CHAPTER NO:01
INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING
{ 6 MARKS}

- 1.1 Creation of java, java byte code, java characteristics
- 1.2 Abstraction, OOP Principles. -Encapsulation, Inheritance and Polymorphism
- 1.3 Constant, Variables and Data Types, Type casting
- 1.4 Operator and Expression, Arithmetic Operators, Relational Operators, Logical Operators, Assignment Operator, Increment and Decrement Operator, Bit wise Operator, Special Operator
- 1.5 Decision making with simple if, if… else, else if ladder statements, The switch statement, The conditional operator
- 1.6 Decision Making with Loops i.e. while, do and for statement, Jumps in Loops, Labelled Loops

1.1 Creation of java, java byte code, java characteristics

Java is a high-level, object-oriented programming language that was originally developed by **James Gosling** and his team at **Sun Microsystems** in the early 1990s. The language was officially released in **1995**.

- **Key Points in Java’s Development:**
- **1991:** Project initiated as the "Green Project" for programming home appliances.
- **1995:** Renamed to **Java** and released to the public.
- **2009:** Sun Microsystems acquired by **Oracle Corporation**, which now maintains Java.

Design Goals of Java:

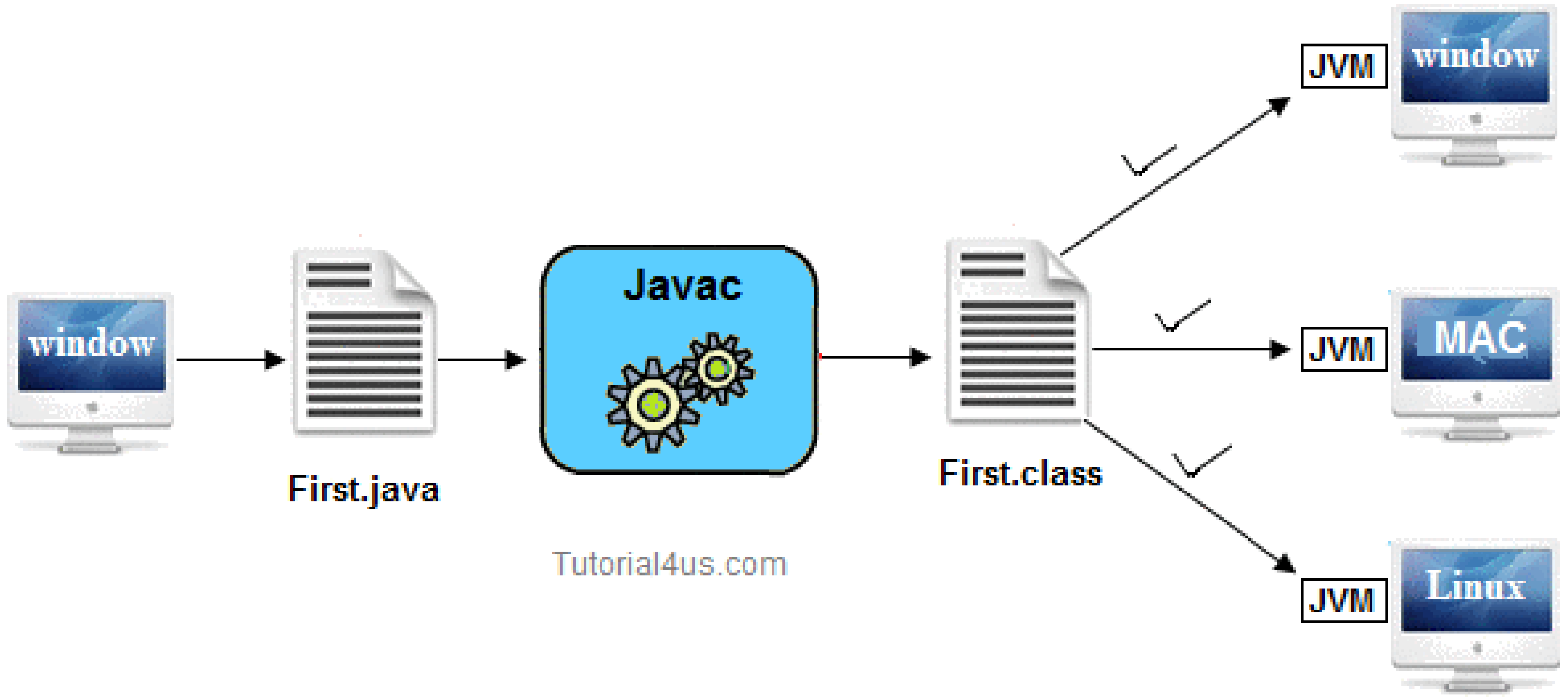
- Simple and familiar
- Object-oriented
- Platform-independent
- Secure and robust
- Architecture-Neutral:
- High performance (with Just-In-Time compilers)
- Multithreaded and dynamic

Java Byte Code:

Java programs are **not** compiled directly into machine code (like C/C++). Instead, they are compiled into an intermediate form called **Java Byte Code**.

What is Java Byte Code?

Java Byte Code is a platform-independent, low-level code generated by the Java compiler (javac) from .java source files.

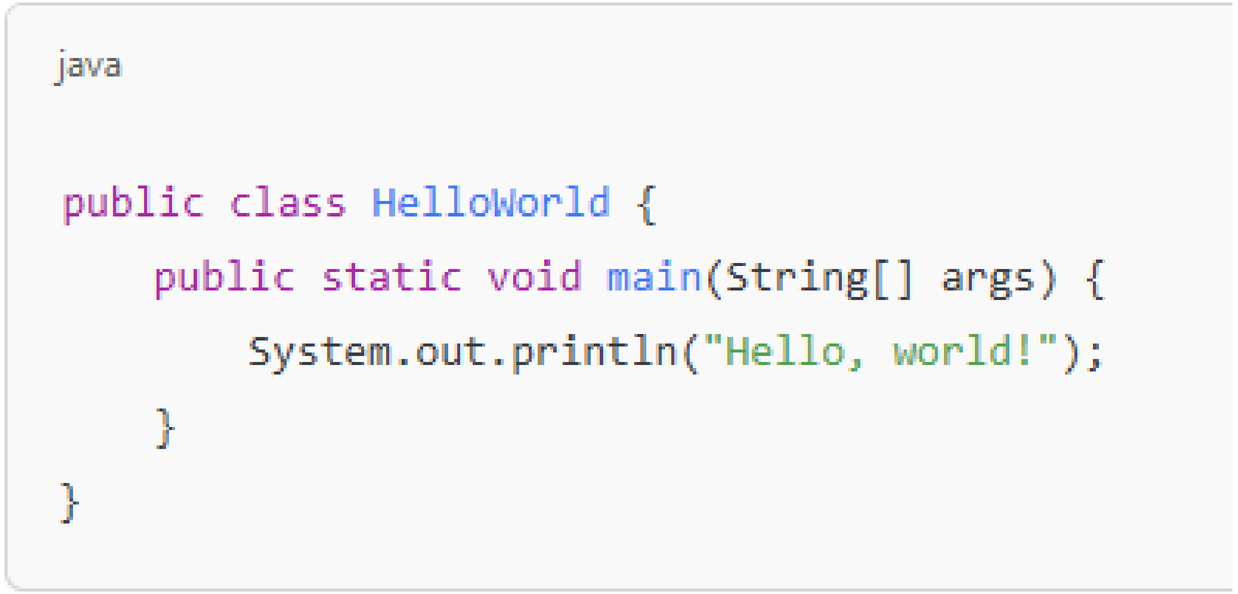

window
First.java
Javac
First.class
JVM
window
JVM
MAC
JVM
Linux
Tutorial4us.com

This diagram illustrates the **platform independence of Java**:

1. A Java program (`File1.java`) is written on a Windows system.
2. It is compiled using the **Java Compiler (`javac`)** into bytecode (`File1.class`).
3. This **bytecode** is **platform-independent** and can be run on any system (Windows, Mac, Linux) that has a **Java Virtual Machine (JVM)**.
4. The **JVM on each OS** interprets the bytecode and executes it, enabling **"Write Once, Run Anywhere"** capability.

Java Program Lifecycle:

1. **Write the source code:**



java

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

File name: `HelloWorld.java`

2. **Compile the source code:**

   ```bash
   javac HelloWorld.java
   ```

   This generates a file called `HelloWorld.class` containing byte code.

