To run java program:

1. Navigate to the folder containing the file:
2. Compile the java file : javac Ex1.java // this create Ex1.class in same folder
3. Run the java program : java Ex1
4. Output will get
5. javac Ex1.java && java Ex1 (one liner command)

==========================================================================================

**Topic : Variables,Data types,Operators,Control**

**Variables:**

A variable is a name that refers to a memory location where data is stored.

It holds values that can change during program execution.

Example:

int age = 25;

String name = "Om";

primitive Variables: These variables store actual values like numbers, characters, or boolean.

Stored directly in: Stack memory

Example :

int age = 25; // age is store in stack

char grade = 'A';

boolean isJavaFun = true;

Reference Variables: These store the memory address (reference) of objects — not the actual data.

Reference (address) is stored in Stack

Actual object is in the Heap

Example :

String name = "Om"; // name is store in stack and om is store in heap

int[] marks = {80, 85, 90};

Student s1 = new Student();

Any class create // custom class

======================================================

public class Main {

public static void main(String[] args) {

int num = 100; // Primitive

String name = "Om Kumar"; // Reference

System.out.println(num); // Prints value directly

System.out.println(name); // Prints object from heap

}

}

======================================================================================================

Primitive Types Go to the Stack:-

The value is stored directly in the stack, because it’s simple and lightweight.

Stack is fast

They do not create objects.

They are usually small in size.

Stack memory is efficient for short-term, simple values.

Reference Types Use Both Stack and Heap:-

for example: Student s1 = new Student();

s1(ref) store in stack and new Student() creates an object → lives in the heap

objects can be large and shared → stored in heap

References are small → stored in stack for fast access

Heap memory is longer-lived (objects stay until garbage collected)

Based on the purpose and position of declaration all variable are divided into 3 types

**Instance variable:** if the value of varied from object to object called Instance variable.

* for every object a seperate copy of instance variable is created.
* Scope of instance variable same as scope of object.
* instance variable store on the heap as the part of object.
* Declared with in the class but outside of any method or block or constructor.
* Can't be directly accessed from static area but with the help of object reference it done
* It always provide values, not required to platform initialization JVM

Example :

class Test{

int i=10;

public static void main(String[] arg)

{

System.out.println(i); // Error: as non static variable can't refernce

Test t1 = new Test();

System.out.println(t.i); // valid , here object is created then call

t.method();

}

public void method()

{

System.out.println(i); // valid , as it call from non static block

}

=============================================================

**Static variable** :

A static variable is a **class-level variable**, shared by **all objects** of that class.  
 It's declared with the static keyword.

* if the value of a variable is not varied from object to object such variable called Static
* only one copy is created to entire class and shared by every object of class
* Scope of variable same as scope of .class file.
* Static vaiables will be stored in method area, it declared inside the class but outside of any method or block or constructor.
* It can be access by classname or by object reference

Example :

class Test {

static int i = 10;

public static void main(String[] args)

{

Test t1 = new Test();

System.out.println(t1.i); // valid : by creating object(not good for production)

System.out.println(Test.i); // valid : by call class name

System.out.println(i); // direct call as static call static

}

}

=======================================================

class Test {

static String s;

public static void main(String[] args)

{

System.out.println(s); // null (default value)

}

}

========================================================

class Test {

int x =10;

static int y = 20;

public static void main(String[] args)

{

Test t1 = new Test();

t1.x = 888;

t1.y = 999; //Even though accessed via object, it changes the **shared static** variable y to 999

Test t2 = new Test(); // still x =10;

System.out.println(t2.x+ "-------------" + t2.y); ans : 10-------999

}

} // here instance var depend upon obj as t2 not change , static var depends on class

**Local variables**

* Declared **inside methods, constructors, or blocks**
* **Created** when the method is called and it is final
* **Destroyed** after the method ends
* **Not accessible** outside the method/block
* **Must be initialized** before use and store in stack
* Highly recommended to perform initialization at time of declaration.

