Java File Types

1. `.class` File

- A **compiled** version of a Java source code file.
- Generated when a Java file is compiled using `javac`.
- Contains **bytecode** that is executed by the Java Virtual Machine (JVM).

2. `.war` (Web Application Archive) File

- A **compressed package** containing multiple `.class` files and HTML files.
- Used for **deploying** Java-based web applications.
- Includes **servlets, JSP files, and configuration files**.

3. `.jar` (Java Archive) File

- A **compressed package** containing multiple `.class` files.
- Used to **bundle Java applications or libraries**.
- Can be executed if it includes a `Main-Class` in `MANIFEST.MF`.

Topics of Discussion

Today's Topics

1. **OOP (Basic Introduction)**
2. **Identifiers (Variables)**
    - Needs further work.
3. **Reserved Words**
4. **Data Types and Its Uses**

Yesterday's Topics

1. **JShell**
    - An interactive Java shell for executing Java statements and expressions.
2. **Main Method**
    - `public static void main(String[] args)`
    - Entry point of a Java program.
3. **Command-Line Arguments Execution**
    - Running Java programs using **IDE** and **command prompt**.

Here's a structured summary of your notes on **Object-Oriented Programming (OOP) and Identifiers in Java**:

Object-Oriented Programming (OOP)

- **Definition:** OOP stands for **Object-Oriented Principles**.
- **Example Objects:** `Car`, `Student`
- **Objects in Real Life:**
  - An **object** is a real-world instance of an entity.
  - Every object has **two parts**:
    1. **What it has** (Attributes/Properties)
    2. **What it does** (Methods/Behavior)

Example: Car Object

```
Car {
   brandName;
   noOfWheels;
   model;
   speed;
}
```

- **What it does**:
  - `move`
  - `brake`

Example: Student Class

```
class Student {
   // "What it has" (Attributes)
   String name;
   int id;
   float height;

   // "What it does" (Behavior)
   void mark() {
      // Method logic here
   }
}
```

- **Attributes (Variables):** Name, ID, Height
- **Methods (Behavior):** `mark()`

Identifiers in Java

- **Definition:** A name used in Java programs for:
  - **Class names**
  - **Method names**
  - **Variable names**

Example: Java Identifiers

```
class Main {
   public static void main(String[] args) {
      int x = 10;  // 'x' is an identifier (variable)
   }
}
```

- Java has **totally 4 types of identifiers**:
  - Can be a **class name, method name, variable name, or label name**.
  - Example: `System`, `out`, `println`.

Here's a structured summary of your notes on **Rules for Writing Identifiers in Java**:

Rules for Writing Identifiers in Java

1. **Allowed Characters**
   - Identifiers can only contain:
     - `A-Z`, `a-z`
     - `0-9`
     - `_` (Underscore)
2. **No Special Characters**
   - Any other character is not allowed in Java identifiers.
3. **Cannot Start with a Digit**
   - Identifiers **cannot** begin with numbers (`0-9`).
4. **Case Sensitivity**
   - Java identifiers are **case-sensitive**.
5. **No Length Limit**
   - There is no restriction on identifier length.
6. **Reserved Words Restriction**
   - Java **reserved words** (e.g., `int`, `class`, `static`) **cannot** be used as identifiers.
7. **Predefined Class Names Usage**
   - **Predefined class names** (e.g., `String`) **can** be used as identifiers but should be **avoided**.

Example: Using a Predefined Class Name as an Identifier

int String = 10;

System.out.println(String);

Output:

10

⚠️ **Note:** Although Java allows predefined class names as identifiers, **it is not a good practice** to do so.

Here's a structured summary of your notes on **Reserved Words in Java**:

What is a Reserved Word?

- A **reserved word** is a **built-in word/keyword** in Java that has a **predefined meaning** and **cannot be used as an identifier**.