3. **Execute the byte code using JVM:**

   ```bash
   java HelloWorld
   ```

   The **JVM** loads the byte code from `HelloWorld.class`, interprets or compiles it into machine code, and runs it.

Advantages of Java Byte Code:

- **Platform Independence:** "Write once, run anywhere"—byte code can run on any device with a JVM.
- **Security:**
  - Byte code runs in a controlled environment (sandbox model / A **sandbox** is like a **safe area** where code can run, but it **can’t harm the system**.).
  - If you download a Java applet from the internet, it runs in the JVM sandbox and cannot access your files or personal data.

Performance:

- When you run Java bytecode, the **Java Virtual Machine (JVM)** doesn’t just interpret it line by line slowly. Instead, it uses a special feature called the **Just-In-Time (JIT) compiler**.
- It **translates bytecode into machine code** (the language your computer’s processor understands) **while the program is running**.
- This makes the program run **much faster** than interpreting bytecode step-by-step.

java characteristics :

- Simple
- Object-oriented
- Robust
- Portable
- Secure
- Multithreaded
- Architecture-Neutral:
- Interpreted & High performance
- Distributed
- Dynamic

1. Simple:

- Java is considered simple and easy to learn, especially if you already have experience with languages like C or C++.
- Java uses a clear and readable syntax, and it removes many complex features of C/C++, such as pointers and manual memory management.
- No need to worry about memory deallocation – Java handles it using automatic **garbage collection**.

- **2. Object Oriented:**
- Java is a fully object-oriented language, which means everything in Java is based on objects and classes.
- Object-oriented programming (OOP) helps you build modular, reusable, and organized code by focusing on real-world entities like objects, rather than just logic and functions.
- Key OOP Concepts in Java:
- Class – Blueprint for creating objects
- Object – Instance of a class
- Inheritance, Polymorphism, Abstraction, Encapsulation – Core principles of OOP

```
class Student
{
String name = "Akhilesh";
 int age = 35;

 void displayInfo()
    {
       System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }

}
```

```
public class StudentInfo
{
   public static void main(String[] args)
 {
         Student S1 = new Student();
         S1.displayInfo();


  }
}
```

- **3. Encapsulation:**
- Encapsulation means wrapping data and code together into a single unit (class).
- It protects data by restricting direct access using private variables and public methods.

```
class Student {
  private int age = 18;  // data is hidden and set inside the class

  public void showAge() {
    System.out.println("Age is: " + age);
  }
}

public class TestStudent {
  public static void main(String[] args) {
    Student s = new Student();
    s.showAge();  // show the age using a method
  }
}
```

- The variable age is marked as private, so it cannot be accessed directly from outside the class.
- The value 18 is already set inside the class.
- We use a public method showAge() to display the value.
- This is called Encapsulation – the data is hidden inside the class and accessed only through a method.
- It keeps the data safe and controlled.

4. Abstraction:

- Abstraction means hiding internal details and showing only essential features.
- It is implemented using abstract classes and interfaces in Java.
- In Java, you can do this using:
- **Abstract classes** (classes that can have both normal and abstract methods)
- **Interfaces** (only method definitions, no code inside)

```
abstract class Animal
 {
   // abstract method (no body) — must be implemented by subclasses
   abstract void sound();

   // normal method with body
   void sleep()
   {
      System.out.println("Animal is sleeping");
   }
}
```

```
class Dog extends Animal
{
    // provide implementation for abstract method
    void sound()
    {
        System.out.println("Dog barks");
    }
}
```

```
public class TestAbstraction
{
   public static void main(String[] args)
 {
      Dog d = new Dog();
      d.sound();  // calls Dog’s version of sound()
      d.sleep();  // calls Animal’s sleep() method
   }
}
```

- Animal is abstract: you cannot create an Animal object directly.
- sound() is abstract: it has no body here, but every subclass (like Dog) must write its own version.
- sleep() is a normal method that any animal can use.
- Dog implements the sound() method.
- When we create a Dog object and call sound(), we get “Dog barks” — but we don’t need to know how it works inside Animal class.

<u>It hides complex details (you don’t see how sound() works in all animals, just that it exists).It forces all subclasses to implement certain important methods.It helps organize code in a clean, modular way.</u>

5.Inheritance:

- Inheritance allows one class to acquire the properties and behaviors of another class.
- It promotes code **reusability** using the extends keyword.

```
class Vehicle
{
   void start()
   {
      System.out.println("Vehicle started");
   }
}
class Car extends Vehicle
 {
   void drive()
 {
      System.out.println("Car is driving");
   }
}
```

```
public class TestInheritance
 {
   public static void main(String[] args)
{
       Car c = new Car();
       c.start();   // from parent class
       c.drive();   // from child class
   }
}
```

Here, Car gets the start() method from Vehicle. No need to rewrite the same code — this is code reusability using inheritance.

6.Polymorphism:

- Polymorphism allows the same method or object to behave differently in different situations.
- It is achieved through method overloading and method overriding

OR

- Polymorphism means **one thing behaving in many ways**.
  It happens in two forms:
- **Method Overloading** (same method name, different parameters)
- **Method Overriding** (child class changes behavior of parent method)

Method Overloading Example:

```
class MathOps
{
    void add(int a, int b)
    {
        System.out.println(a + b);
    }

    void add(int a, int b, int c)
    {
        System.out.println(a + b + c);
    }
}
```

```
public class TestOverloading
{
   public static void main(String[] args)
  {
      MathOps m = new MathOps();
      m.add(2, 3);        // calls 2-arg version
      m.add(1, 2, 3);     // calls 3-arg version
   }
}
```

Method Overriding Example:

```
class Animal
{
    void sound()
   {
        System.out.println("Animal makes sound");
    }
}

class Dog extends Animal
{
    void sound()
   {
        System.out.println("Dog barks");
    }
}
```

```
public class TestPolymorphism
{
   public static void main(String[] args)
  {
      Animal a1 = new Animal();  // base class object
      Animal a2 = new Dog();     // base class reference, derived class object

      a1.sound();  // Output: Animal makes sound
      a2.sound();  // Output: Dog barks
   }
}
```

Same method name behaves differently based on situation — that’s **polymorphism**.

7. Robust:

- Java is a strictly typed language, which means you have to clearly define the type of data (like int, String, etc.) when writing code.
- It also **checks your code in two steps**:
  - **At compile time** (before running) – to catch errors early
  - **At run time** (while running) – to catch any unexpected problems
- Memory management can be a difficult, tedious task in traditional programming environments.
- **For example,** in C/C++, the programmer will often manually allocate and free all dynamic memory. This sometimes leads to problems, if programmer forgets to free memory that has been previously allocated.
- Deallocation is completely automatic, because Java provides **garbage collection** for unused objects, so you don't need to free memory yourself.
- It also has **object-oriented exception handling**, which helps you manage errors in a clean and organized way.

8. Portability:

- Portability refers to the ability to run a program on different machines.
- The Java program is getting compiled into bytecode which is platform independent.
- For **example**, the same applet must can be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet.
- There is no need to keep different versions of the applet for different computers. The same code must run on all computers.

9. Security:

- Downloading from the Internet can be risky because files might contain viruses, Trojan horses, or other harmful programs.
- Java applets run inside a secure environment called the Java execution environment.
- They are restricted and cannot access your computer’s files or disk partitions, keeping your system safe.
- Java does not allow pointers (direct memory access), which helps prevent many common programming errors and security problems.

10. Multithreaded:

- Multithreading means dividing a program into executable sub programs(threads) which can run concurrently (parallelly)
- **Examples:**
- **On a mobile home screen** : (wall paper, battery level, signal strength, location, time, etc.)
- **In a browser window: (**different tabs)
- **In a word document: (**writing process, printing process, spell check process, etc.)
- **Applications: (**Gaming, Animation, Networking programs)
- Java provides powerful and easy-to-use tools for managing multiple threads and making sure they don't interfere with each other (called synchronization). This helps programmers build smooth and interactive applications.

11. Architecture-Neutral:

- Java programs **don’t depend on the operating system**, processor, or hardware.
- This means if you upgrade your OS or change your computer’s processor, **your Java programs will still run without changes.**
- The idea behind Java and the Java Virtual Machine (JVM) is:
- **“Write once; run anywhere, any time, forever.”**

12. Distributed:

- Java is built to work well on the **Internet and networks.**
- It supports **TCP/IP protocols**, which are the basic communication rules of the internet.
- Accessing something over the Internet using a URL (like a web address) is as easy as reading a file on your computer.
- Java also supports **Remote Method Invocation (RMI),** which means a program can call methods and get results from a program running on another computer over the network.

