**Java is Strongly typed language**

**Reason:**

1. java demands variable declaration with datatype. as we can't

create a variable without knowing the range of values it can hold.

2.Once declared, the data type of the variable cannot be changed.

Justification:

While it's true that typecasting can be used to temporarily treat a value as a different data type, the underlying data type of a variable remains the same. The data type of a variable determines the range of values it can hold and the operations that can be performed on it.

For example, if a variable is declared as an int, it will always be an int and can only hold integer values, regardless of whether or not it is typecasted to a different data type. The typecasting only affects how the value is interpreted at runtime and doesn't change the underlying data type of the variable.

In this sense, it is accurate to say that the data type of a variable cannot be changed once it has been declared. While typecasting can be used to work with values of different data types, it does not change the fundamental nature of the data type of a variable.

**why we say that java is both compiled and interpreted language**

Java is considered both a compiled and interpreted language because it uses a two-stage compilation process.

In the first stage, the Java source code is compiled into bytecode by the Javac compiler. This bytecode is a platform-independent format that can be executed on any machine that has a Java Virtual Machine (JVM) installed.

In the second stage, the bytecode is interpreted by the JVM at runtime, which converts it into machine code that can be executed by the host machine. This interpretation process allows Java to be platform-independent and to be executed on any machine that has a compatible JVM.

So, while Java code is compiled into bytecode before execution, the bytecode is interpreted at runtime, making Java both a compiled and interpreted language.

**why jvm is platform dependant**

The Java Virtual Machine (JVM) is platform dependent because it is designed to provide a layer of abstraction between the Java bytecode and the underlying hardware and operating system. The JVM is responsible for interpreting the bytecode and executing the Java program on the host machine.

To achieve this, the JVM needs to be aware of the host machine's architecture, operating system, and hardware configuration, including the CPU, memory, and I/O devices. The JVM uses this information to optimize the execution of the Java program and to provide a consistent and reliable environment for running Java applications.

Because of the JVM's close integration with the underlying hardware and operating system, it is platform dependent, meaning that different JVM implementations are required for different platforms. Each JVM implementation is specifically designed for a particular platform, and it includes platform-specific optimizations and features that allow it to run efficiently on that platform.

However, the Java programming language itself is platform-independent, meaning that Java code can be compiled into bytecode on any platform and then executed on any other platform that has a compatible JVM installed.

**JDK,JRE,JVM**

Java Development Kit (JDK)

It is platform-dependent.

As it has different OS platform versions for Windows, Linux, Mac, etc.

Java Runtime Environment (JRE)

It is platform-dependent.

The JRE includes class libraries along with JVM and its supporting files.

justification: in java we can use native libraries to execute the extra functionalities apart from the java standard library. so they can be different for different platforms.

In general, Java libraries are designed to be platform-independent,

but it's important to keep in mind that some libraries and APIs may be specific to a particular operating system.

JVM(java virtual machine) =>

it is platform dependant.

it is designed to execute Java bytecode, which is a machine-independent representation of Java code.

the implementation of the JVM is specific to a particular operating system and hardware architecture,

and must be installed and configured for each platform that a Java application will run on.

Therefore I can say that JDK,JRE,JVM are all platform dependent as their performance may vary as per the platform

rather I can say that the byte code file is platform independent which can be executed on any platform.

**Can string,array be considered as primitive datatye in java**

No, strings and arrays are not considered primitive data types in Java.

Strings are actually objects in Java and are instances of the java.lang.String class. They are used to represent sequences of characters and are not primitive data types.

Arrays, on the other hand, are also not primitive data types. An array in Java is an object that stores a fixed-size sequential collection of elements of the same type.

In Java, there are only 8 primitive data types: boolean, char, byte, short, int, long, float, and double. These data types are the building blocks of all other data types in the language and are stored directly in the memory of the computer.

**why String is object in java**

In Java, a string is considered an object because it is an instance of the java.lang.String class, which is a built-in class in the Java API. This means that strings in Java have their own set of methods and properties that allow you to manipulate and work with the string data.

One of the reasons for making strings objects in Java is to provide a convenient and flexible way to represent and manipulate sequences of characters. With the String class, you can perform operations such as concatenating strings, extracting substrings, searching for specific characters or substrings, and so on.