Classification of Reserved Words in Java

- **Total Reserved Words: 53**
  - **Keywords: 50**
  - **Reserved Literals: 3**

1. Keywords (50)

- **Categories of Keywords:**

  - **Data Type Keywords** (e.g., `int`, `double`, `char`)
  - **Flow Control Keywords** (e.g., `if`, `else`, `switch`, `while`, `for`)
  - **OOP Keywords** (e.g., `class`, `interface`, `extends`, `implements`)
  - **Modifiers** (e.g., `public`, `private`, `static`, `final`)
  - **Class-related Keywords** (e.g., `new`, `instanceof`, `super`, `this`)
  - **Object-related Keywords** (e.g., `import`, `package`)
  - **Exception Handling Keywords** (e.g., `try`, `catch`, `throw`, `throws`, `finally`)

- **Unused Keywords (2)**
    - `goto`
    - `const`
      (These are reserved but not used in Java.)

2. Reserved Literals (3)

- `true` (Boolean literal)
- `false` (Boolean literal)
- `null` (Represents the absence of an object)

Important Notes

- `void` is a **keyword associated with methods** (it specifies that a method does not return a value).
- Java is a **statically typed language**, meaning variables must have a **declared data type**.

Example Code

```
class Test {

   public static void main(String[] args) {

      int data = 10; // Statically typed language

      System.out.println(data);

   }

}
```

- **Literal:** `10` is a literal value assigned to the variable.

Here's a structured summary of your notes on **Data Types in Java**:

Compiler and Data Type Checking

- The **compiler** checks whether the **stored value** can be handled by the **data type** or not.
- This **checking is done by the compiler** at compile time.

Boolean Data Type Rules

- The **only values allowed** for a **boolean variable** are:
  - `true`
  - `false`
- **Any other value** will result in a **compiler error**.

Java Data Types Overview

- **Every variable has a type.**
- **Every expression has a type.**
- **Java is statically typed** → Type checking is done at compile time.

Primitive Data Types

Primitive data types are commonly classified into:

1. **Numeric values** (Integer & Floating-point numbers)
2. **Character values** (`char` type)

Integer Data Types in Java

Used to **store whole numbers**:

- `byte`
- `short`
- `int`
- `long`

Data Type	Size	Minimum Value	Maximum Value
byte	1 byte (8 bits)	-128	127

short	2 bytes (16 bits)	-32,768	32,767
int	4 bytes (32 bits)	-2B	+2B
long	8 bytes (64 bits)	Very large	Very large

Key Notes

- **byte** is the smallest integer type, used for **memory-efficient storage**.
- **short** is rarely used.
- **int** is the **default integer type** in Java.
- **long** is used for **very large numbers**.

Here's an improved and well-structured version of your notes:

Java Data Types – Key Notes

1. Interview Question

byte a = true; // Compilation Error: Incompatible type

- `byte` cannot store `true` or `false`, as it is **not a boolean type**.

---

2. When to Use `byte` Data Type?

- Commonly used when handling **data coming from streams or networks**.
- Example: **Streams** → **`java.io` package** (used for reading/writing binary data efficiently).

---

3. `short` Data Type

- **Not commonly used in Java.**

- Mostly relevant for **old processors (like 8086)** but rarely used in modern Java applications.

---

4. `long` Data Type

- Used when `int` is **not enough** to hold large values.

Example:
long a = 12L;

- (`L` or `l` is required to indicate a `long` literal.)

Use Cases:

- **Network operations** → Typically deal with **bytes**.
- **Normal calculations** → Use `int`.
- **File handling** → Files with large sizes require `long`.
- **Handling large files** in Java programs (e.g., **GBs of data**).

---

5. Floating-Point Numbers (`float` and `double`)

5.1 `float` Data Type

Example:
float f = 10.5f;

-
- **Size:** 4 bytes (32 bits)
- **Range:** ~ ±2B (±2 * 10^9)

Important Note:

- By **default**, any decimal number (`10.5`, `20.3`, etc.) is treated as `double`** by the compiler.
- To **store it as `float`**, suffix it with **`f` or `F`**.