13. Dynamic:

- Java can load classes when they are needed, not all at once.
- This is called dynamic loading.
- This makes Java programs flexible and efficient because they load only what’s necessary during execution.
- Java programs also carry extra information about their types at run-time to check and manage objects safely.
- Plus, Java can use functions written in other languages like C and C++ when needed, through something called native methods.

Writing "Hello World" Java Program

HelloWorld.java

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

Note: Save the file with the name of class inside which main( ) resides

Java Program Structure

```
//  comments about the class
class HelloWorld
{




}
```

class header

class body

Comments can be placed almost anywhere

Java Program Structure

```java
class HelloWorld
{
   //  comments about the method
   public static void main (String args[])
   {

   }

}
```

method header

method body

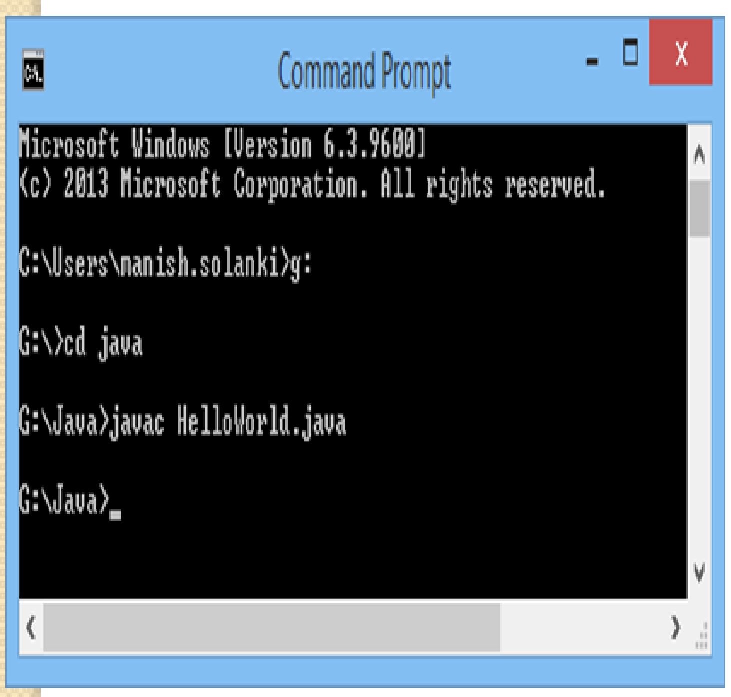
Comments

- Comments in a program are called *inline documentation*
- They should be included to explain the purpose of the program and describe processing steps
- They do not affect how a program works
- Java comments can take three forms:

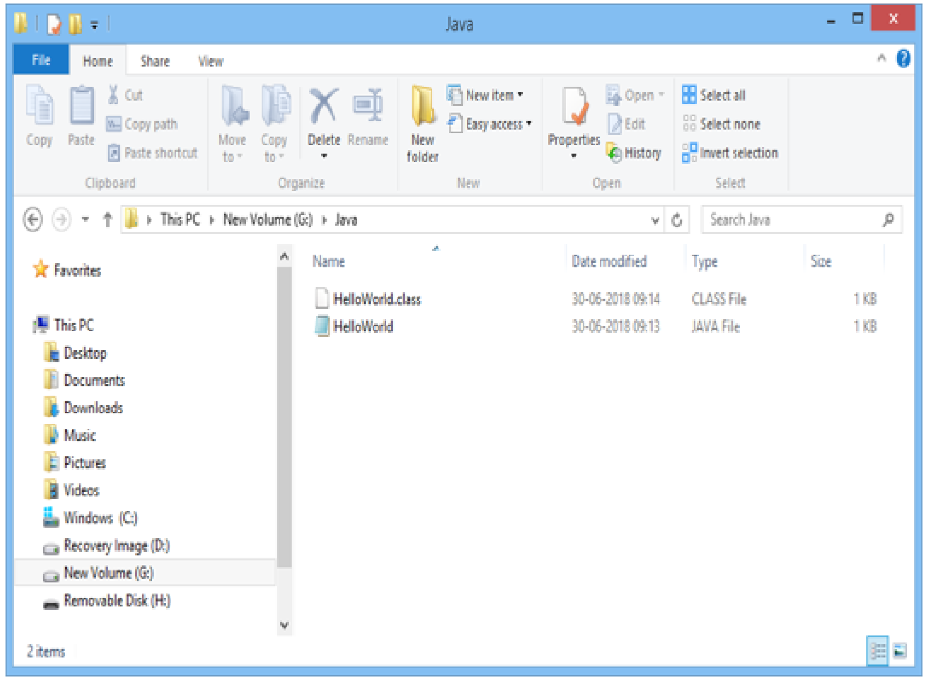
```
// this comment runs to the end of the line

/*  this comment runs to the terminating
    symbol, even across line breaks        */

/** this is a javadoc comment   */
```

Compiling "Hello World" Java Program

- Open Command Prompt
- Make a directory (inside which java programs are residing) as working/present directory
- Compile the program with javac program.java



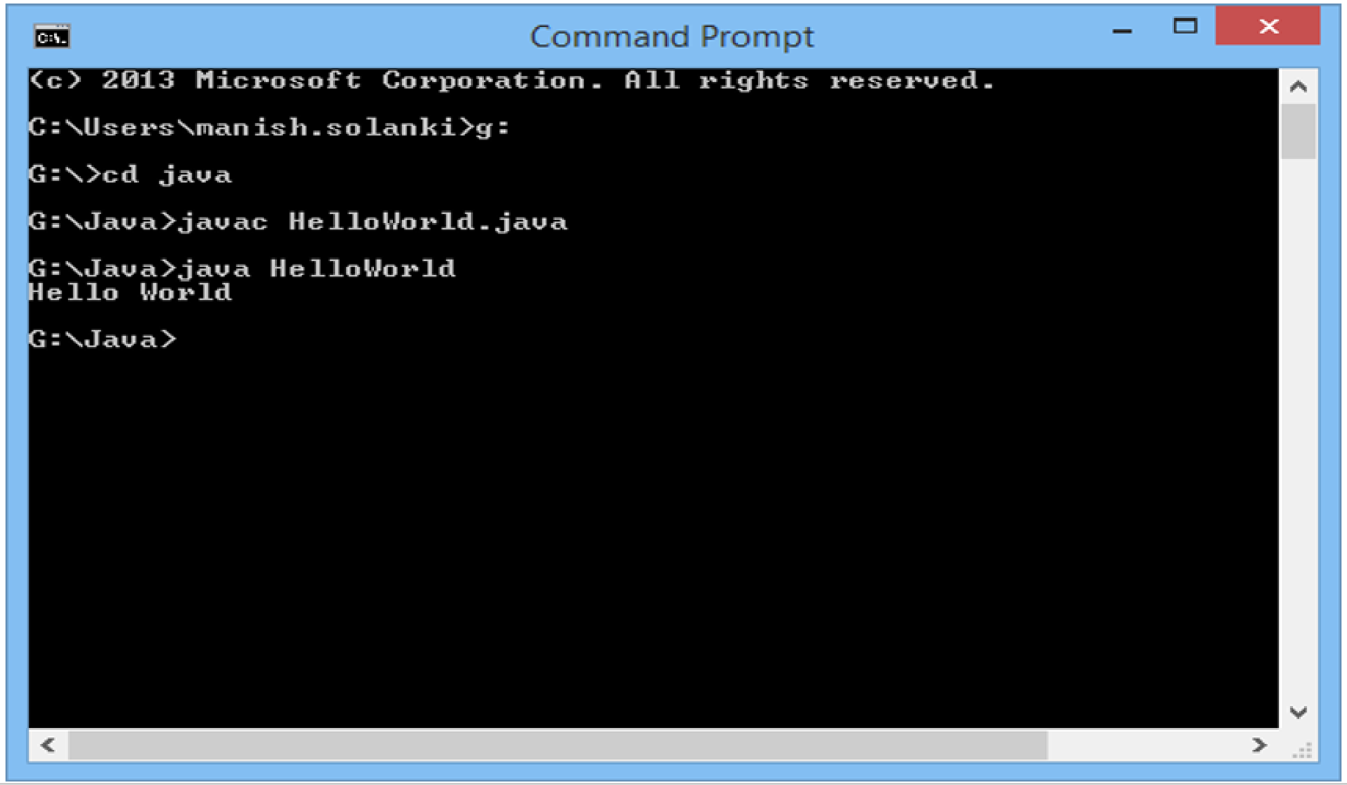
```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\manish.solanki>g:

G:\>cd java

G:\Java>javac HelloWorld.java

G:\Java>_
```



