Advanced Programming

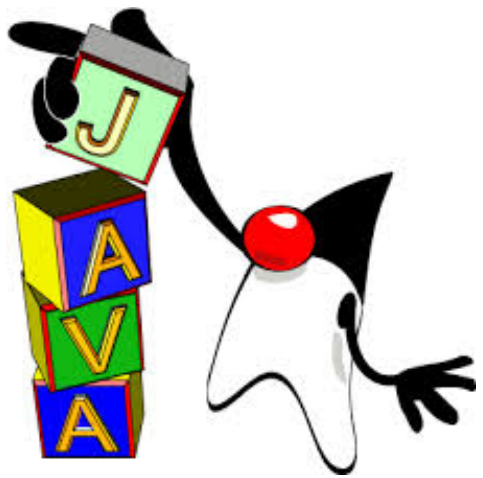
Lecture 1

Introduction

2025 - 2026

Course Description

- `edu.info.uaic.ro/programare-avansata`
- `profs.info.uaic.ro/cristian.frasinaru` (A+B)
- `profs.info.uaic.ro/bogdan.prelipcean` (E)
- The Goal
- The Motivation
- Lectures and Assignments
- Programming Platform
- Resources
- Evaluation
  - Lab: problems, quizzes $\rightarrow$ easy
  - Exam: written test $\rightarrow$ hard



Syllabus

1. Introduction: Java Standard Edition
2. Objects and Classes
3. Interfaces. Generics. Collections
4. Collections. Java Stream API
5. Exceptions. Input/Output Streams. Working with Files
6. Creating Database Applications with JDBC
7. REST Services
8. Graphical User Interfaces (Swing, Java2D, JavaFX)
9. Concurrency: Threads and Locks
10. Introduction to Java Persistence API (JPA)
11. Network Programming
12. Class Loading. Reflection API. Annotations
13. Internationalization and Localization.

Programming Languages

- **Imperative**: C, Java, Python, etc.
- **Object-Oriented (OOP)**: Java, C++, C#, Scala, Kotlin, etc.
- **Functional**: Haskell, Lisp/Scheme, Elixir, Clojure, Scala, etc.
- **Scripting**: Python, JavaScript, Perl, Ruby, Groovy, etc.
- **Low level / System**: Assembly, C, Rust, etc.
- **Domain specific**: SQL, template languages (FTL, VTL), etc.
- **Educational**: Scratch, Logo, etc.
- ...

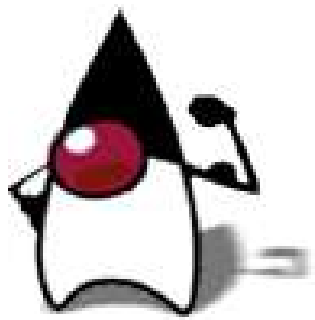
Some focus on a **single paradigm**, such as C, but most of them are **multi-paradigm**, such as Java.

Types of Applications

- **Desktop**
  Run on personal computers, usually with a GUI.
  Office tools, design software, IDEs, point of sales (POS), etc.
- **Web**
  Backend: Run on servers, offering services to frontends.
  Fontend: Run in browsers using HTML, CSS, JavaScript, etc.
  Dynamic apps, SPA, PWA, etc.
- **Enterprise**
  Large-scale systems for business operations.
- **Mobile**
  Native apps (**Android**/iOS), Cross-platform apps, Hybrid apps.
- **Embedded**, **Data-driven**, **Scientific**, **Games**, ...

What Exactly Is "Java"

- Programming **language**
  $\rightarrow$ High level, general-purpose, object-oriented
- Programming **platform**
  $\rightarrow$ Mutiple languages (Java, Kotlin, Scala, Clojure, Groovy, etc.), runtimes, tools, frameworks, application servers and other resources.
- Born in 1995, "younger" than C++ (1985) or Python (1991)
- Created by **James Gosling** and his team at **Sun Microsystems**
- Acquired by **Oracle** in 2010
- One of the most versatile and used programming language.



First Program

A Java program that prints "Hello, World!" on the screen:

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

- `HelloWorld` is then **main class** (contains `main`).
- The signature of `main` must be exactly like this.
- The class must be in a file called `HelloWorld.java`.
- `System.out` is a reference to the standard output stream.