Here's an improved and structured version of your notes:

Java Data Types – Advanced Notes

1. `double` Data Type

- **Size:** 64 bits (8 bytes)
- **Range:**
  - **Minimum:** `4.9E-324`
  - **Maximum:** `1.79E+308`

Note:

- **Data types in Java are reserved words** and must be written in **lowercase** (`int`, `double`, `char`, etc.).
- Unlike **C/C++**, Java does not allow type aliases.

---

2. Wrapper Classes (JDK 1.5+)

- Java introduced **Wrapper Classes** to represent primitive data types as objects.
- This is useful for working with **collections, serialization, and generics**.

Primitive Data Type → Corresponding Wrapper Class

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`

`boolean` `Boolean`

---

3. Primitive Data Types

Character Data Type (`char`)

Example:
char letter = 'A';

char symbol = '#';

char unicodeChar = '\u20AC'; // Unicode for €

-
- **Stores a single character** (letter, digit, or symbol).
- Uses **2 bytes (16 bits)** to support **Unicode characters**.

---

4. Understanding Data Representation

- Data in memory is stored in **binary (0s and 1s).**

Example of binary representation:

1011 1001 (binary) = Decimal equivalent

-

Example:

int a = 10; // Stored in memory as binary

Here's a structured and refined version of your notes:

Character Encoding Systems in Java

1. Binary Representation

- Characters are represented using binary codes.

Example:

A → 00

B → 01

C → 10

D → 11

-

Binary powers:

$2^2 = 4$

$2^3 = 8$

-

---

2. ASCII (American Standard Code for Information Interchange)

- **Total characters:** 128
- **Representation:** Uses **7 bits** (0–127)
- **Encodes:** English letters, digits, punctuation, and control characters.

Example:
A → 65

a → 97

Z → 90

z → 122

-

---

3. Unicode and UTF Encoding

- **Unicode** is a universal encoding standard that supports multiple languages (English, Hindi, Chinese, etc.).
- **Total Characters Supported:** 65,536 (2^{16})
- **Encoding Size:**
  - Uses **16 bits (2 bytes)** to store each character.
  - Ensures uniformity across different platforms.

Key Points:

- **UTF (Unicode Transformation Format) → 2 bytes per character.**
- **Java uses UTF (2 bytes per character).**

---

4. Character Encoding Comparisons

Encoding System	Bits Used	Number of Characters Supported	Usage
ASCII	7 bits	128	English letters, digits, symbols
Extended ASCII	8 bits	256	Includes additional symbols
Unicode (UTF-16)	16 bits	65,536	Supports multiple languages
UTF-8	Variable (1-4 bytes)	Over a million	Web and international applications

---

5. Geek Table (Decimal Representation)

Character	Decimal Representation
A	65
B	66

C	67
...	...
Z	90

Here's a structured and refined version of your notes:

Character Handling in Java

1. Java and Unicode

- Java uses **Unicode** for character encoding.
- **UTF (Unicode Transformation Format)** is used.
- `char` **in Java → 2 bytes (16 bits)** per character.

---

2. Syntax of Character Declaration

Syntax	Description	Validity
`char a = 'A';`	Single character enclosed in **single quotes (')**	✅ Valid
`char a = "A";`	Uses **double quotes ("")**, which are for strings	❌ Invalid
`char a = 'AB';`	More than one character in single quotes	❌ Invalid

---

3. `char` and `Character` Class

- `char` → **Primitive Data Type**
- `Character` → **Wrapper Class for** `char`

---

4. Primitive Data Types in Java

Categorization:

1. **Integer Types**: `byte`, `short`, `int`, `long` → **Base 2 (Binary)**
2. **Floating-Point Types**: `float`, `double`
3. **Boolean Type**: `boolean` (Stores `true` or `false`)
4. **Character Type**: `char` (Uses **Unicode** encoding)

---

5. Strings in Java

- **String** is an **array of characters**.
- **String is an Object**, not a primitive type.

Example:
String str = "Hello";

-

---

6. Primitives vs Wrapper Classes

- **Primitive Types** are **stored in stack memory**.
- **Wrapper Classes** (e.g., `Integer`, `Character`, `Double`) **wrap primitive types into objects**.