Name	Date modified	Type	Size
HelloWorld.class	30-06-2018 09:14	CLASS File	1 KB
HelloWorld	30-06-2018 09:13	JAVA File	1 KB

Executing "Hello World" Java Program

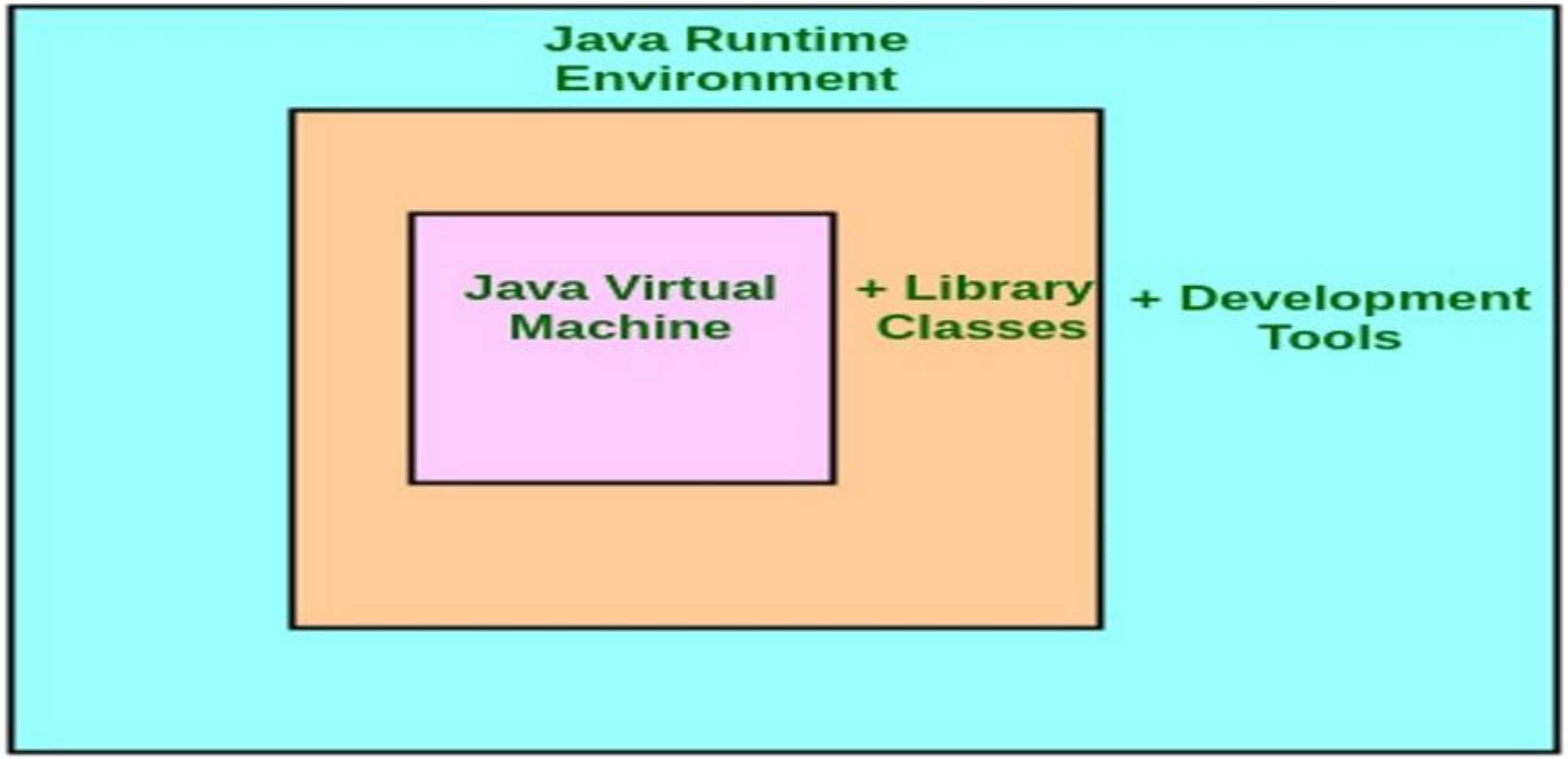
- After successfully compilation of program .class file (bytecode) is generated
- To execute, we have to write : java filename command



Command Prompt

```
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\manish.solanki>g:

G:\>cd java

G:\Java>javac HelloWorld.java

G:\Java>java HelloWorld
Hello World

G:\Java>
```

Java Ecosystem



JDK = JRE + Development Tool
JRE = JVM + Library Classes

JVM: JAVA VIRTUAL MACHINE

- JVM (Java Virtual Machine) is an abstract machine. It is called virtual machine because it doesn't physically exist.
- It is a specification that provides runtime environment in which java bytecode can be executed
- JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS are different from each other
- Functions of JVM:

  -Loads code

  -Verifies code

  -Executes code

  -Provides runtime environment

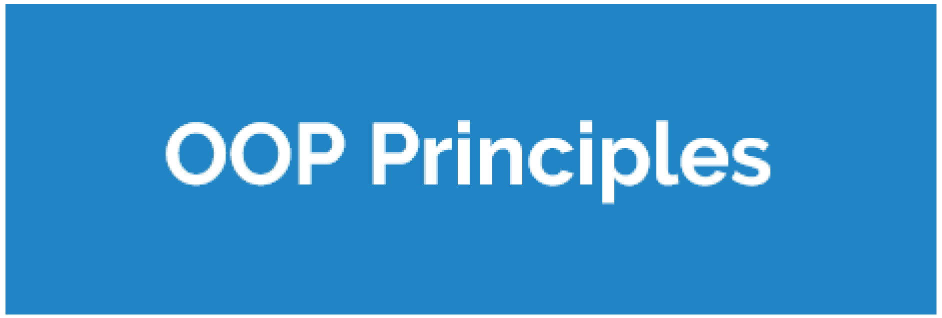
JRE: Java Runtime Environment

- JRE is an acronym for Java Runtime Environment
- It is also written as Java RTE.
- The Java Runtime Environment is a set of software tools which are used for developing java applications
- It is used to provide runtime environment. It is the implementation of JVM
- It physically exists.
- It contains set of libraries + other files that JVM uses at runtime

JDK: JAVA DEVELOPMENT KIT

- The Java Development Kit (JDK) is a software development environment which is used to develop java applications and applets.
- It physically exists.
- It contains JRE + development tools(i.e. javac,java,jar,etc.)
- JDK is an implementation of any one of the below given Java Platforms released by Oracle corporation:
- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform
- Applets(An **applet** in Java is a small program that is typically embedded in a web page and runs in a web browser using the Java Virtual Machine (JVM). Applets were commonly used in the past for adding interactive features to web applications, such as games, calculators, or visualizations.)

1.2 Abstraction, OOP Principles. -Encapsulation, Inheritance and Polymorphism



OOP Principles

Class:

- We can create a user defined data type in Java using class.
- A class can be considered as a “Template” or “Blue Print” or “Dye” which declares data and code(function) inside it.
- A class is a logical reality from which we can create n no. of instances i.e. from a dye, we can create so many idols, from a blue print (plan/design/pattern), we can construct so many buildings physically, etc. Thus a class is a collection of objects of similar types.

Object:

- Basic run time entities in OOP i.e. a person, a bank, an employee, a student, etc.
- Are variables of type class
- Is a physical reality
- Each object contains data and code(function) to manipulate data
- Objects interact with each other by sending messages. i.e. a customer object can request an account object for the bank balance.

Data Encapsulation:

- The wrapping up of data and functions into a single unit (class) is known as Encapsulation
- Only functions declared inside the class can access the data. (outside interference is restricted – security)
- Also called as “Data Hiding” or “Information Hiding”
- The functions provide an interface between an object’s data and the program.
- Encapsulation is achieved trough access modifiers private and protected in Java.