Example :

=====================================================

class Demo {

public static void main(String[] args) {

int a;

System.out.println(a); // Compile error: variable a might not have been initialized

}

}

====================================================

class Test {

public static void main(String[] args) {

int x;

if (args.length > 0) { // cammand line

x = 10;

}

System.out.println(x); // Compilation error

}

}

=========================================

int x;

if (args.length > 0) {

x = 10;

} else {

x = 0;

}

System.out.println(x); // depend on user input it gives result

as if :java Test --> 0 and if java Test hii --> 10;

========================================================================

**Data Type:** **Data types** specify the size and type of values that can be stored in variables

**1. Primitive Data Types (8 types)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Description** | **Size** | **Example** |
| **byte** | Integer (small) | 1 byte | byte a = 10; |
| **short** | Integer (medium) | 2 bytes | short s = 1000; |
| **int** | Integer (default) | 4 bytes | int x = 50000; |
| **long** | Large integer | 8 bytes | long l = 10000000000L; |
| **float** | Decimal (single precision) | 4 bytes | float f = 5.6f; |
| **double** | Decimal (high precision) | 8 bytes | double d = 10.5; |
| **char** | Single character | 2 bytes | char c = 'A'; |
| **boolean** | True or false | 1 bit | boolean flag = true; |

**2. Non-Primitive Data Types (Reference Types)**

These are created by the programmer or Java itself.

|  |  |
| --- | --- |
| **Type** | **Example** |
| **String** | String name = "Om"; |
| **Arrays** | int[] arr = {1, 2, 3}; |
| **Classes** | class Student {} |
| **Interfaces** | interface Drawable {} |
| **Objects** | Student s = new Student(); |

**🔸**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Primitive Data Types** | **Non-Primitive Data Types** |
| **Definition** | Basic, built-in data types | Reference types created using classes/objects |
| **Examples** | int, float, char, boolean, etc. | String, Array, Class, Interface, Object |
| **Stores** | Actual value | Reference (memory address of object) |
| **Memory Location** | Stored in **stack memory** | Object is in **heap**, reference is in **stack** |
| **Size** | Fixed (known at compile time) | Varies depending on the object's structure |
| **Access Speed** | Fast (since values are directly accessible) | Slower (due to reference and heap access) |
| **Null Allowed?** | Cannot be null | Can be null |
| **Default Values** | Example: int → 0, boolean → false | Example: String → null, Array → null |
| **Mutable?** | Immutable (value change replaces value) | Mostly mutable (e.g., arrays, objects can change) |
| **Inheritance** | Cannot inherit | Supports inheritance, polymorphism |

**Primitive Types**

* Faster access and performance
* Low memory usage
* Easy to use for simple operations
* Not flexible (cannot store complex data)
* Cannot use OOP features (like methods, inheritance)

**Non-Primitive Types**

* Can store complex data (arrays, objects)
* Can use OOP features (methods, inheritance, polymorphism)
* Highly customizable
* Slower access due to references
* More memory consumption
* Slightly complex syntax

=====================================================================

**Operators** **:** Operators are special symbols used to perform operations on variables and value

**Types of Operators in Java**

**1. Arithmetic Operators**

Used to perform basic mathematical operations.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Result** |
| + | Addition | 10 + 5 | 15 |
| - | Subtraction | 10 - 5 | 5 |
| \* | Multiplication | 10 \* 5 | 50 |
| / | Division | 10 / 2 | 5 |
| % | Modulus (remainder) | 10 % 3 | 1 |

**2. Relational / Comparison Operators**

Used to compare two values.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Result** |
| == | Equal to | 5 == 5 | true |
| != | Not equal to | 5 != 3 | true |
| > | Greater than | 10 > 5 | true |
| < | Less than | 10 < 5 | false |
| >= | Greater than or equal | 5 >= 5 | true |
| <= | Less than or equal | 4 <= 3 | false |

**3. Logical Operators**

Used to perform logical operations, often in conditions.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Result** |
| && | Logical AND | true && false | false |
| ` |  | ` | Logical OR |
| ! | Logical NOT | !true | false |

