Question 01

Statically Typed Language

A statically typed language is a programming language in which the data types of variables are known and checked at compile time, before the program is executed. This means that variables must be explicitly declared with their data types, and the compiler enforces type safety by ensuring that operations and assignments are performed only on compatible data types. If there is a type mismatch, the compiler generates an error, preventing many potential runtime errors. Statically typed languages offer better performance and can catch certain bugs early in the development process, but they often require more explicit type annotations compared to dynamically typed languages. Examples of statically typed languages include Java, C++, and Swift.

Dynamically Typed Language

A dynamically typed language is a programming language in which the data types of variables are determined at runtime, during program execution. Unlike statically typed languages, there is no need to explicitly declare variable types; the type of a variable is automatically inferred based on the value it holds. This flexibility allows for more concise code and rapid development but can lead to potential type-related errors during runtime if not handled properly. Dynamically typed languages include Python and JavaScript.

Strongly Typed Language

A strongly typed language is a programming language that enforces strict type checking, ensuring that variables and expressions are used only in ways that are compatible with their declared data types. In a strongly typed language, the compiler or interpreter actively checks and enforces type rules, preventing certain type-related errors during compilation or runtime. This approach promotes safety and reliability, as it reduces the risk of unexpected behaviors due to type conversions or mismatched data types. Examples of strongly typed languages are Java and C#.

Loosely Typed Language

A loosely typed language is a programming language that allows for more flexible handling of data types. In contrast to strongly typed languages, loosely typed languages do not enforce strict type checking, and variables can change their data type dynamically during program execution without explicit type declarations. This flexibility can lead to concise and expressive code, but it also increases the risk of type-related errors and may require additional effort to ensure data consistency and safety. Examples of loosely typed languages include JavaScript and PHP.

Question 02

Case Sensitive:

A programming language is considered case sensitive when it distinguishes between uppercase and lowercase characters in identifiers (such as variable names, function names, and keywords). This means that 'Hello' and 'hello' are treated as two different identifiers, and the language would recognize them as separate entities.

Ex (Java): int myVariable = 10; and int MyVariable = 20; are two different variables due to case sensitivity.

Case Insensitive:

A programming language is considered case insensitive when it does not differentiate between uppercase and lowercase characters in identifiers. In this case, 'Hello' and 'hello' would be treated as the same identifier, and the language would treat them interchangeably.

Ex (SQL): SELECT * FROM myTable; and select * from MyTable; are SQL statements are equivalent due to case insensitivity.

Case Sensitive-Insensitive:

Some programming languages can be a combination of case sensitive and case insensitive. For example, variable names might be case sensitive, while keywords (reserved words) are case insensitive.

Regarding Java, it is a case-sensitive programming language. Java distinguishes between uppercase and lowercase characters in identifiers, making 'Hello' and 'hello' two distinct identifiers in Java. This means you need to be consistent with the capitalization when using variable names, class names, method names, and other identifiers in your Java code.

Question 03

In Java, an identity conversion is a type conversion that requires no actual conversion at all. It is the most straightforward form of type compatibility, where a value can be assigned to a variable or passed to a method without any modification because the source and target types are the same. The Java compiler performs identity conversions implicitly, ensuring type safety without the need for explicit casting.

ex 1:

```
public class IdentityConversionEx1 {
    public static void main(String[] args) {
        int x = 42;
        int y = x;

        System.out.println("x: " + x);
        System.out.println("y: " + y);
```

ex2:

```
public class IdentityConversionEx2 {
    public static void main(String[] args) {
        String name = "John";
        String copyOfName = name;

        System.out.println("name: " + name);
        System.out.println("copyOfName: " + copyOfName);
 }
}
```

Question 04

Primitive widening conversion in Java is an automatic type conversion that occurs when a value of a smaller data type is assigned to a variable of a larger data type. The Java compiler performs this conversion implicitly, without the need for explicit casting, to ensure that no data loss occurs during the assignment.

Example 1: byte to int

```
public class WideningConversionEx1 {
    public static void main(String[] args) {
        byte smallNumber = 10;
        int largerNumber = smallNumber;
    }
}
```

Example 2: short to long

```
public class WideningConversionEx2 {
    public static void main(String[] args) {
        short smallValue = 1000;
```

```
        long largeValue = smallValue;
    }
}
```

Question 05

In Java, the terms "run-time constant" and "compile-time constant" refer to different types of constants based on when their values are determined during the program's life cycle:

Compile-time constant:
A compile-time constant is a constant whose value is known and can be computed by the compiler at compile time. The compiler replaces references to compile-time constants with their actual values during the compilation process, and these values are fixed and cannot change during runtime.

Run-time constant:
A run-time constant is a constant whose value is determined and known only during runtime. The value of a run-time constant can be computed during program execution, and it is not known until the program runs.