Data Encapsulation

```
Example: class Employee
{ // data members
  private  int code;
  private  String  name;
  private  float salary;
  private  int experience;
// member methods
void setEmp()
 {
    // data input
 }
 void getEmp()
 {
  // data output
 }
} // end of class
```

The attributes i.e. code, name, salary, experience are called as "data members"

The functions that operate on this data are called as "member functions " or " methods"

```
class EmployeeMain
{
   public static void main(String args[])
  {
  Employee e=new Employee();
  e.code=1; // error becuase of private access
specifier
   }
}
```

1.3 Constant, Variables and Data Types, Type casting :

- A constant is a variable whose value cannot be changed once it is assigned. In Java, we use the final keyword to declare a constant.
- **Syntax:** final dataType CONSTANT_NAME = value;
- **Example:** final double PI = 3.14159;

Variables:

- A variable is like a container that holds a value. This value can change while the program runs.
- **Syntax:** datatype variableName = value;
- **Example:** int age = 25;   or String name = "John";

Data Types:

🔷 A. Primitive Data Types (8 types):

Data Type	Size	Example	Description
`int`	4 bytes	100	Whole numbers
`double`	8 bytes	3.14	Decimal numbers (more precise)
`float`	4 bytes	2.5f	Decimal numbers (less precise)
`char`	2 bytes	'A'	Single character
`boolean`	1 bit	true/false	Logical value (yes/no)
`byte`	1 byte	127	Small integers
`short`	2 bytes	32000	Shorter range integers
`long`	8 bytes	1000000000L	Very large integers

🔷 B. Non-Primitive Data Types:

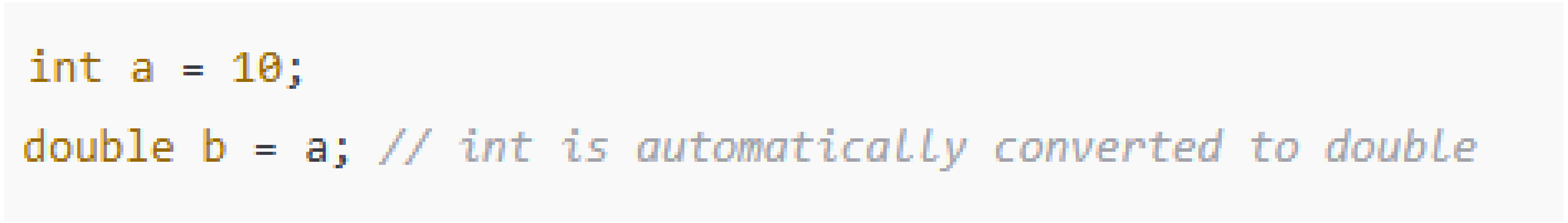
Includes **String**, **Arrays**, **Objects**, etc.

```java
String city = "London";
int[] marks = {90, 80, 70}; // Array of integers
```

Type casting : Type Casting means converting one data type into another.

A. Implicit Casting (Widening)

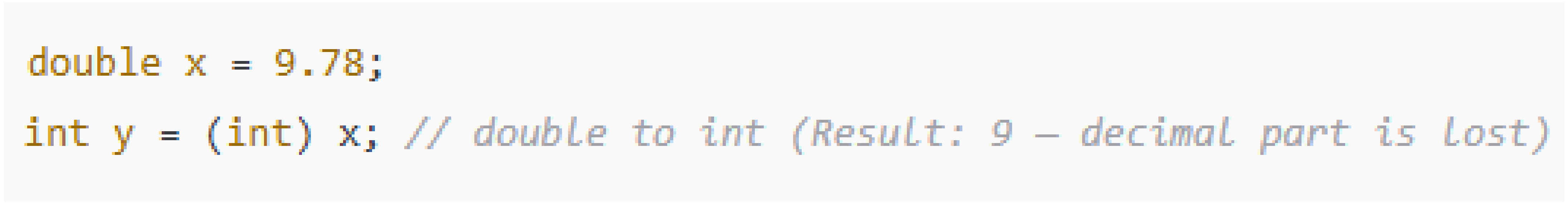
- Java **automatically converts** smaller data types into larger ones (no data loss).



```
int a = 10;
double b = a; // int is automatically converted to double
```

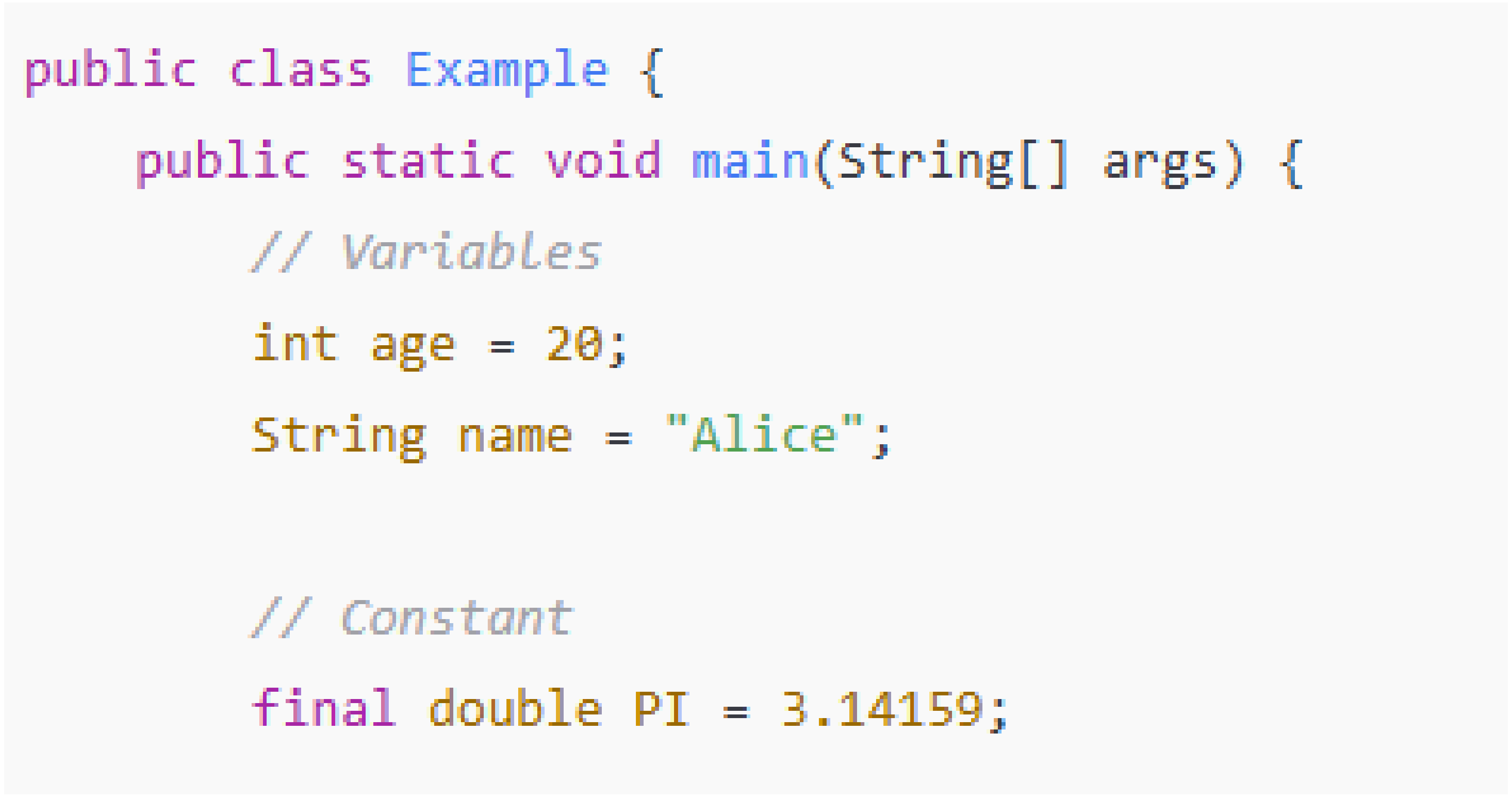
B. Explicit Casting (Narrowing):

- You have to **manually convert** larger data types into smaller ones (possible data loss).



```
double x = 9.78;
int y = (int) x; // double to int (Result: 9 – decimal part is lost)
```

EXAMPLE:



```java
public class Example {
    public static void main(String[] args) {
        // Variables
        int age = 20;
        String name = "Alice";

        // Constant
        final double PI = 3.14159;
```

```java
// Data Types
boolean isPassed = true;
char grade = 'A';
float percentage = 85.6f;

// Type Casting
int a = 50;
double b = a; // Implicit casting
double price = 99.99;
int newPrice = (int) price; // Explicit casting
```

```java
        // Output
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Grade: " + grade);
        System.out.println("Passed: " + isPassed);
        System.out.println("PI: " + PI);
        System.out.println("Implicit Casting (int to double): " + b);
        System.out.println("Explicit Casting (double to int): " + newPrice);
    }
}
```

1.4 Operator and Expression, Arithmetic Operators, Relational Operators, Logical Operators, Assignment Operator, Increment and Decrement Operator, Bit wise Operator, Special Operator .