This is the same, but in Groovy :)

```
println "Hello, World!"
```

How to Run It

Java code is **both compiled and interpreted**.

```
javac HelloWorld.java
```

The compiler **javac** will produce a **compilation unit**, named `HelloWorld.class`, which is a binary file containing **bytecode** (intermediate language). This is the "executable" of our program.

```
java HelloWorld
Hello, World!
```

java is the executable for the **Java Runtime Environment (JRE)**.

- It receives **the name of the main class**, not its filename (!).
- It provides the necessary software components to run Java apps:
  - **Java Virtual Machine (JVM)**
  - **Java Class Libraries**

Java Development Kit (JDK)

- **JDK**: the core tools to **develop** a Java app, library, etc.
- Includes:
  - Java Compiler
  - **JRE**: all that is required to **run** a Java program
  - Debugger, profiler, documentation generator, dissasembler, etc.
- **Popular JDK Distributions:**
  - Oracle JDK
  - OpenJDK (open-source)
  - Others: Amazon Corretto, Eclipse Temurin
- Each JDK targets a **specific architecture and OS**: 32/64 bits, Windows / macOS / Linux, etc.
- Needed by **Integrated Development Editors (IDE)**, such as IntelliJ, NetBeans, Eclipse. JDK does not contain an IDE.
- System variables of interest: **JAVA_HOME, PATH**
- New Java versions are released every 6 months, and LTS (Long Time Support) every 2 years.

