

Primitive Types & Using Objects

AP Computer Science A ■ Unit 1

AP Computer Science A

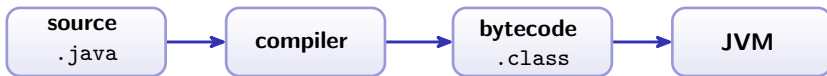
```
int x = 5;
```

What we will cover

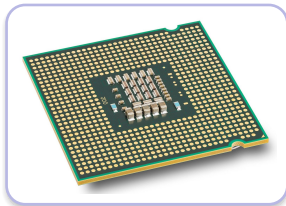
- 1 Programs & compilers
- 2 Variables & data types
- 3 Expressions & casting
- 4 Methods & the Math class
- 5 Objects & strings



From source code to running program



An **algorithm** is a step-by-step plan; a **program** is that plan in a **programming language** like Java. The compiler turns human-readable source into **bytecode** the JVM runs.



Cpu chips

Documentation with Comments

Comments are ignored by the compiler but explain code to humans: `//` for a single line, `/*` ...

`*/` for a **Javadoc** comment that documents a method's purpose, parameters, and return value.

Method Signatures

A **method signature** is a method's name plus its parameter types, e.g.

To call a method you must supply **arguments** that match the parameters in number, type, and order.

Variables & primitive types

A **variable** is a named box that stores a value of a fixed **type** .

Java's main **primitive types** are 'int' (whole numbers), 'double' (decimals), and 'boolean' ('true'/'false').

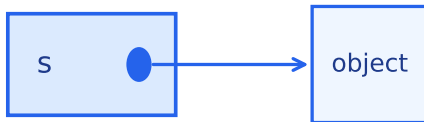
Variables & primitive types, in detail

primitive



holds the value directly

reference



holds an arrow to the object

two references can alias the same object; assignment of a primitive copies the value

A **variable** is a named box that stores a value of a fixed **type** .

Variables & primitive types, continued

<i>type</i>	<i>stores</i>	<i>example</i>
<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 characters)	<code>"hello"</code>
<code>boolean</code>	true or false	<code>true</code>

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

A **variable** is a named box that stores a value of a fixed **type**.

Java's main **primitive types** are `'int'` (whole numbers), `'double'` (decimals), and `'boolean'` (`'true'`/`'false'`).



Expressions, casting & range

An **expression** combines values with **operators** (+ - * / %), following order of operations.

Integer division & casting

7 / 2 is 3 (integer division drops the remainder); 7 % 2 is 1.

Cast to get a decimal: (double) 7 / 2 is 3.5. A value too big for its type causes **overflow**.

Compound assignment : `x += 3` means `x = x + 3`.

Integer division truncates, and the cast position decides

the trap

`7 / 2` is 3, not 3.5 – two ints divide to an int

the fix

cast **one operand** first: `(double) 7 / 2` is 3.5

the wrong fix

`(double) (7 / 2)` is 3.0 – the truncation already happened inside the brackets

so

the **position of the cast** decides whether the truncation happens at all

escape sequence

`\n` and `\t` inside a String – a backslash changes what the next character means

Watch for integer division producing a truncated result where a decimal was wanted. It compiles, it runs, and it is silently wrong – a **logic error**.

Assignment Statements and Input

An **assignment** `x = expr;` evaluates the right side and stores it in the left variable.

Read input with a Scanner:

The shorthand operators

compound assignment

`x += 5` means `x = x + 5`; likewise `--`, `*=`, `/=`, `%=`

increment and decrement

`x++` and `x--` add or subtract one

why they matter

they are everywhere in `for` loops, so you must read them fluently even if you never write them

the caution

`x /= 2` on an `int` still truncates – the shorthand does not change the type rules

These are pure convenience: every one expands to a longer form you already know. Expand it mentally when tracing.



Methods, the API & the Math class

A **method** is named, reusable code. Its **signature** is its name, parameters & return type.

API & libraries

documentation of ready-made classes to reuse

Math class

`Math.sqrt(9)`, `Math.pow`,
`Math.random()`

A **comment** (`// ...`) documents code for humans; the compiler ignores it.

Class methods, instance methods, and where code comes from

class (static) method

belongs to the **class itself**, so you call it on the class name:
`Math.abs(x)`. **No object needed**

instance method

belongs to an **object**, so you call it on a reference:
`s.length()`

a library

pre-written classes you can use; Java's is the **API**

compiled

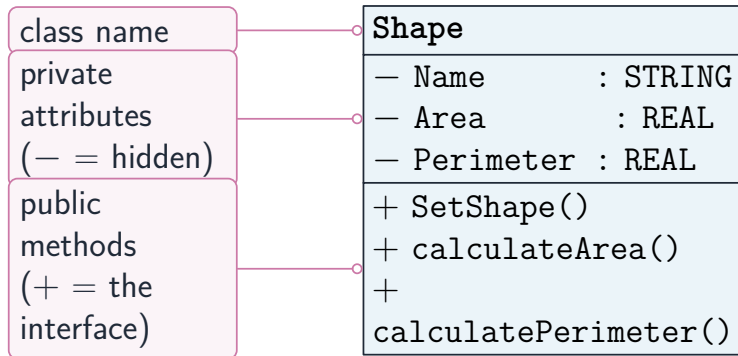
Java source is compiled once to bytecode, then run – so
syntax errors surface **before** the program starts

abstraction

using a method by its name and contract, without knowing
how it works inside

If you find yourself creating an object just to call a method on it, check whether the method is static – `Math.sqrt` on a new `Math()` does not compile.