Bit wise Operator:

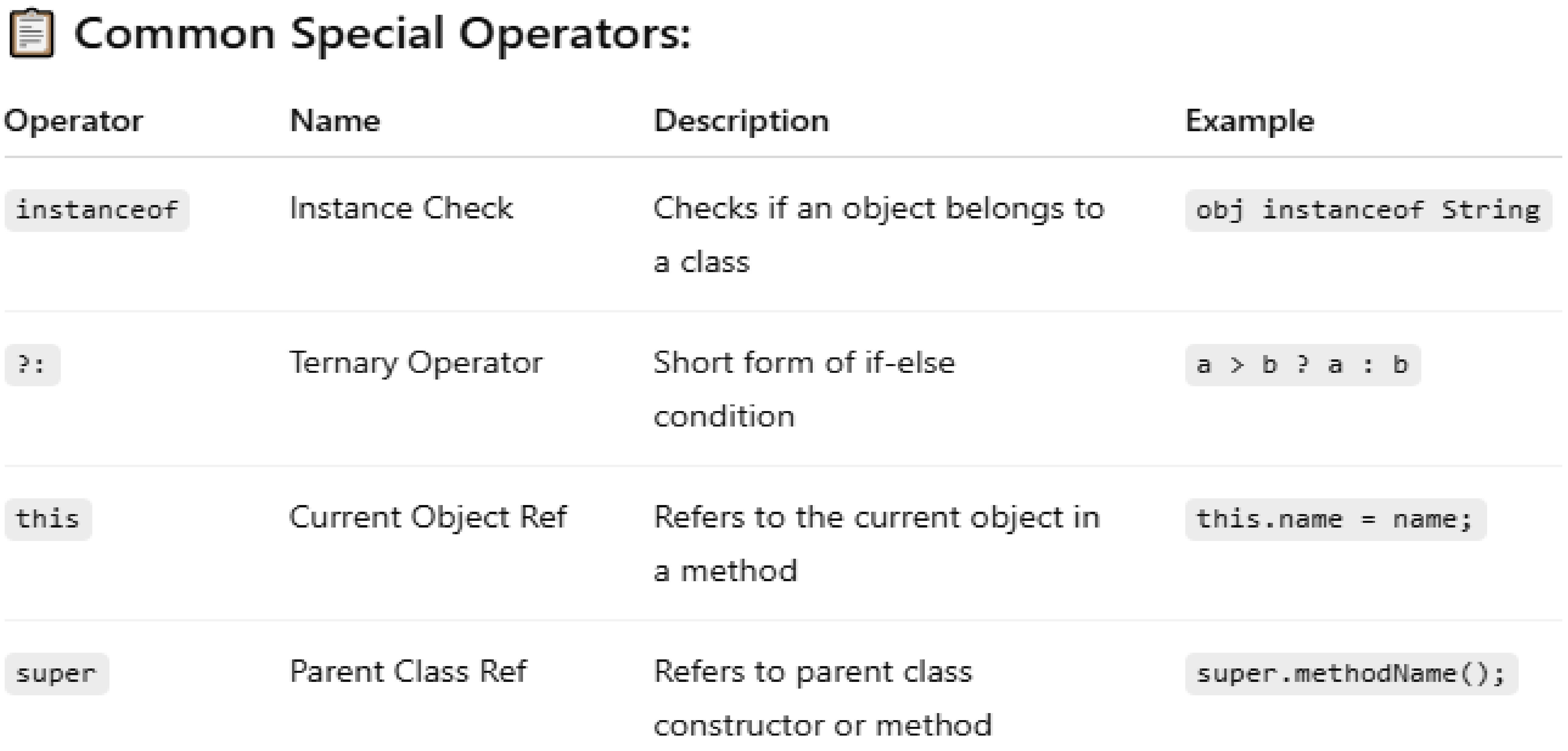
Bitwise operators work on **individual bits (0 or 1)** of integers. These are mainly used for low-level programming, performance, or bit manipulation.

📋 List of Bitwise Operators:

Operator	Name	Description	Example (`a=5, b=3`)
`&`	AND	1 if **both** bits are 1	`a & b = 1`
`	`	OR	1 if **any one** of the bits is 1
`^`	XOR	1 if bits are **different**	`a ^ b = 6`
`~`	NOT	Inverts each bit (1→0, 0→1)	`~a = -6`
`<<`	Left Shift	Shifts bits to the left, adds 0s on the right	`a << 1 = 10`
`>>`	Right Shift	Shifts bits to the right, keeps the sign bit	`a >> 1 = 2`
`>>>`	Unsigned Right Shift	Same as right shift, but fills 0s from the left	`a >>> 1 = 2`

```java
public class BitwiseExample {
    public static void main(String[] args) {
        int a = 5;  // Binary: 0101
        int b = 3;  // Binary: 0011

        System.out.println("a & b: " + (a & b));   // 1
        System.out.println("a | b: " + (a | b));   // 7
        System.out.println("a ^ b: " + (a ^ b));   // 6
        System.out.println("~a: " + (~a));         // -6
        System.out.println("a << 1: " + (a << 1)); // 10
        System.out.println("a >> 1: " + (a >> 1)); // 2
    }
}
```

Special Operator: Java has some special-purpose operators that don’t fit into other categories like arithmetic or logical.



📋 Common Special Operators:

Operator	Name	Description	Example
`instanceof`	Instance Check	Checks if an object belongs to a class	`obj instanceof String`
`?:`	Ternary Operator	Short form of if-else condition	`a > b ? a : b`
`this`	Current Object Ref	Refers to the current object in a method	`this.name = name;`
`super`	Parent Class Ref	Refers to parent class constructor or method	`super.methodName();`

1. `instanceof`

```java
String s = "Hello";
System.out.println(s instanceof String);  // true
```

2. `?:` (Ternary)

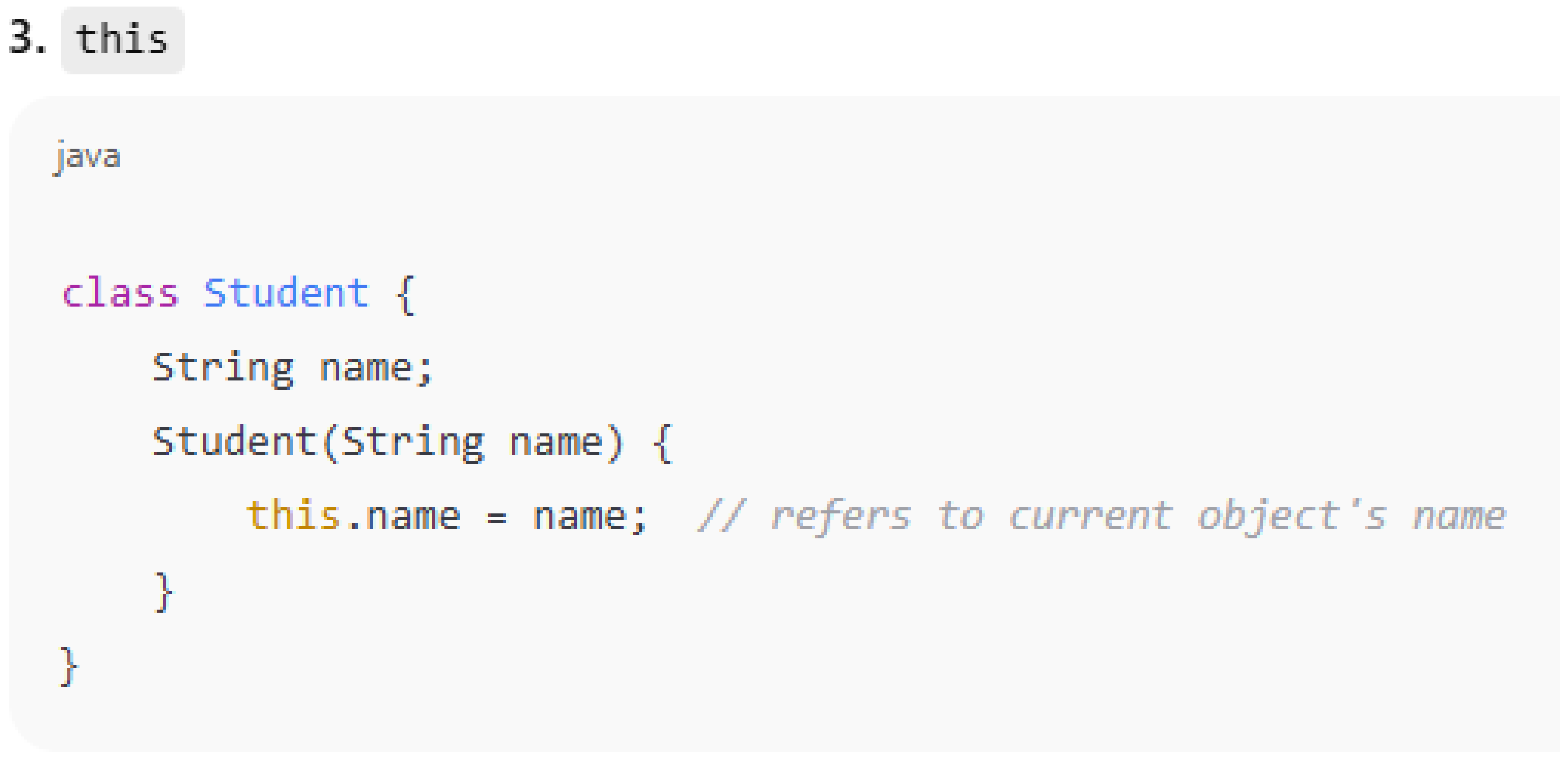
```java
int a = 10, b = 20;
int max = (a > b) ? a : b;  // returns 20
System.out.println("Max: " + max);
```