Question 06

Implicit (Automatic) Narrowing Primitive Conversions and Explicit Narrowing Conversions (Casting) are two ways to convert data between primitive data types with a smaller range to those with a larger range. The key difference between them lies in how the conversions are performed and the involvement of the programmer in the process.

Implicit (Automatic) Narrowing Primitive Conversions:
Implicit narrowing primitive conversions occur automatically by the Java compiler when a value of a larger data type is assigned to a variable of a smaller data type. The conversion is performed without any explicit casting, and the compiler ensures that no data loss occurs during the conversion. This conversion happens when the target data type can represent the range of values of the source data type without losing information.

Explicit Narrowing Conversions (Casting):
Explicit narrowing conversions (casting) occur when a programmer explicitly converts a value of a larger data type to a variable of a smaller data type. This process involves explicitly stating the desired type through casting. The programmer must explicitly acknowledge the possibility of data loss during the conversion because the target data type may not be able to represent the full range of values of the source data type.

Conditions for Implicit Narrowing Primitive Conversion:
An implicit narrowing primitive conversion can occur if the following conditions are met:

1. The value of the source data type is within the valid range of the target data type.
2. No precision loss occurs during the conversion.

Question 07

The assignment of a long data type (64 bits) into a float data type (32 bits) in Java involves a type conversion known as a narrowing primitive conversion. Java allows this conversion, but it is important to understand that it may result in a loss of precision. The discrepancy arises due to the difference in how long and float data types store their values and the range

they can represent. In Java, the long data type is a 64-bit signed integer, which can represent integer values in the range of approximately -9.2 quintillion to +9.2 quintillion. On the other hand, the float data type is a 32-bit single-precision floating-point number, which can represent fractional values with a much larger range but with less precision than long.

When you assign a long value to a float variable, the compiler performs an implicit narrowing primitive conversion, which essentially truncates the long value to fit within the float's 32-bit representation. This truncation can lead to a loss of precision, particularly when the long value has significant digits beyond the 32-bit range.

Question 08

The choice of int as the default data type for integer literals and double as the default data type for floating-point literals in Java is primarily driven by a balance between performance and precision. This design decision takes into consideration the common use cases and aims to provide a practical and efficient approach to handling numeric literals.

int as the default data type for integer literals:
Java uses int as the default data type for integer literals because 32-bit integers (int) cover a wide range of practical use cases, from small to moderately large whole numbers. Most integer computations fall within this range, making int the most efficient choice in terms of memory consumption and performance. Additionally, modern computer architectures are optimized for 32-bit integer operations, which further enhances the performance of int.

double as the default data type for floating-point literals:
Java uses double as the default data type for floating-point literals because 64-bit double-precision floating-point numbers offer a good balance between precision and performance. double provides a high level of accuracy for a wide range of real-world applications, such as scientific computations, financial calculations, and graphics rendering. It can represent a vast range of values with reasonable precision and is the most commonly used floating-point type in many programming languages. The use of double as the default floating-point type is influenced by the IEEE 754 standard, which defines the representation and arithmetic behavior of floating-point numbers. The widespread adoption of this standard in hardware and software implementations makes double the most efficient choice for handling floating-point computations.
By defaulting to int and double, Java ensures that most numeric calculations are efficient and reasonably accurate for typical programming tasks. However, programmers have the flexibility to explicitly specify other data types when more precision or performance is required, or when dealing with specific use cases like large integers (long) or higher precision floating-point numbers (BigDecimal).

Question 09

Implicit narrowing primitive conversion only takes place among byte, char, int, and short because these data types are considered to be the most common and frequently used integer types in Java. Java's language design prioritizes simplicity and ease of use, and allowing implicit narrowing conversions only among these types strikes a balance between convenience and safety. The four types (byte, char, int, and short) are all 32-bit signed integers (except char, which is a 16-bit unsigned integer), and their range of representable values is related. Therefore, implicit narrowing conversions between these types do not result in a significant loss of information in most common scenarios. On the other hand, Java is designed to be a strongly typed language, emphasizing type safety to avoid potential errors and bugs. Allowing implicit narrowing conversions among these closely related types provides a degree of flexibility while still enforcing some level of type safety. However, going beyond these

types for implicit narrowing conversions could lead to more ambiguous or error-prone situations, especially with larger data types like long and float. By restricting implicit narrowing conversions to these four types, Java encourages developers to be explicit about type conversions when there is a possibility of data loss or when working with more significant differences in data types. In such cases, explicit casting is required to indicate that the programmer is aware of the potential loss of information. In summary, implicit narrowing primitive conversions are limited to byte, char, int, and short in Java to strike a balance between convenience and safety. It provides flexibility for common use cases while still promoting type safety and making the language less prone to subtle errors related to type conversions. For cases outside these four types, explicit casting is used to signal the programmer's intention and awareness of the potential consequences of the conversion.