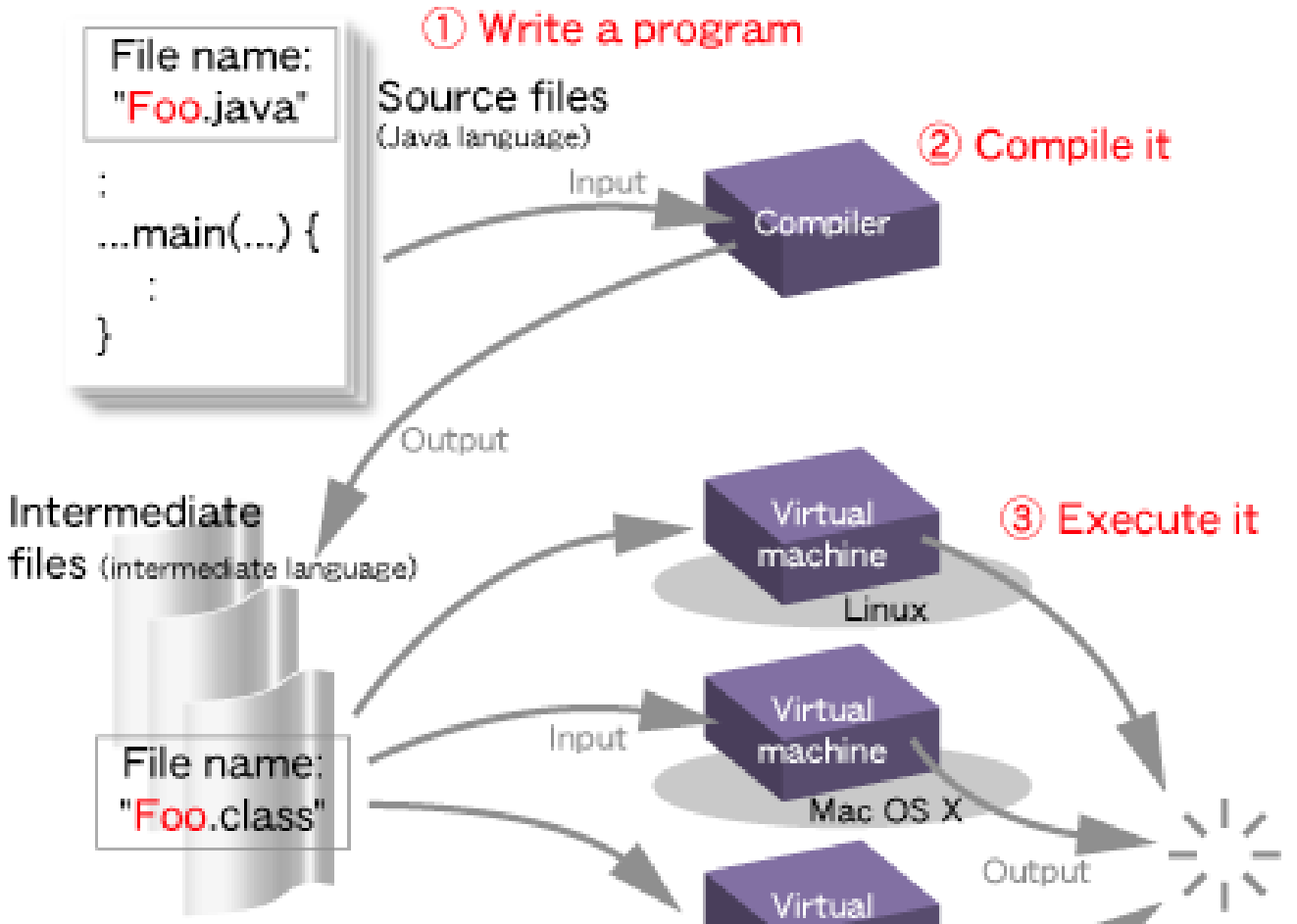
Compiled and Interpreted

- **Interpreted**: the source code is executed instruction-by-instruction (line-by-line) by an interpreter at runtime.
  - \+ simplicity, portability
  - – low execution speed
- **Compiled**: the source code is translated into binary instructions (usually machine code) by a compiler, before the program is run.
  - \+ high execution speed
  - – no portability

Java code is **both compiled and interpreted**. The Java compiler does not generate machine code (native hardware instructions). Rather, it generates **bytecode**, a high-level, machine-independent code for a hypothetical machine:

→ **The Java Virtual Machine (JVM)** 🚀

Java Virtual Machine (JVM)



- **Actually in many cases, a program is distributed in the form of a JAR file, which stands for Java ARchive file.**

Why Study Java?

"Everything should be made as simple as possible, but not simpler."

- **"As simple as possible"**
  - Readable
  - Easy to learn
  - Clean object-oriented programming
  - Automatic memory management (garbage collection)
- **"But not simpler"** (doesn't oversimplify)
  - Strong static typing (type safety)
  - Multithreading support
  - Secure class loading and verification
  - Performance
- **Architecture neutrality**: independent of the underlying hardware architecture (x86, ARM, etc.)
- **Portability**: independent of the operating system and environment (Windows, macOS, Linux, etc)
- **Backward compatibility**

Is Java Slow?

Let's make an "intensive" $O(n^2)$ bubble sort, empirical study:

```
int n = 100_000;
int a[] = new int[n];
for (int i = 0; i < n; i++) { a[i] = n - i; }

long t1 = System.currentTimeMillis();
for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
        if (a[j] > a[j + 1]) {
            int aux = a[j];
            a[j] = a[j + 1];
            a[j + 1] = aux; //there is no swap method (!)
        }
    }
}
long t2 = System.currentTimeMillis();
System.out.println(t2 - t1); // 1 sec = 1000 ms
```

Java: **5 s**, C++ Release: **4 s**, C++ Debug: **20 s**, Python (try it yourself). Arrays.sort(a): **4 ms** 💡

JVM Performance

- In early versions, bytecode was only interpreted.
- Now, there is **Just-in-time (JIT) compilation**
  1. Source code is compiled to bytecode (.class files).
  2. Bytecode is interpreted by the JVM.
  3. Frequently executed bytecode ("hot code") is detected.
  4. Hot bytecode is transformed into native machine code, just in time, during execution, by the JIT compiler.
  5. Next time the hot code runs, it **runs much faster (10x)**.
- This is why, **HotSpot** is the name of the default Java Virtual Machine (JVM) implementation used by Oracle JDK.
- **Cold start**: The first request made to a Java application is often substantially slower than the average response time during its lifetime. This happens because: **JVM startup costs**, **lazy class loading** and **JIT compilation**.
- When benchmarking a Java app, we should **pre-warm** it.

The Character Set: Unicode

"Without Unicode, Java wouldn't be Java, and the Internet would have a harder time connecting the people of the world.", *J. Gosling*

```
public class Приветмир {
    public static void main(String[] args) {
        System.out.println("Привет, мир!");
    }
}
```

- Unicode is a **universal character encoding standard**.
- Each character has a numerical value, called **code point**.
- **Unicode Transformation Format (UTF)** specifies how character **code points** are represented:
  - **UTF-8**: 1-4 bytes, most common for files, I/O, Web
  - **UTF-16**: 2 or 4 bytes, used in Java, C#, Windows
  - **UTF-32**: 4 bytes, not very common.
- Java uses **UTF-16** encoding, a `char` is always **2 bytes**.
- To specify a char by its code: **\uXXXX** (**\u03B1** is α)