Syntax:

```
condition ? value_if_true : value_if_false;
```

```
int num = 7;
String result = (num % 2 == 0) ? "Even" : "Odd";
System.out.println("Number is: " + result);
```

Part	Description
`condition`	Expression to check (like `a > b`)
`?`	Separator between condition and true value
`value_if_true`	Returned if condition is true
`:`	Separator between true and false values
`value_if_false`	Returned if condition is false



3. this

java

```java
class Student {
    String name;
    Student(String name) {
        this.name = name;  // refers to current object's name
    }
}
```

The **this** keyword refers to the **current object** — the object whose method or constructor is being called.

We mostly use this to **avoid confusion** when local (method/constructor) variables have **the same name** as instance variables (class variables).

```java
class Student {
    String name;

    // Constructor
    Student(String name) {
        this.name = name;  // 'this.name' = class variable, 'name' = constructor
    }

    void display() {
        System.out.println("Name: " + this.name); // optional use of this
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Ravi");
        s1.display();  // Output: Name: Ravi
    }
}
```

4. `super`

```java
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    void sound() {
        super.sound();  // calls parent class method
        System.out.println("Dog barks");
    }
}
```

1. Operator and Expression in Java

- **Operator:** Symbols that perform operations on variables and values.
- **Expression:** Combination of variables, operators, and method calls that evaluates to a value.

```
int a = 5;
int b = 3;
int c = a + b * 2;  // Expression: a + b * 2 evaluates to 5 + 3*2 = 11
```

2. Arithmetic Operators in Java

Operator	Description	Example	Result
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`5 - 3`	`2`
`*`	Multiplication	`5 * 3`	`15`
`/`	Division	`10 / 3`	`3` (integer division)
`%`	Modulus (remainder)	`10 % 3`	`1`

```java
int x = 10, y = 3;

int sum = x + y;        // 13
int diff = x - y;       // 7
int product = x * y;    // 30
int quotient = x / y;   // 3 (integer division)
int remainder = x % y;  // 1
```

3. Relational Operators in Java

3. Relational Operators

Used to compare two values, result is a boolean (`true` or `false`).

Operator	Meaning	Example	Result
`==`	Equal to	`a == b`	`true` if equal
`!=`	Not equal to	`a != b`	`true` if not equal
`>`	Greater than	`a > b`	`true` if a is greater
`<`	Less than	`a < b`	`true` if a is less
`>=`	Greater or equal	`a >= b`	`true` if a >= b
`<=`	Less or equal	`a <= b`	`true` if a <= b

```java
int a = 5, b = 8;

boolean result1 = (a == b);  // false
boolean result2 = (a != b);  // true
boolean result3 = (a > b);   // false
boolean result4 = (a < b);   // true
boolean result5 = (a >= b);  // false
boolean result6 = (a <= b);  // true
```

4. Logical Operators in Java

Logical operators are used to combine or invert **boolean** expressions (expressions that result in `true` or `false`). These operators help you control the flow of your program by testing multiple conditions together.

- `p && q` → `true && false` → `false`
- `p || q` → `true || false` → `true`
- `!p` → `!true` → `false`

- In Java, logical **AND** is `&&`
- Logical **OR** is `||`
- Logical **NOT** is `!`

```java
boolean p = true, q = false;

boolean andResult = p && q;  // false
boolean orResult = p || q;   // true
boolean notResult = !p;      // false
```

5. Assignment Operator in Java

Assigns value to a variable.

- Simple assignment: `=`
- Compound assignment: `+=`, `-=`, `*=`, `/=`, `%=`
  Example: `a += 5;` means `a = a + 5;`

```
int num = 10;  // Assignment


num += 5;   // Equivalent to num = num + 5; num is now 15
num -= 3;   // num = num - 3; num is now 12
num *= 2;   // num = num * 2; num is now 24
num /= 4;   // num = num / 4; num is now 6
num %= 5;   // num = num % 5; num is now 1
```

6. Increment and Decrement Operators in Java

Increase or decrease a variable's value by 1.

- `++` increment operator
- `--` decrement operator

Two forms:

- **Prefix:** `++a` (increment, then use)
- **Postfix:** `a++` (use, then increment)

```
int i = 5;

int preIncrement = ++i;  // i becomes 6, preIncrement = 6
int postIncrement = i++; // postIncrement = 6, i becomes 7

int preDecrement = --i;  // i becomes 6, preDecrement = 6
int postDecrement = i--; // postDecrement = 6, i becomes 5
```

Java Program Demonstrating All Operators

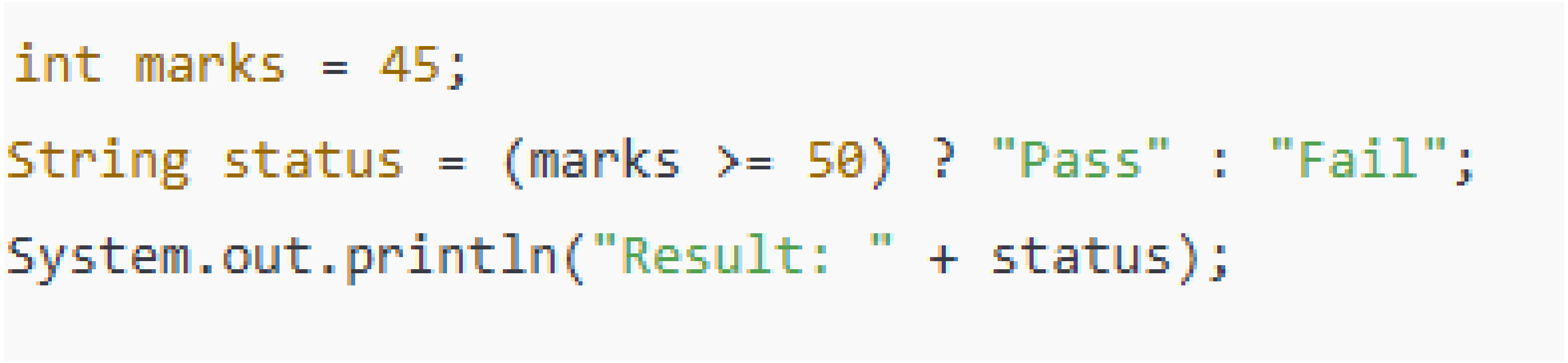
```java
public class OperatorDemo {
    public static void main(String[] args) {
        // Arithmetic Operators
        int a = 10, b = 3;
        System.out.println("Arithmetic Operations:");
        System.out.println("a + b = " + (a + b));  // 13
        System.out.println("a - b = " + (a - b));  // 7
        System.out.println("a * b = " + (a * b));  // 30
        System.out.println("a / b = " + (a / b));  // 3 (integer division)
        System.out.println("a % b = " + (a % b));  // 1

        // Relational Operators
        System.out.println("\nRelational Operations:");
        System.out.println("a == b: " + (a == b)); // false
        System.out.println("a != b: " + (a != b)); // true
        System.out.println("a > b: " + (a > b));   // true
        System.out.println("a < b: " + (a < b));   // false
        System.out.println("a >= b: " + (a >= b)); // true
        System.out.println("a <= b: " + (a <= b)); // false
```

```java
// Logical Operators
boolean p = true, q = false;
System.out.println("\nLogical Operations:");
System.out.println("p && q: " + (p && q)); // false
System.out.println("p || q: " + (p || q)); // true
System.out.println("!p: " + (!p));          // false

// Assignment Operators
System.out.println("\nAssignment Operations:");
int c = 5;
System.out.println("Initial c: " + c);
c += 3;  // c = 8
System.out.println("c += 3: " + c);
c -= 2;  // c = 6
System.out.println("c -= 2: " + c);
c *= 4;  // c = 24
System.out.println("c *= 4: " + c);
c /= 6;  // c = 4
System.out.println("c /= 6: " + c);
c %= 3;  // c = 1
System.out.println("c %= 3: " + c);
```

```
        // Increment and Decrement Operators
        System.out.println("\nIncrement and Decrement:");
        int i = 10;
        System.out.println("Initial i: " + i);
        System.out.println("++i (prefix increment): " + (++i)); // 11
        System.out.println("i++ (postfix increment): " + (i++)); // 11
        System.out.println("After i++: " + i);                   // 12
        System.out.println("--i (prefix decrement): " + (--i)); // 11
        System.out.println("i-- (postfix decrement): " + (i--)); // 11
        System.out.println("After i--: " + i);                   // 10
    }
}
```