Here's a structured version of your notes on **Type Casting & Rounding in Java**:

Type Casting & Numeric Promotion in Java

1. Integer Division & Rounding

int a = 25;

int b = 2;

int c = a / b;

System.out.println(c); // Output: 12

- Since both a and b are **integers**, division truncates the decimal part.
- The result is **rounded down** to the nearest integer.

---

2. Implicit Type Casting (Widening)

- Happens **automatically** when a smaller data type is assigned to a larger data type.
- **No data loss occurs.**

Example:

byte a = 45;

double b;

b = a; // Implicit type casting

System.out.println(b); // Output: 45.0

Concept:
📦 `byte` (1 byte) → `double` (8 bytes) ✅ Allowed

---

3. Explicit Type Casting (Narrowing)

- **Manually converting** a larger data type into a smaller one.
- **May result in data loss** due to precision reduction.

Example:

double a = 45.5;

byte b;

b = (byte) a; // Explicit type casting

System.out.println(b); // Output: 45 (fraction lost)

Concept:
🛑 `double` (8 bytes) → `byte` (1 byte) ❌ Precision loss

---

4. Numeric Type Promotion

- When performing arithmetic operations, **smaller types are promoted** to larger types automatically.
- Example:

int a = 5;

int result = a + 10; // a promoted to int

System.out.println(result); // Output: 15

Key Takeaways

1. **Implicit Casting** (Widening) → No data loss, done automatically.
2. **Explicit Casting** (Narrowing) → Might cause precision loss, needs manual conversion.
3. **Integer Division** → Truncates decimal part (use float/double for accurate results).
4. **Numeric Type Promotion** → Smaller types automatically promoted in expressions.

Here's an **improved** and more **structured version** of your notes with **better formatting, explanations, and a cleaner truth table.** 🚀

📌 Increment & Decrement Operators in Java

1️⃣ Increment Operators (++)

These operators **increase** a variable's value by **1**.

🔹 **Types:**

- **Post-increment (a++)** → Uses value first, then increments.
- **Pre-increment (++a)** → Increments first, then uses value.

Example:

int a = 5;

System.out.println(a++); // Output: 5 (Uses value first)

System.out.println(a); // Output: 6 (Incremented now)

int b = 5;

System.out.println(++b); // Output: 6 (Increments first, then prints)

System.out.println(b); // Output: 6

📝 Explanation (Step-by-Step)

Expression	Value of `a`	Printed Output
`int a = 5;`	5	-
`System.out.println(a++);`	5 → 6	5
`System.out.println(a);`	6	6
`System.out.println(++b);`	6	6

2️⃣ Decrement Operators (`--`)

These operators **decrease** a variable's value by **1**.

🔹 **Types:**

- **Post-decrement (`a--`)** → Uses value first, then decrements.
- **Pre-decrement (`--a`)** → Decrements first, then uses value.

Example:

int a = 5;

System.out.println(a--); // Output: 5 (Uses value first)

System.out.println(a); // Output: 4 (Decremented now)

int b = 5;

System.out.println(--b); // Output: 4 (Decrements first, then prints)

System.out.println(b); // Output: 4

📝 Explanation (Step-by-Step)

Expression	Value of `a`	Printed Output
`int a = 5;`	5	-
`System.out.println(a--);`	5 → 4	5
`System.out.println(a);`	4	4
`System.out.println(--b);`	4	4

📌 Data Types & Memory Size in Java

Data Type	Size (Bytes)	Range
`byte`	1 byte	-128 to 127
`short`	2 bytes	-32,768 to 32,767
`int`	4 bytes	-2^{31} to $2^{31}-1$
`long`	8 bytes	-2^{63} to $2^{63}-1$
`float`	4 bytes	Approx. 6-7 decimal digits
`double`	8 bytes	Approx. 15 decimal digits
`char`	2 bytes	Unicode Characters

📌 Operators in Java

1️⃣ Arithmetic Operators

These operators perform basic **mathematical operations**.

Operator	Meaning	Example

`+`	Addition	`a + b`
`-`	Subtraction	`a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	Modulus (Remainder)	`a % b`

2️⃣ Logical Operators

These operators work with **Boolean values** (`true` or `false`).