Another reason is that strings are immutable in Java, meaning that once a string is created, its value cannot be changed. This can have important implications for performance and memory usage, as it allows the Java runtime to optimize the storage and processing of strings.

So, in summary, strings are objects in Java because they provide a convenient and flexible way to represent and manipulate character data, and because of the immutability of strings in the language.

**expression,statement,block in java**

In Java, an **expression** is a combination of values, variables, and operators that evaluates to a value.

For example, 2 + 3 is an expression that evaluates to 5.

A **statement** is a unit of code that performs an action.

For example, an assignment statement such as int x = 2 + 3; assigns the value of the expression 2 + 3 to the variable x.

A statement forms a complete unit of execution.

The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).

1. Assignment expressions → aValue = 8933.234;
2. Any use of ++ or – —→ aValue++; a–;
3. Method invocations → System.out.println("Hello World!");
4. Object creation expressions → Bicycle myBike = new Bicycle();
5. Declaration statement ⇒ int value=34;
6. Control flow statement ⇒ if else, else if,switch, break,continue,return,for while,do while

Such statements are called expression statements.

A **block** is a group of statements surrounded by curly braces {}. Blocks can be used to group statements together, control the scope of variables, and define the body of methods, constructors, and other control structures. For example, the following code is an example of a block:

{

int x = 2 + 3;

int y = x \* 4;

System.out.println(y);

}

A block is a group of zero or more statements between balanced braces.

**Control Flow Statements**

Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code

This section describes

1. decision-making statements (if, if-else-if, switch),
2. looping statements (for, while, do-while),
3. branching statements (break, continue, return)

* **If**

It tells your program to execute a certain section of code only if a particular test evaluates to true.

void applyBrakes() {

// the "if" clause: bicycle must be moving

if (isMoving){

// the "then" clause: decrease current speed

currentSpeed--;

}

}

**Scenarios:**

* + **User authentication**: If a user is not logged in, an if statement can be used to redirect the user to the login page.
  + **Shopping cart validation:** If the shopping cart is empty, an if statement can be used to display a message to the user to add items to the cart

**If-Else-If**

"if-else if" statement is a series of "if" conditions, each followed by an optional "else if" condition.

The conditions are checked in the order they are written, and the first condition that evaluates to true will execute its corresponding block of code.

void applyBrakes() {

if (isMoving) {

currentSpeed--;

} else {

System.err.println("The bicycle has already stopped!");

}

}

**Scenario:**

Promotional code validation: If a user enters a valid promotional code, an if-else statement can be used to apply the discount. If the code is invalid, the else branch can display an error message.

Prime membership: show products with discounted price only for prime membership holders. Show normal prices for regular users.

**Switch Statement**

The "switch" statement, provides a more concise way to handle multiple branches of execution based on the value of an expression. It works by matching the value of the expression with a case label, and executing the code associated with that label

the switch statement can have a number of possible execution paths.

public class SwitchDemo {

public static void main(String[] args) {

int month = 8;

String monthString;

switch (month) {

case 1: monthString = "January";

break;

case 2: monthString = "February";

break;

case 3: monthString = "March";

break;

case 4: monthString = "April";

break;

case 5: monthString = "May";

break;

case 6: monthString = "June";

break;

case 7: monthString = "July";

break;

case 8: monthString = "August";

break;

case 9: monthString = "September";

break;

case 10: monthString = "October";

break;

case 11: monthString = "November";

break;

case 12: monthString = "December";

break;

default: monthString = "Invalid month";

break;

}

System.out.println(monthString);

}

}

The body of a switch statement is known as a *switch block*. A statement in the switch block can be labeled with one or more case or default labels. The switch statement evaluates its expression, then executes all statements that follow the matching case label.

the final break is not required because flow falls out of the switch statement. Using a break is recommended so that modifying the code is easier and less error prone. The default section handles all values that are not explicitly handled by one of the case sections.

