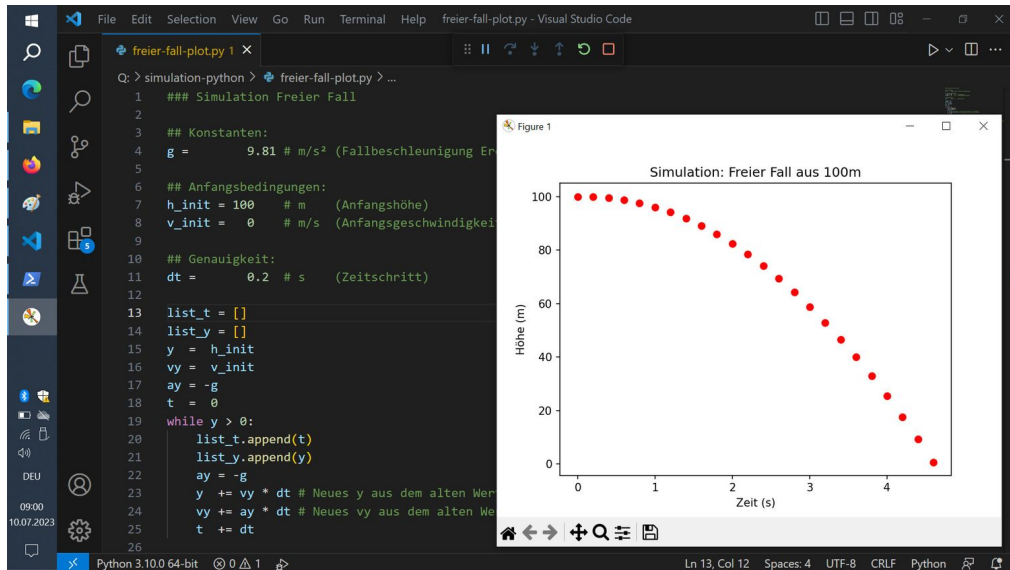


Using Objects and Methods

AP Computer Science A

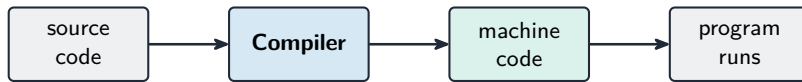
Introduction to Algorithms, Programming, and Compilers



Source code on a workstation — programs are written, compiled, and run as precise instructions

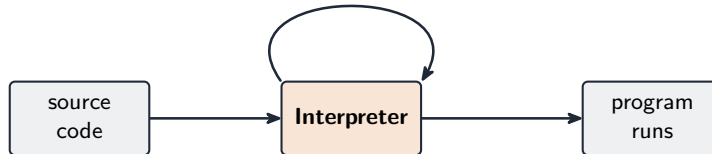
An **algorithm** 算法 is a finite, step-by-step procedure that solves a problem. A **program** 程序 expresses an algorithm in a language a computer can run. Java is **compiled** 编译: the **compiler** 编译器 translates your source code into **bytecode**, which the **Java Virtual Machine (JVM)** runs. A **syntax error** 语法错误 (breaking the grammar) is caught by the compiler; a **logic error** 逻辑错误 (wrong result) is not – the program runs but misbehaves.

Compiler — translate the whole program once



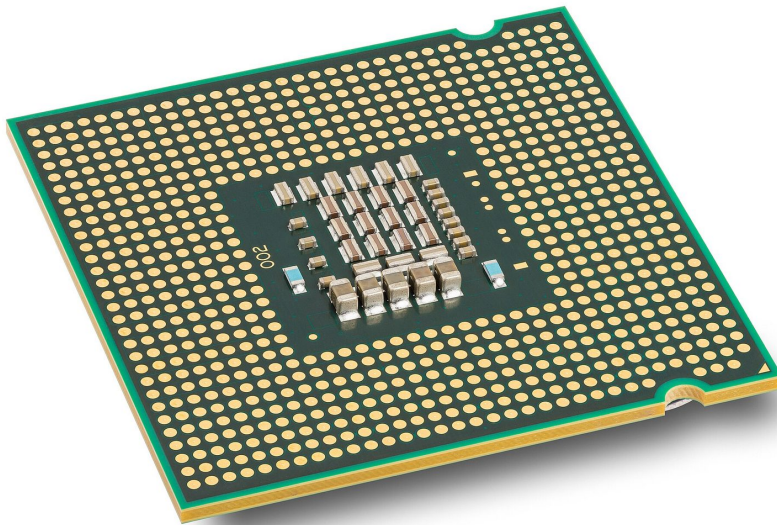
translated once; the machine code then runs many times