1.5 Decision making with simple if, if… else, else if ladder statements, The switch statement, The conditional operator :

The conditional operator :



```
int marks = 45;
String status = (marks >= 50) ? "Pass" : "Fail";
System.out.println("Result: " + status);
```

1. Simple if Statement

- Executes a block of code **only if** a specified condition is `true`.
- If the condition is `false`, the block is skipped.

Syntax:

```java
if (condition) {
    // code to execute if condition is true
}
```

Example:

```java
if (a > 10) {
    System.out.println("a is greater than 10");
}
```

2. if...else Statement

- Executes one block of code if the condition is `true`, otherwise executes another block if the condition is `false`.

Syntax:

```java
if (condition) {
    // executes if condition is true
} else {
    // executes if condition is false
}
```

Example:

```java
if (a > 10) {
    System.out.println("a is greater than 10");
} else {
    System.out.println("a is 10 or less");
}
```

3. else if Ladder

- Used to check multiple conditions sequentially.
- Executes the block of the **first true** condition and skips the rest.

Syntax:

```java
if (condition1) {
    // executes if condition1 is true
} else if (condition2) {
    // executes if condition2 is true
} else if (condition3) {
    // executes if condition3 is true
} else {
    // executes if none of the above conditions are true
}
```

Example:

```java
if (a > 10) {
    System.out.println("a is greater than 10");
} else if (a == 10) {
    System.out.println("a is equal to 10");
} else {
    System.out.println("a is less than 10");
}
```

4. switch Statement

- Used to select one of many code blocks to be executed.
- The variable/expression is compared with different `case` values.
- When a matching case is found, the corresponding block is executed.
- `break` is used to exit the switch block after a case matches.
- `default` block executes if none of the cases match.

Syntax:

```java
switch (expression) {
    case value1:
        // code to execute if expression == value1
        break;
    case value2:
        // code to execute if expression == value2
        break;
    // more cases...
    default:
        // code to execute if none of the above cases match
}
```

Example:

```
int day = 3;

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

Complete Java Program Demonstrating Decision Making

```java
public class DecisionMakingDemo {
    public static void main(String[] args) {
        int a = 15;

        // Simple if
        if (a > 10) {
            System.out.println("Simple if: a is greater than 10");
        }

        // if...else
        if (a % 2 == 0) {
            System.out.println("if...else: a is even");
        } else {
            System.out.println("if...else: a is odd");
        }
```

```java
// else if ladder
if (a > 20) {
    System.out.println("else if ladder: a is greater than 20");
} else if (a == 15) {
    System.out.println("else if ladder: a is exactly 15");
} else {
    System.out.println("else if ladder: a is less than 15");
}
```

```java
        // switch statement
        int day = 3;
        System.out.print("switch: Day " + day + " is ");
        switch (day) {
            case 1:
                System.out.println("Monday");
                break;
            case 2:
                System.out.println("Tuesday");
                break;
            case 3:
                System.out.println("Wednesday");
                break;
            case 4:
                System.out.println("Thursday");
                break;
            default:
                System.out.println("Invalid day");
        }
    }
}
```

1.6 Decision Making with Loops i.e. while, do and for statement, Jumps in Loops, Labelled Loops :

Loops are used to execute a block of code repeatedly based on a condition.

1. while loop

- **Description:** Repeats a block of code as long as the condition is true.
- **Syntax:**

```
while (condition) {
    // statements
}
```

- The condition is checked **before** the loop body is executed (pre-test loop).

- **Example:**

java

```java
int i = 1;
while (i <= 5) {
    System.out.println("i = " + i);
    i++;
}
```

This will print `i` from 1 to 5.

2. do-while loop

- **Description:** Executes the block of code once first, then repeats the loop as long as the condition is true.

- **Syntax:**

```java
do {
    // statements
} while (condition);
```

- The condition is checked **after** the loop body (post-test loop).

Example:

```
int i = 1;
do {
    System.out.println("i = " + i);
    i++;
} while (i <= 5);
```

This will also print `i` from 1 to 5, but guarantees the loop runs at least once.

3. for loop

- **Description:** Best used when the number of iterations is known. It has initialization, condition, and increment/decrement all in one line.

- **Syntax:**

```java
for (initialization; condition; update) {
    // statements
}
```

Example:

```java
for (int i = 1; i <= 5; i++) {
    System.out.println("i = " + i);
}
```

Output:

```
i = 1
i = 2
i = 3
i = 4
i = 5
```

🔶 1. **Jumps in Loops**

In Java, jump statements are used to control the **flow of loops**. The main jump statements are:

✅ A. `break`

> Used to **exit** the loop immediately.

```java
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        break;
    }
    System.out.println(i);
}
```

💻 **Output:**

```
1
2
```

👉 Loop stops when `i == 3`.

✅ B. `continue`

> Skips the **current iteration** and goes to the **next** one.

java Copy Edit

```java
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;
    }
    System.out.println(i);
}
```

🖥 **Output:**

Copy Edit

```
1
2
4
5
```

👉 `i == 3` is skipped.

✅ C. `return`

> Exits the **method completely**, even if inside a loop.

```java
public class Main {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                return;
            }
            System.out.println(i);
        }
        // This line won't run
        System.out.println("End of loop");
    }
}
```

Output:

```
1
2
```

👉 The method exits completely when `i == 3`.

🔶 2. **Labelled Loops**

✅ What are Labelled Loops?

Java allows you to give a **name (label)** to a loop. This is useful when you have **nested loops**, and you want to break or continue **outer** loops, not just the inner one.

🔷 Syntax:

```java
labelName:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (someCondition) {
            break labelName; // breaks the outer loop
        }
    }
}
```

✅ Example: `break` with label

```java
outer:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (i == 2 && j == 2) {
            break outer; // exits the outer loop
        }
        System.out.println(i + " " + j);
    }
}
```

🖥 Output:

```
1 1
1 2
1 3
2 1
```

👉 When `i == 2` and `j == 2`, it breaks out of the outer loop.

✅ Example: `continue` with label

java

```java
outer:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) {
            continue outer; // skips to next iteration of outer loop
        }
        System.out.println(i + " " + j);
    }
}
```

💻 Output:

```
1 1
2 1
3 1
```

👉 When `j == 2`, it skips the remaining part of the **outer** loop and continues with the next `i`.

Java Keywords

abstract	else	interface	switch
assert	enum	long	synchronized
boolean	extends	native	this
break	false	new	throw
byte	final	null	throws
case	finally	package	transient
catch	float	private	true
char	for	protected	try
class	goto	public	void
const	if	return	volatile
continue	implements	short	while
default	import	static	
do	instanceof	strictfp	
double	int	super	