Example of Using Characters

```
public class TestCharacters {

    public static void main(String args[]) throws Exception {
        for (char c = 'a'; c <= 'z'; c++) {
            System.out.println(c + " has the code " + (int)c );
            //The cast operator: (Type) expression
        }
        for (int i = 'a'; i <= 'z'; i++) {
            System.out.println(i + " is the code for " + (char)i );
        }
        //Let's write the greek alphabet to a file
        var out = new java.io.PrintWriter("greek.txt", "UTF-8");
        char alpha = '\u03B1';
        char omega = '\u03C9';
        for (char c = alpha; c <= omega; c++) {
            out.println(c + " has the code " + (int)c);
        }
        out.close();
        //We'll talk about IO streams later
    }
}
```

Java Basic Syntax

- **Similar to C++**
- **Keywords**: 53 (C++:95, C#: 79, Python: 36, Go:25, SmallTalk:0)
- **Literals**: "Hello World", 'J', 'a', 'v', 'a', 10, 010, 0xA, 0b11, 12.3, 12.3d, 12.3f, 12e3, 123L, true, false, null, 0722_123_456
- **Separators**: $()\{\}[];,.$
- **Operators**: mostly as in C++
- Example:
  (char)65 + "nna" + " has " + $(8 >> 2)$ + " apples"
- **Java Tutorial**
  `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html`

Comments

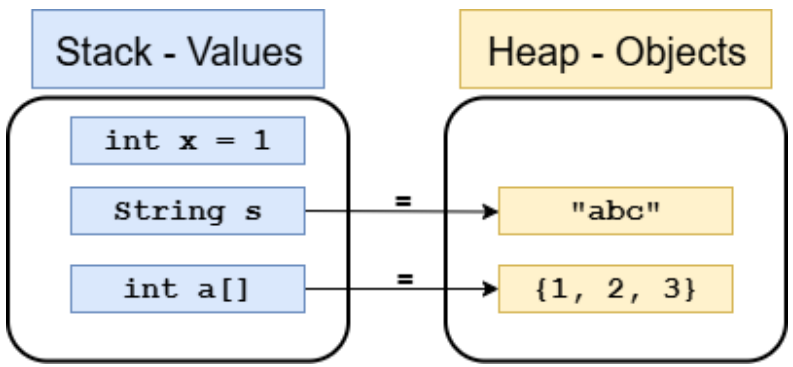
```
/* To change this template file, choose Tools | Templates
and open the template in the editor. */

/**
* Main class of the application
* @author Duke
*/
public class HelloWorld {
    /**
    The execution of the application starts here.
    @param args the command line arguments
    */
    public static void main(String args[]) {
        // TODO code application logic here
        System.out.println("Hello, World!"); // Done!
    }
}
```

The doc comments /** are used by **javadoc** - a tool for generating API documentation in HTML format.

Data Types

- **Primitive types** (a variable of primitive type **holds a value**)
  - **Arithmetic** (Integer types)
    → `byte (1), short (2), int (4), long (8)`
  - **Floating point**
    → `float (4), double (8)`
  - **Character**
    → `char (2)`
  - **Logical**
    → `boolean (?)`
- **Reference types** (variables that **point to objects**)
  - Arrays
  - Classes
  - Interfaces
  - Records
- ~~`pointer, struct, union`~~



Numerical Data Types: Compromise vs Performance

- **Performance**: High control over precision / Platform specific

```
//C++
short, int, long, long long, unsigned int, unsigned long, ...
int8_t, int16_t, uint64_t, int_least16_t, int_fast32_t, ...
//no built-in support for arbitrary large numbers
```

- **Compromise**: Fixed, fast, cross-platform types / Less control

```
//Java
byte, short, int, long
//no support for unsigned
//java.util.BigInteger class for arbitrary large integers
```

- **Convenience**: Rapid prototyping / No control, Slow

```
//Python
int // a built-in data type, not a keyword
x = 2**100;
//arbitrary precision
```

Primitive Numeric Type Precision

- **Integer Types (Exact Precision)** on n bits: $[-2^{n-1}, 2^{n-1} - 1]$

`byte`	$[-2^7, 2^7 - 1]$	$[-128, 127]$
`short`	$[-2^{15}, 2^{15} - 1]$	$[-32\,768, 32\,767]$
`int`	$[-2^{31}, 2^{31} - 1]$	$[-2\,147\,483\,648, 2\,147\,483\,647]$
`long`	$[-2^{63}, 2^{63} - 1]$	[0x8000000000000000L, 0x7fffffffffffffffL]