Interpreter — one line at a time



reads, translates and runs one line at a time (every run); stops at the first error

A compiler translates the whole program at once; an interpreter runs it line by line



Your Java program is compiled to instructions a CPU like one of these actually runs

Variables and Data Types

A **variable** 变量 is a named box that stores a value of a fixed **type** 类型. Java's main **primitive types** 基本类型 are `int` (whole numbers), `double` (decimals), and `boolean` (true/false). Declare with the type first:

type	stores	example
<code>int</code>	whole number	<code>7</code>
<code>double</code>	decimal number	<code>3.5</code>
<code>char</code>	one character	<code>'A'</code>
<code>String</code>	text (a sequence of chars)	<code>"hello"</code>
<code>boolean</code>	true or false	<code>true</code>

primitives: `int`, `double`, `char`, `boolean` · `String` is an object

Java's basic data types, each storing a different kind of value

```
int score = 90;
double price = 4.99;
boolean passed = true;
```

Expressions and Output

An **expression** 表达式 combines values and **operators** to compute a result: `+` `-` `*` `/` and `%` (**modulus** 取模, the remainder). **Integer division** truncates: `7 / 2` is 3, while `7 % 2` is 1. **Operator precedence** follows math (`*`, `/`, `%` before `+`, `-`). Print with:

```
System.out.print("no newline");
System.out.println("with newline");
```

Dividing an integer by the integer 0 (like `7 / 0`) is not allowed and crashes at run time with an **ArithmeticException**. Inside a string, a backslash marks an **escape sequence** 转义序列: `\` prints a double quote, `\\` a single backslash, and `\n` starts a new line – so `System.out.println("She said \"hi\"");` prints `She said "hi"`.

Assignment Statements and Input

An **assignment** 赋值 `x = expr;` evaluates the right side and stores it in the left variable. Read input with a `Scanner`:

```
Scanner in = new Scanner(System.in);
int age = in.nextInt();
String name = in.next();
```

Casting and Range of Variables

Each type has a fixed range; an `int` overflows past about 2.1 billion. **Casting** 类型转换 converts between types. Widening (`int` to `double`) is automatic; narrowing needs an explicit cast, which **truncates** (does not round):

```
double avg = (double) total / count;    // force real division
int whole = (int) 3.9;                  // 3, truncated
```

Exam skill: watch for integer division producing a truncated result when a decimal was expected – cast one operand to `double` first.

Worked example. Trace each expression:

- $7 / 2 \rightarrow 3$ (both `int`, so division **truncates**);
- $7.0 / 2 \rightarrow 3.5$ (one `double` forces real division);
- $7 \% 2 \rightarrow 1$ (the remainder);
- $(\text{double})\ 7 / 2 \rightarrow 3.5$ (the cast binds tighter than `/`, so it is $7.0 / 2$);
- $(\text{double})\ (7 / 2) \rightarrow 3.0$ (the parentheses compute $7 / 2 = 3$ in `int` **first**, then widen).

The last two look alike but differ – the position of the cast decides whether the truncation happens.

Compound Assignment Operators

Shorthands combine an operation with assignment: `x += 5` means `x = x + 5`; likewise `--`, `*=`, `/=`, `%=`. The **increment** and **decrement** operators `x++` and `x--` add or subtract one.

Application Program Interface (API) and Libraries

An **API** (Application Programming Interface) 应用程序接口 is the published list of classes and methods you may use. A **library** 库 is a collection of ready-made classes (like `Math`, `String`, `Scanner`). You read the API documentation to learn what a method needs (its parameters) and returns, without seeing its inner code – an example of **abstraction** 抽象.

Documentation with Comments

Comments 注释 are ignored by the compiler but explain code to humans: `//` for a single line, `/* ... */` for a block, and `/** ... */` for a **Javadoc** comment that documents a method's purpose, parameters, and return value. Precise **preconditions** and **postconditions** are written here.

Method Signatures

A **method signature** 方法签名 is a method's name plus its parameter types, e.g. `nextInt()` or `substring(int, int)`. To call a method you must supply **arguments** 实参 that match the parameters in number, type, and order. The method header (the full declaration) also states the **return type** – the type of value the method gives back (`void` if none) – but the return type is *not* part of the signature, which is why two methods cannot differ by return type alone.