Operator	Meaning	Example
`&&`	Logical AND	`(a > 5 && b < 10)`
`		`
`!`	Logical NOT	`!(a > 5)`

🧮 Truth Table for Logical Operators

A	B	A && B	•	A || B	!A
T	T	T		T	F
T	F	F		T	F
F	T	F		T	T
F	F	F		F	T

📌 Key Takeaways

✅ **Increment (`++`) and Decrement (`--`)** change values by **1**.
✅ **Pre (`++a`) vs. Post (`a++`)** determines when the value is updated.
✅ **Logical operators (`&&`, `||`, `!`)** follow Boolean algebra rules.
✅ **Arithmetic operators (`+`, `-`, `*`, `/`, `%`)** perform basic math.

Here's an improved and well-structured version of your notes with better formatting and explanations:

📌 Java Operators and Conditional Statements

1️⃣ Relational Operators

Relational operators are used to compare two values and return a boolean (`true` or `false`).

Operator	Meaning	Example (`a = 7, b = 6`)	Output

`>`	Greater than	`a > b`	`true`
`<`	Less than	`a < b`	`false`
`>=`	Greater than or equal to	`a >= b`	`true`
`<=`	Less than or equal to	`a <= b`	`false`
`==`	Equal to	`a == b`	`false`
`!=`	Not equal to	`a != b`	`true`

2️⃣ Compound Assignment Operators

These operators combine arithmetic operations with assignment.

Operator	Meaning	Equivalent To
`+=`	Add and assign	`a += 10` → `a = a + 10`
`-=`	Subtract and assign	`a -= 5` → `a = a - 5`
`*=`	Multiply and assign	`a *= 3` → `a = a * 3`

`/=`	Divide and assign	`a /= 2 → a = a / 2`
`%=`	Modulus and assign	`a %= 4 → a = a % 4`

Example:

int a = 10;

a += 5; // a = a + 5 (Now a = 15)

System.out.println(a);

3️⃣ Assignment Operators

Assignment operators assign values to variables.

Operator	Meaning	Example
`=`	Assign	`a = 10;`
`+=`	Add and assign	`a += 5;`
`-=`	Subtract and assign	`a -= 2;`

Unary vs Binary Operators

- **Unary Operator:** Requires only **one operand**
  ✅ Example: `a++`, `b--`
- **Binary Operator:** Requires **two operands**
  ✅ Example: `c = a + b`

Example:

```
int a = 10;  // Unary operation (one operand)

int b = 5, c;

c = a + b;   // Binary operation (two operands)
```

4️⃣ Conditional Statements

Conditional statements allow us to execute code based on conditions.

Syntax of if-else:

```
if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}
```

Example:

```
int a = 10, b = 20;

if (a > b) {
    System.out.println("a is greater");
} else {
    System.out.println("b is greater");
}
```

📌 **Output:** `b is greater`

Using `else if` for Multiple Conditions:

```
if (a > b) {
    System.out.println("a is greater");
} else if (a == b) {
    System.out.println("a and b are equal");
} else {
    System.out.println("b is greater");
}
```

Here's an improved and well-structured version of your notes with better formatting and explanations:

📌 Java Control Structures & Operators

1️⃣ Nested Loops

A **nested loop** is a loop inside another loop.

Syntax:

```
for (int i = 1; i <= 3; i++) { // Outer loop
    for (int j = 1; j <= 3; j++) { // Inner loop
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

📌 **Explanation:**

- The **outer loop** runs first.
- The **inner loop** executes completely for each iteration of the outer loop.

2️⃣ Ternary Operator (`? :`)

The ternary operator is a shorthand for `if-else`.

Syntax:

variable = (condition) ? value_if_true : value_if_false;

Example:

int a = 100, b = 20;

int max = (a > b) ? a : b; // If a > b, assign 'a' to max, else assign 'b'

System.out.println("Maximum: " + max);

📌 **Output:** `Maximum: 100`

Nested Ternary Operator

int a = 10, b = 20, c = 30;

int max = (a > b) ? (a > c ? a : c) : (b > c ? b : c);

System.out.println("Maximum: " + max);

📌 **Output:** `Maximum: 30`

3️⃣ Switch Case

The `switch` statement is used when we have multiple conditions to check.

Syntax:

switch (variable) {