- **Floating-point (Approximate Precision)**
  IEEE 754 standard: $value = (-1)^{sign} \times 1.mantissa \times 2^{exponent-bias}$

`float`	32	$\approx 6 - 7$ digits precision
`double`	64	$\approx 15 - 17$ digits precision

```
float x = 0.1f;
System.out.printf("%.20f%n", x); //0.10000000149011612000
double y = 1234567890.1234567890d;
System.out.printf("%.20f%n", y); //1234567890.12345670000000000000
```

Variables

- Java is **statically typed**: variables must be declared with a type.

```
int value;
long numberOfElements = 12345678L; //primitive type
String myFavouriteDrink = "water"; //reference type
x = 1; //not legal, unless x was declared before
```

- For local variables, **the data type can be inferred**.

```
var myFavouriteDrink = "water"; //String is inferred
```

- A variable's name can start with a letter, "$", or "_".
  Subsequent characters can be letters, digits, "$", or "_".
  Cannot be a Java reserved keyword.

```
double $price; //legal, but not recommended
char _letter;  //legal, but not recommended
var var = "?"; //legal, but not recommended
//var is a contextual keyword (not a reserved keyword)
int 2ndNumber; //not legal
String short;  //not legal
```

Example of Declaring Variables

Let's compute π using Leibniz series: $\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$

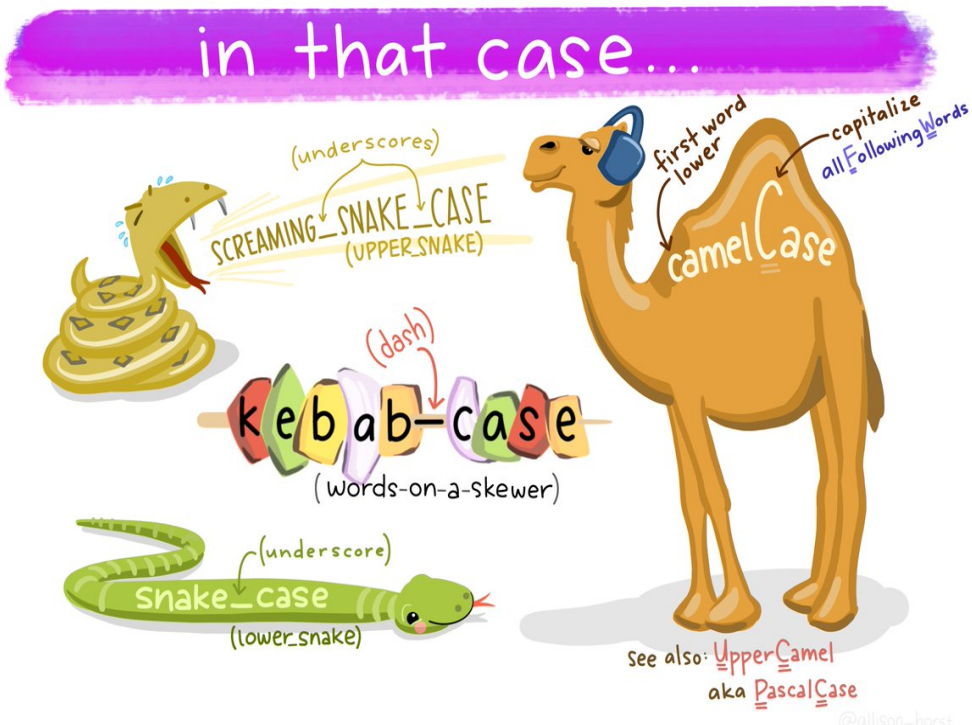
```
public class LeibnizSeries {
    public static final double PI = 3.141592653589793; //constant
    private static final double EPSILON = 1e-6;
    private static final int MAX_STEPS = 10_000_000;
    private double result; //class member variable, 0 by default

    public double computePi(int nbOfSteps) { //method argument
        result = 0.0; //is this necessary?
        double sign = 1.0; //local to the method
        for (int n = 0; n < nbOfSteps; n++) { //n is local to the block
            result += (sign / (2 * n + 1)); sign = -sign;
        }
        result *= 4;
        return result;
    }
    public static void main(String ars[]) {
        var app = new LeibnizSeries(); //type inference
        System.out.println(app.computePi(MAX_STEPS));
        System.out.println(Math.abs(app.result - PI) < EPSILON);
    }
}
```

Naming Conventions

Rules for naming classes, methods, variables, constants and packages
$\rightarrow$ readability, maintainability, consistency with standard practices.

