                                  ✓ fixed
4  System.out.printn("Score: " + score);            ✗ reported now
```

Score.class: none

Recompile: the syntax is now correct, the semantic check runs and cannot find a method named **printn**.

Step 3 / 5

```
1 public class Score {
2     public static void main(String[] args) {
3         int score = 85
▶         System.out.printn("Score: " + score);
5     }
6 }
```

VARIABLES

`compile = failed` `errors = 1`

OUTPUT

```
> javac Score.java
Score.java:4: error: cannot find symbol
        System.out.printn("Score: " +
score);
                  ^
  symbol:   method printn(String)
  location: variable out of type PrintStream
1 error
```

Code Trace: Fixing Errors in Order

```
3  int score = 85;                                  ✓ fixed
4  System.out.println("Score: " + score);          ✓ fixed
```

Score.class: none

The **symbol:** line names what was not found: fix the method name to `println`.

Step 4 / 5

```
1 public class Score {
2     public static void main(String[] args) {
3         int score = 85
▶         System.out.printn("Score: " + score);
5     }
6 }
```

VARIABLES

`compile = not run` `errors = ?`

OUTPUT

```
> javac Score.java
Score.java:4: error: cannot find symbol
        System.out.printn("Score: " +
score);
                  ^
  symbol:   method printn(String)
  location: variable out of type PrintStream
1 error
```

Code Trace: Fixing Errors in Order

```
3  int score = 85;                                  ✓ fixed
4  System.out.println("Score: " + score);          ✓ fixed
```

Score.class: created

The third compile quietly succeeds, Score.class appears and the program runs. The error count may not drop after a fix; it can even grow. Always start with the first error.

Step 5 / 5

```
1 public class Score {
2     public static void main(String[] args) {
3         int score = 85
▶         System.out.println("Score: " + score);
5     }
6 }
```

VARIABLES

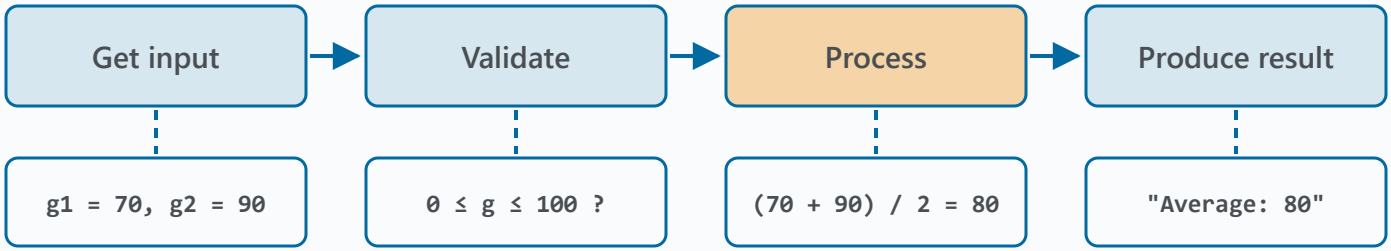
`compile = ok` `errors = 0`

OUTPUT

```
> javac Score.java
> java Score
Score: 85
```

DESIGN

A Program Is a Recipe



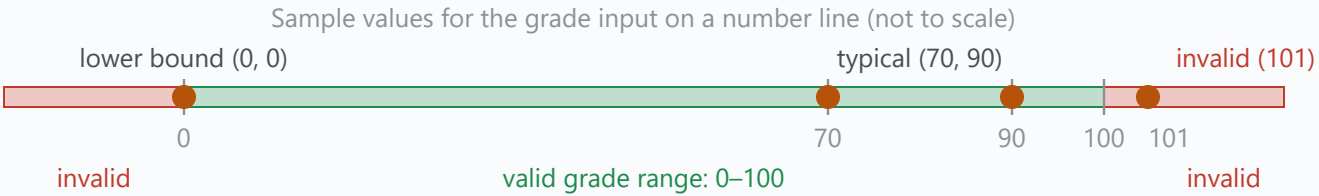
- For each step, state its precondition and the value it produces.
- Large recipes split into subtasks (in Java these will later become **methods**).

Test the recipe by hand before coding: tracing the steps for 70 and 90, do you get 80? If 101 is entered, which step stops it?

DESIGN

Before Coding

- Write a sample input and the expected output.
- Break the solution into steps in plain language or pseudocode.
- Decide on the boundary cases and the error behavior.
- Then turn the steps into Java constructs.

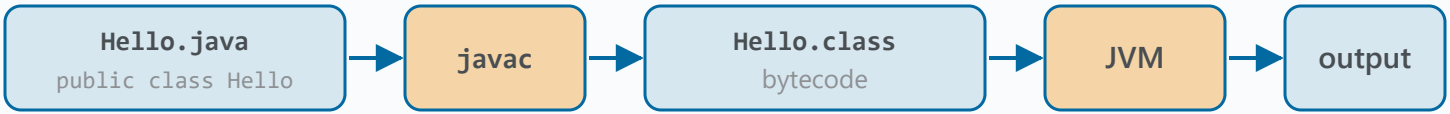


Sample input	Expected output	Why?
70, 90	80	typical case
0, 0	0	lower bound
101, 50	error message	invalid input

SUMMARY

Chapter Summary

- A program is a set of instructions the computer executes to carry out a task.
- A Java source file is compiled to bytecode with javac and runs on the JVM.
- The public class name matches the file name; main is the program's entry point.
- Expressions produce values, statements perform actions, blocks group statements.
- Before coding, settle the sample input, the expected output and the boundary cases.



File name = public class name; the JVM starts execution at the main method.

SUMMARY

Review Questions

1. How are the JDK, bytecode and the JVM related?
2. Which commands compile and run Hello.java?
3. What is the role of the main method?
4. What is the difference between an expression and a statement?
5. How do you tell a syntax error from a logic error?
6. If the source is changed but not recompiled, what does the `java` command run?
7. Why does `"Score: " + 40 + 45` give `Score: 4045`?
8. How do command-line arguments reach the program?

SUMMARY

Exercises

1. Write an `AboutMe` program that prints your name, department and student number on three separate lines; compile and run it.
2. Produce the same output with a single `println` and `\n`.
3. Print a three-line course–credit table using `\t`.
4. Delete the `;` in `System.out.println("Hello")` and compile; interpret the line number and the ^ in the error message.
5. Name the class `Hello` and the file `hello.java`, then compile. What happens, and why?
6. Predict the output of `System.out.println(5 + 3);` and `System.out.println("5 + 3");`; run them to check.

SUMMARY

References

- Deitel and Deitel, Java: How to Program, Early Objects, 11th ed., Pearson, 2018.
- Liang, Introduction to Java Programming and Data Structures, 13th ed., Pearson, 2024.
- Oracle, The Java Tutorials: **A Closer Look at the "Hello World!" Application**.
- Oracle, **The Java Language Specification, Java SE 21 Edition**.
- Sedgewick and Wayne, Computer Science: An Interdisciplinary Approach, Addison-Wesley, 2016: **1.1 Your First Program**.
- Downey and Mayfield, Think Java, 2nd ed., O'Reilly, 2019 (**free online edition**).

The End

Chapter 5: Getting Started with Java

Previous: Chapter 4.3: Introduction to Programming · Next: Chapter 6: Variables, Data Types and Operators