The following code example, [SwitchDemo2](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/SwitchDemo2.java), shows how a statement can have multiple case labels. The code example calculates the number of days in a particular month:

class SwitchDemo2 {

public static void main(String[] args) {

int month = 2;

int year = 2000;

int numDays = 0;

switch (month) {

case 1: case 3: case 5:

case 7: case 8: case 10:

case 12:

numDays = 31;

break;

case 4: case 6:

case 9: case 11:

numDays = 30;

break;

case 2:

if (((year % 4 == 0) &&

!(year % 100 == 0))

|| (year % 400 == 0))

numDays = 29;

else

numDays = 28;

break;

default:

System.out.println("Invalid month.");

break;

}

System.out.println("Number of Days = "

+ numDays);

}

}

This is the output from the code:

Number of Days = 29

## **Using Strings in switch Statements**

In Java SE 7 and later, you can use a String object in the switch statement's expression. The following code example, [StringSwitchDemo](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/StringSwitchDemo.java), displays the number of the month based on the value of the String named month:

public class StringSwitchDemo {

public static int getMonthNumber(String month) {

int monthNumber = 0;

if (month == null) {

return monthNumber;

}

switch (month.toLowerCase()) {

case "january":

monthNumber = 1;

break;

case "february":

monthNumber = 2;

break;

case "march":

monthNumber = 3;

break;

case "april":

monthNumber = 4;

break;

case "may":

monthNumber = 5;

break;

case "june":

monthNumber = 6;

break;

case "july":

monthNumber = 7;

break;

case "august":

monthNumber = 8;

break;

case "september":

monthNumber = 9;

break;

case "october":

monthNumber = 10;

break;

case "november":

monthNumber = 11;

break;

case "december":

monthNumber = 12;

break;

default:

monthNumber = 0;

break;

}

return monthNumber;

}

public static void main(String[] args) {

String month = "August";

int returnedMonthNumber =

StringSwitchDemo.getMonthNumber(month);

if (returnedMonthNumber == 0) {

System.out.println("Invalid month");

} else {

System.out.println(returnedMonthNumber);

}

}

}

Scenario:

* In currency conversion: to convert prices to a different currency based on the currency selected by the user.
* In product categorization: to categorize products based on different criteria, such as type, brand, etc.

In e-commerce applications, both "if-else if" and "switch" statements are commonly used to make decisions and control the flow of the application. The specific use of each statement may vary based on the requirements of the application.

Switch v/s if-else-if

* "if-else if" statement:
  + When you need to evaluate a range of values or multiple conditions for an expression.
  + When the conditions you need to check are complex and cannot be easily expressed as simple values.
  + When you need to perform different actions based on different conditions, and the conditions are not mutually exclusive.
* "switch" statement:
  + When you need to evaluate a single expression against a set of fixed, discrete values.
  + When the conditions you need to check are simple and can be expressed as simple values.
  + When the cases you need to handle are independent of each other, and you want to avoid the possibility of multiple cases being executed by mistake.
  + When you want to make your code more readable by breaking down complex conditions into simple, easily recognizable cases.

**while statement**

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

while (expression) {

statement(s)

}

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the *statement*(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false.

You can implement an infinite loop using the while statement as follows:

while (true){

// your code goes here

}

1. "while" loop:
   * When you do not know the number of iterations beforehand.
   * When you need to repeat a block of code until a certain condition is met.
   * When you want to execute the loop body only if the condition is true.

Scenario:

In product search: to search for a product based on user input until the desired product is found.

**Do While**

The Java programming language also provides a do-while statement, which can be expressed as follows:

do {

statement(s)

} while (expression);

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once.

1. "do-while" loop:
   * When you need to execute the loop body at least once, even if the condition is false.
   * When you need to repeat a block of code until a certain condition is met.
   * When you want to execute the loop body after checking the condition, instead of before.

Scenario:

In user authentication: to repeat the authentication process until the user provides valid credentials.

# **for Statement**

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

for (*initialization*; *termination*;

*increment*) {

*statement(s)*

}

When using this version of the for statement, keep in mind that:

* The *initialization* expression initializes the loop; it's executed once, as the loop begins.
* When the *termination* expression evaluates to false, the loop terminates.
* The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

When to use:

1. "for" loop:
   * When you know the number of iterations beforehand.
   * When you need to repeat a block of code a specific number of times.
   * When you want to loop over a range of numbers, an array, or any other collection that has a definite size.

Scenario:

* In order processing: to process all the items in an order and calculate the total price.
* In customer reviews: to display all the reviews for a particular product.