**4. Assignment Operators**

Used to assign values to variables.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Equivalent To** |
| = | Assign value | x = 10 | — |
| += | Add and assign | x += 5 | x = x + 5 |
| -= | Subtract and assign | x -= 2 | x = x - 2 |
| \*= | Multiply and assign | x \*= 3 | x = x \* 3 |
| /= | Divide and assign | x /= 2 | x = x / 2 |
| %= | Modulus and assign | x %= 2 | x = x % 2 |

**5. Unary Operators**

Work on a single operand.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Result** |
| + | Unary plus (positive) | +a | a |
| - | Unary minus (negation) | -a | Negative a |
| ++ | Increment (prefix/postfix) | ++a or a++ | a + 1 |
| -- | Decrement (prefix/postfix) | --a or a-- | a - 1 |
| ! | Logical NOT | !true | false |

**6. Bitwise Operators**

Operate at the bit level (used in low-level programming).

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Result (in bits)** |
| & | Bitwise AND | 5 & 3 | 0001 (1) |
| ` | ` | Bitwise OR | `5 |
| ^ | Bitwise XOR | 5 ^ 3 | 0110 (6) |
| ~ | Bitwise Complement | ~5 | -6 |
| << | Left shift | 5 << 1 | 10 |
| >> | Right shift | 5 >> 1 | 2 |

**7. Ternary Operator**

Shorthand for if-else.

int a = 10, b = 20;

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

|  |  |
| --- | --- |
| **Syntax** | **Meaning** |
| condition ? true\_value : false\_value | If condition is true, return true\_value; else false\_value |

**8. Instanceof Operator**

Checks if an object is an instance of a specific class.

String s = "Hello";

System.out.println(s instanceof String); // true

**9. Type Cast Operator**

Used to convert one type into another explicitly.

double d = 10.5;

int x = (int) d; // x = 10

|  |  |
| --- | --- |
|  | **Control Statements** |

**1. Conditional Statements (Decision-making)**

These help the program take decisions based on conditions (like “if this, then that”).

**a) if statement**

Used to run a block of code **only if a condition is true**.

if (marks > 40) {

System.out.println("Passed");

}

**b) if-else statement**

Adds an **alternative block** if the condition is false.

if (marks >= 40) {

System.out.println("Passed");

} else {

System.out.println("Failed");

}

**c) if-else-if ladder**

Used to check **multiple conditions one after another**.

if (marks >= 90) {

System.out.println("Grade A");

} else if (marks >= 75) {

System.out.println("Grade B");

} else {

System.out.println("Needs Improvement");

}

**d) switch statement**

Best when you have to compare **one variable with many fixed values**.

int day = 2;

switch (day) {

case 1: System.out.println("Monday"); break;

case 2: System.out.println("Tuesday"); break;

default: System.out.println("Other Day");

}

**🔸 2. Looping Statements (Repeating things)**

These help us run the same block of code **multiple times**.

**a) for loop**

Used when you **know how many times** you want to repeat.

for (int i = 1; i <= 5; i++) {

System.out.println(i); // prints 1 to 5

}

**b) while loop**

Used when you **don't know how many times** to repeat — runs as long as the condition is true.

int i = 1;

while (i <= 5) {

System.out.println(i);

i++;

}

**c) do-while loop**

Same as while, but it **runs at least once** even if the condition is false.

int i = 1;

do {

System.out.println(i);

i++;

} while (i <= 5);

**🔸 3. Jump Statements (Control movement inside loops)**

**a) break**

Used to **exit the loop or switch** early.

for (int i = 1; i <= 5; i++) {

if (i == 3) break;

System.out.println(i); // prints 1, 2

}

**b) continue**

Skips the current iteration and **jumps to the next loop cycle**.

for (int i = 1; i <= 5; i++) {

if (i == 3) continue;

System.out.println(i); // prints 1, 2, 4, 5

}

**c) return**

Exits from the **current method** immediately.

public static void show() {

System.out.println("Start");

return;

// This line won't run

System.out.println("End");

}