Calling Class Methods

A **class (static) method** 类方法 belongs to the class itself, so you call it on the **class name**: `ClassName.method(args)`. No object is needed.

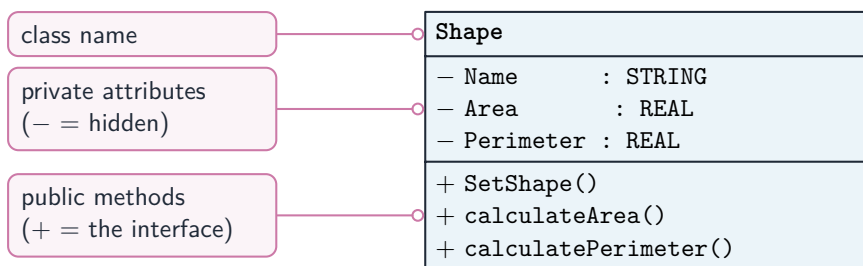
Math Class

The `Math` class provides static math methods: `Math.abs(x)`, `Math.pow(base, exp)`, `Math.sqrt(x)`, and `Math.random()` (a double in $[0, 1)$). To get a random integer from 0 to `n-1`: `(int)(Math.random() * n)`.

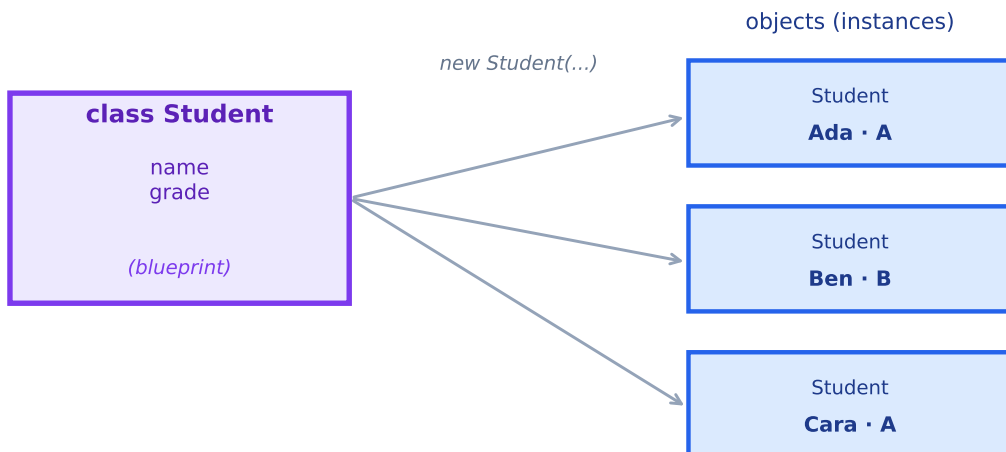
Objects: Instances of Classes

A **class** 类 is a blueprint; an **object** 对象 is a concrete **instance** 实例 built from it. A class bundles **data** (fields) with **behavior** (methods) – the heart of **object-oriented programming** 面向对象编程. `String`, `Scanner`, and `ArrayList` are all classes you instantiate.

Classes can be organised into a hierarchy. A **superclass** 父类 holds attributes and behaviors shared by several **subclasses** 子类 that **extend** it – an **inheritance relationship** 继承关系. Every class in Java is ultimately a subclass of the built-in `Object` class, which is why every object already has a `toString` method; writing a subclass method with the same signature as a superclass one is **method overriding** 方法重写. (Designing your own inheritance is beyond this course, but you are expected to recognise this vocabulary.)