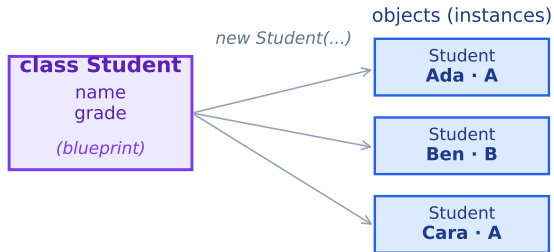
Methods, the API & the Math class, in detail



A class diagram: private attributes and public methods



Objects: instances of classes



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

A **class** is a blueprint; an **object** is a built **instance** .

Instantiate with `new`:

```
Scanner s = new  
Scanner(...);
```

Call **instance methods** on an object with the dot operator: `s.nextInt()`.

Superclass and subclass

superclass

holds attributes and behaviours **shared** by several subclasses

subclass

extends the superclass, **inheriting** everything it has and adding its own

inheritance relationship

an **is-a** relationship: a Dog *is a* Animal

method overriding

a subclass redefines an inherited method with the same signature, and **its** version runs

why bother

shared code is written once, and code that works on the superclass works on every subclass

Ask “is-a” before writing extends. A Car *has a* Engine, so that is a field, not a superclass – the commonest design error at this level.

Object Creation and Storage (Instantiation)

Instantiation creates an object with the `new` keyword, which calls a **constructor** :

Two references can point to the **same** object; comparing them with `==` compares addresses, not contents.

String manipulation

"COMPUTER".substring(0, 4) → "COMP"



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

A **String** is an object – an ordered sequence of characters, indexed from **0**.

- s.length()
- s.substring(0, 4)
- s.indexOf('`x`')

Concatenate with +: ``Hi, ' ' + name.

The two most-tested String traps

substring(a, b)

includes index a but **excludes** b – so its length is b - a

comparison

use `.equals`, **never** `==` – `==` compares *references*, so two equal Strings can test false

immutable

a String cannot be changed; every “modification” returns a **new** String

indexing

`indexOf` returns -1 when not found, and indices run 0 to `length()-1`

Both traps compile cleanly and both give wrong answers at run time. They are among the most-tested points on the paper for exactly that reason.

Key points

Java **source** → compiler
→ **bytecode** → JVM

Strongly typed: int,
double, boolean

$7/2=3$; **cast** for decimals; $x += 3$

A **class** is a blueprint;
an **object** is an instance

Worked example – the excluded endpoint

```
String s = "COMPUTER"; (indices 0–7)
```

```
s.length() → 8
```

```
s.substring(0, 4) → "COMP" (index 4  
is excluded)
```

```
s.substring(4) → "UTER" (index 4 to the  
end)
```

```
s.indexOf("PU") → 3
```

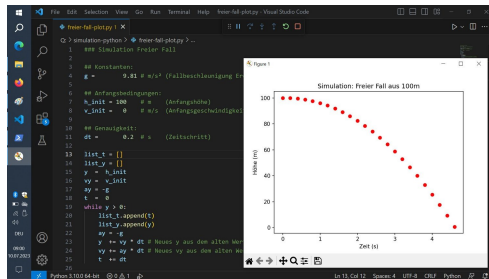
```
s.indexOf("X") → -1 (not found)
```

Miscounting substring's **excluded endpoint** is the single most common slip on this paper.

Introduction to Algorithms, Programming, and Compilers

An **algorithm** is a finite, step-by-step procedure that solves a problem.

A **program** expresses an algorithm in a language a computer can run.



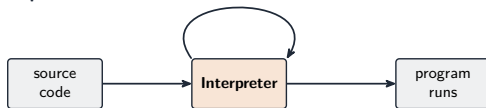
A compiler translates the whole program at

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

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` ($\text{int} \div 0$), `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.



Unit 1 – key takeaways

1

Trace

line by line in a table – no guessing

2

Types

$7/2 = 3$; cast or double for real division

3

Errors

compile-time vs run-time vs logic

4

FRQ

complete compilable Java; match the header

Check yourself

- 1 "COMPUTER".substring(0,4) gives what?
- 2 What does indexOf return when the string is absent?
- 3 Is int a primitive or a reference type?
- 4 What does the compiler turn source code into?

Answers 1. "COMP" 2. -1 3. primitive 4. bytecode, run by the JVM