Classes	UpperCamelCase
Variables	camelCase
Methods	camelCase
Constants	SCREAMING_SNA
Packages	lower.case



Control Flow Statements

- **Decision-making**

  `if-else`, `switch`

- **Looping**

  `for`, `while`, `do-while`

- **Exception handling**

  `try-catch-finally`, `throw` `//we'll cover this later`

- **Branching**

  `break`, `continue`, `return`, ~~`goto`~~

```
int i = 0;
outer:
while (i < 10) {
    for (int j = 0; j < 10; j++) {
        if (i == j) continue;
        if (i + j >= 10) break outer;
    }
    i++;
}
```

Switch Statements

```
//Used as an expression, switch may return a value.
//-> arrow syntax replaces case + break
char ch = 'a';
String type = switch (ch) {
    case 'a', 'e', 'i', 'o', 'u' -> "Vowel";
    default -> "Consonant";
};

//Switch also works with strings, not only with primitive types
//Classical switch
String answer = "yes";
int value;
switch (answer) {
    case "no":
        value = 0;
        break;
    case "yes":
        value = 1;
        break;
    default:
        value = -1;
}
```

Arrays

- An array holds **a fixed number of values of the same type**.

```
int[] values; //declared, no memory allocated
int[] values = new int[10]; //10 integers, 0 by default
int size = 10; int[] values = new int[size]; //no problem

char[] letters = {'a', 'b', 'c'}; //declared and initialized
String colors[] = {"Red", "Green", "Blue"};

int[10] values; //not legal, compile error
int[] values; values[0] = 1; //runtime exception
```

- Arrays are **zero-based indexed**

```
for(int i = 0, n = values.length; i < n; i++) values[i] = i;
```

- Multidimensional arrays are **arrays of arrays**.

```
boolean[][] matrix = new boolean[2][3]; // 2 rows, 3 columns
int[][] array2d = { {1, 2}, {3, 4, 5, 6} }; // different sizes
int[][] adjList = new int[3][]; //dynamic
adjList[0] = {1}; adjList[1] = {0, 2}; adjList[2] = {1};
```

Example: Matrix Multiplication

$A_{m,n} \times B_{n,p} = C_{m,p}$

```
double[][] multiplyMatrices(double[][] a, double[][] b) {
    int m = a.length; //number of rows in a
    int n = a[0].length; //number of columns in a
    int p = b[0].length; //number of columns in b

    //number of rows in b should equal number of columns in a
    if (b.length != n) {
      throw new IllegalArgumentException("Incompatible!");
      //this is a runtime exception, we'll cover them later
    }
    //create the product matrix
    double[][] c = new double[m][p]; //all zeros by default
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            for (int k = 0; k < n; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    return c;
}
```

String, StringBuilder

- A `String` is an **imutable object** that represents a sequence of characters. Once created, it cannot change.

```
String hello = "Hello"; //literal
String world = new String("World"); //same thing, but not quite
char[] letters = {'a', 'b', 'c'}; String s = new String(letters);
```

- **+ operator** is overloaded for Strings (no other operator overloading exists or is it possible).

```
String helloWorld = hello + ", " + world;
```

- A `StringBuilder` is used for **mutable strings**. It implements the **Builder** design pattern.

```
String abc = "a" + "b" + "c"; //inefficient, not recommended
var sb = new StringBuilder();
sb.append("a").append("b").append("c");
String abc = sb.toString();
```

Performance comparison: String vs. StringBuilder

Concatenation of Strings with + (or .concat) is much more expensive than using a StringBuilder, because each + creates a new String object. In the example below, the StringBuilder variant is 300 times faster.

```
public class TestStringBuilder {
    int n = 300_000;
    public static void main(String args[]) {
        testStringPlus(); testStringBuilder();
    }
    private static void testStringPlus() {
        long t0 = System.currentTimeMillis();
        String s = "";          //or s = s.concat("a");
        for (int i = 0; i < n; i++) s += "a";
        System.out.println(System.currentTimeMillis() - t0); //4500ms
    }
    private static void testStringBuilder() {
        long t0 = System.currentTimeMillis();
        var s = new StringBuilder();
        for (int i = 0; i < n; i++) s.append("a");
        System.out.println(System.currentTimeMillis() - t0); //15ms
    }
}
```