A class diagram: private attributes and public methods



class = blueprint (type) · object = instance with its own field values

A class is a blueprint; each object is one instance built from it

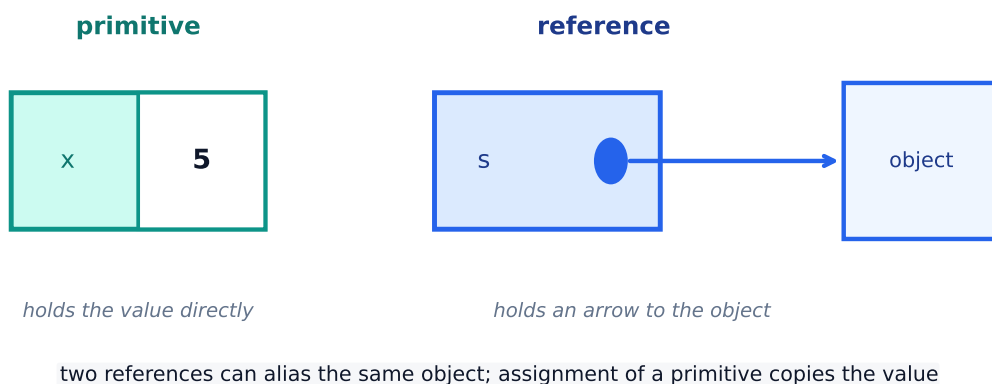
Object Creation and Storage (Instantiation)

Instantiation 实例化 creates an object with the **new** keyword, which calls a **constructor** 构造函数:

```
Scanner in = new Scanner(System.in);  
String s = new String("hi"); // or just "hi"
```

The variable holds a **reference** 引用 (the object's address), not the object itself. Two references can point to the **same** object; comparing them with `==` compares addresses, not contents.

A reference can also point to nothing: the special value **null** 空值 means "not attached to any object". Calling a method on a null reference crashes at run time with a `NullPointerException`. Guard against it by testing with `==/!=` and checking null *first*, so `&&` short-circuits before the method runs: `if (s != null && s.length() > 0)`.



A primitive variable holds its value directly, a reference holds an arrow to the object

Calling Instance Methods

An **instance method** 实例方法 acts on a specific object, so you call it on the **object reference**: `object.method(args)`. Example: `in.nextInt()`, `word.length()`.

String Manipulation

String objects are **immutable** 不可變 – methods return a **new** string rather than changing the original. Key methods (all indices start at 0):

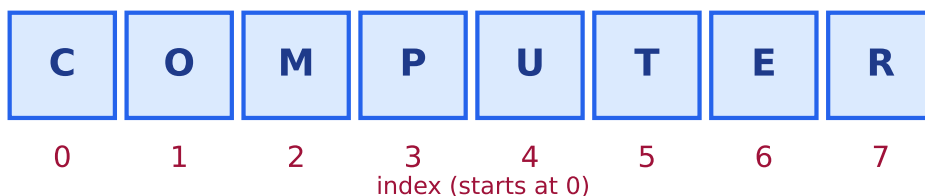
```
s.length();           // number of characters
s.substring(2, 5);     // chars at index 2,3,4 (5 excluded)
s.indexOf("ab");       // first position, or -1
s.equals(other);       // content comparison (never use == for Strings)
s.compareTo(other);    // <0, 0, >0 by dictionary order
```

Exam skill: `substring(a, b)` includes index `a` but **excludes** `b`, and String comparison must use `.equals`, not `==` – two of the most-tested String pitfalls.

Worked example. Let String `s = "COMPUTER"`; (indices 0–7). Then `s.length()` is 8; `s.substring(0, 4)` is "COMP" (indices 0,1,2,3 – index 4 excluded); `s.substring(4)` is "UTER" (from index 4 to the end); `s.indexOf("PU")` is 3; and `s.indexOf("X")` is -1 (not found). Counting the excluded endpoint of `substring` is the single most common slip.

Asking for an index outside 0 to `length()-1` (a bad `substring` or `charAt` argument, e.g. `s.substring(0, 20)` here) crashes with a `StringIndexOutOfBoundsException` – the String cousin of the array-index error.

"COMPUTER".substring(0, 4) → "COMP"



valid indices: 0 ... `s.length() - 1` · `s.charAt(i)` is the char at `i`

String indices start at 0

Exam tips

- Trace code by hand line by line, tracking each variable's value in a table — the exam rewards careful tracing over guessing.
- Know Java's primitive types and that integer division truncates (7/2 gives 3); use a cast or a double for real division.

- Distinguish **compile-time** errors (syntax, types) from **run-time** errors – know the named ones: `ArithmeticException` (`int ÷ 0`), `NullPointerException` (method on a null reference), `StringIndexOutOfBoundsException` / `ArrayIndexOutOfBoundsException` – and **logic** errors (wrong output).
- Follow operator precedence and initialise every variable before you use it.
- On the free-response, write **complete, compilable** Java — return the right type and match the method header exactly.