```
    case value1:
        // Code to execute if variable == value1
        break;
    case value2:
        // Code to execute if variable == value2
        break;
    default:
        // Code to execute if none of the cases match
}
```

Example 1: Without `break` (Fall-through behavior)

```
int day = 2;
switch (day) {
    case 1:
        System.out.println("Monday");
    case 2:
        System.out.println("Tuesday");
    case 3:
        System.out.println("Wednesday");
    default:
        System.out.println("Invalid day");
}
```

📌 **Output:**

```
Tuesday
Wednesday
```

Invalid day

(Since there is no `break`, it falls through all cases)

Example 2: With `break` (Correct usage)

```
int choice = 2;

switch (choice) {
    case 1:
        System.out.println("Option 1 selected");
        break;
    case 2:
        System.out.println("Option 2 selected");
        break;
    default:
        System.out.println("Invalid choice");
}
```

📌 **Output:** `Option 2 selected`

Key Points:

✅ **Nested Loops** allow iteration inside another loop.
✅ **Ternary Operator** simplifies `if-else` statements.
✅ **Switch Case** is useful for handling multiple conditions efficiently.
✅ Always use `break` in a switch case to avoid fall-through.

Here's an improved and well-structured version of your notes with better formatting and explanations:

📌 Java Control Structures & Operators

1️⃣ Nested Loops

A **nested loop** is a loop inside another loop.

Syntax:

```
for (int i = 1; i <= 3; i++) { // Outer loop
    for (int j = 1; j <= 3; j++) { // Inner loop
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

📌 **Explanation:**

- The **outer loop** runs first.
- The **inner loop** executes completely for each iteration of the outer loop.

2️⃣ Ternary Operator (`? :`)

The ternary operator is a shorthand for `if-else`.

Syntax:

```
variable = (condition) ? value_if_true : value_if_false;
```

Example:

```
int a = 100, b = 20;
int max = (a > b) ? a : b; // If a > b, assign 'a' to max, else assign 'b'
System.out.println("Maximum: " + max);
```

📌 **Output:** `Maximum: 100`

Nested Ternary Operator

```
int a = 10, b = 20, c = 30;

int max = (a > b) ? (a > c ? a : c) : (b > c ? b : c);

System.out.println("Maximum: " + max);
```

📌 **Output:** `Maximum: 30`

3️⃣ Switch Case

The `switch` statement is used when we have multiple conditions to check.

Syntax:

```
switch (variable) {
    case value1:
        // Code to execute if variable == value1
        break;
    case value2:
        // Code to execute if variable == value2
        break;
    default:
        // Code to execute if none of the cases match
}
```

Example 1: Without `break` (Fall-through behavior)

```
int day = 2;
```

```
switch (day) {
    case 1:
        System.out.println("Monday");
    case 2:
        System.out.println("Tuesday");
    case 3:
        System.out.println("Wednesday");
    default:
        System.out.println("Invalid day");
}
```

📌 **Output:**

```
Tuesday
Wednesday
Invalid day
```

(Since there is no `break`, it falls through all cases)

Example 2: With `break` (Correct usage)

```
int choice = 2;
switch (choice) {
    case 1:
        System.out.println("Option 1 selected");
        break;
    case 2:
        System.out.println("Option 2 selected");
```

```
        break;
    default:
        System.out.println("Invalid choice");
}
```

📌 **Output:** `Option 2 selected`

Key Points:

✅ **Nested Loops** allow iteration inside another loop.
✅ **Ternary Operator** simplifies `if-else` statements.
✅ **Switch Case** is useful for handling multiple conditions efficiently.
✅ Always use `break` in a switch case to avoid fall-through.

Let me know if you need **more examples or explanations!** 🚀