Equality Testing

- **Primitive equality: ==**

```
int x = 123; int y = 123;
System.out.println(x == y); //true
```

- **Object equality: .equals**

```
String s1 = new String("abc");
String s2 = new String("abc");
System.out.println(s1 == s2);       // false, different objects
                   (s1.equals(s2)); // true, same content
                   ("abc" == "abc"); // true, String interning
                   (s1.intern() == s2.intern()); //true
```

- **Array equality: Arrays.equals, .deepEquals**

```
int a[] = {1, 2, 3}; int b[] = {1, 2, 3};
System.out.println(a == b);       // false, different objects
                   (a.equals(b)); // false, arrays are "special"
                   (Arrays.equals(a,b)); //true
                   //use .deepEquals for arrays of arrays
//Arrays is a class from java.util package.
```

Primitive Type Conversions

- **Widening**: Larger types are converted to smaller types automatically, since it is a safe operation - no cast needed.

```
byte -> short -> int -> long -> float -> double
        char  ->
double d = 100; //int to double, safe
```

- **Narrowing**: Converting smaller to larger types may cause **data truncation** or **overflow** - explicit cast is required.

```
int n = (int) 123.45; //data truncation
byte b = (byte) 128;  //overflow
```

- **Type promotion in expressions**: In expressions, smaller types are promoted to larger ones, in order to optimize computations.

```
byte, short, char -> int -> long -> float -> double
byte a = 10; byte b = 20;
byte c = a + b; //compilation error
int c = a + b; // byte was promoted to int
```

Boxing / Unboxing

- Each primitive type has a **wrapper** class. This is because some data structures work only with objects and not primitive values.

byte	Byte	float	Float
short	Short	double	Double
int	Integer	char	Character
long	Long	boolean	Boolean

  All wrapper classes are **immutable**.

- **Boxing**: primitive type $\rightarrow$ wrapper object

```
Integer boxed = 100; //usually, no reason to do this (use int)
List<Integer> list = List.of(10, 20, 30);
```

- **Unboxing**: wrapper object $\rightarrow$ primitive type

```
int unboxed = new Integer(123); //deprecated, don't do this
int unboxed = Integer.valueOf(123); //valueOf returns an Integer
int unboxed = list.get(0); //get method actually returns Integer
```

Conversions Between Strings and Numbers

- **String → Number**

```
double pi = Double.parseDouble("3.14");
boolean ok = Boolean.parseBoolean("true");
try {
  int n = Integer.parseInt(someString);
} catch(NumberFormatException e) {
  System.err.println(someString + " is not an integer");
}
```

- **Number → String**

```
String str = Integer.toString(0x7fffffff);
String str = String.valueOf(Integer.MAX_VALUE); //more convenient
String str = String.valueOf(Double.POSITIVE_INFINITY);
```

- **Casting** between strings and numbers is not allowed.

```
String str = 123;   //incompatible types, compilation error
int number = "123"; //incompatible types, compilation error
```

Command-Line Arguments

- Command-line arguments are the **values passed to a program when it starts**. They are received by the JVM.

```
java MainClass "Hello World" 2025
```

- JVM passes the command-line arguments to the **`main`** method, in the main class (the one specified when we launch java):

```
public class MainClass {
    public static void main (String args[]) {
        if (args.length < 2) {
            System.out.println("Not enough arguments!");
            System.exit(-1);
        }
        String name = args[0];
        int year = Integer.parseInt(args[1]); //use a try-catch
    }
}
```

Recap

- Programming language / platform
- Architecure neutrality, Portability, Cross-platform
- Java Development Kit (JDK), Java Runtime Environmnet (JRE)
- Compiled, Interpreted, Bytecode, Java Virtual Machine (JVM)
- Just-In-Time compilation (JIT), HotSpot
- Unicode, UTF-16, UTF-8
- Primitive and Reference data types
- Basic Java syntax, Naming conventions
- Data type conversions, Boxing/Unboxing
- Equality testing for primitives and objects
- Command-line arguments
- Immutable and Builder design patterns