1.

Statically Typed Language

Statically typed languages are those in which data types are explicitly declared for variables at compile-time. This means that the data type of each variable must be known and specified before the program is executed. The compiler checks for type correctness during compilation, and any type errors are caught early in the development process. C, C++, and Java are examples of statically typed languages.

Dynamically Typed Language

Dynamically typed languages are those in which data types are determined at runtime, i.e., t he data type of a variable is resolved during program execution. Unlike statically typed languages, we don't need to declare the data type of variables explicitly. Python, JavaScript, and Ruby are examples of dynamically typed languages.

Strongly Typed Language

Strongly typed languages are those in which type-checking is strictly enforced, and implicit type conversion is limited. In a strongly typed language, we cannot perform operations that involve incompatible data types without explicit type conversion. This ensures type safety and reduces the risk of unintended errors. Java, C++, and Python are examples of strongly typed languages.

Loosely Typed Language

Loosely typed languages are those in which type-checking is more lenient, and automatic type conversion (coercion) between different data types is allowed. Variables can change their data type on the fly, and the language will implicitly perform type conversions when required. PHP is an example of a loosely typed language.

Java falls into the category of a **Statically Typed Language** and a **Strongly Typed Language.** Also Java can be considered as Dynamically typed Language.

2.

Case Sensitive

Case sensitivity refers to the distinction made between uppercase and lowercase letters in programming languages. In a case-sensitive language, variables, function names, keywords, and identifiers must be spelled with the exact casing (uppercase or lowercase) as they are declared. This means that "Variable" and "variable" would be considered two different identifiers in a case-sensitive language.

Case Insensitive

Case insensitivity means that uppercase and lowercase letters are treated as the same in programming languages. In a case-insensitive language, identifiers are not distinguished based on casing, so "Variable" and "variable" would be considered the same identifier.

Case Sensitive-Insensitive (Mixed Case Sensitivity)

Some programming languages are case-sensitive for some parts and case-insensitive for others. For example, variable names might be case-sensitive, while function names are case-insensitive.

Java is a Case-Sensitive language. It makes a clear distinction between uppercase and lowercase letters in identifiers, keywords, and other parts of the code.

3.

In Java, an Identity Conversion is a type of casting or conversion where no actual conversion is performed on the value. It's a special kind of conversion that is allowed when the target type is the same as the source type, or when dealing with compatible primitive types.

For primitive types, an identity conversion can occur when the types are the same (e.g., int to int) or when they are compatible, meaning the conversion does not lose information (e.g., widening conversions like int to long).

Example 1: Identity Conversion with Primitive Types

```
int intValue = 42
long longValue = intValue          // Identity conversion from int to long (widening conversion)
System.out.println(longValue)      // Output: 42
```

In this example, we assign an int value to a long variable. This is an identity conversion since int can be implicitly converted to long without any data loss.

Example 2: Identity Conversion with Reference Types

```
class MyClass {
    // Some class members and methods
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        MyClass sameObj = obj; // Identity conversion from MyClass
```

```
to MyClass
        System.out.println(obj == sameObj); // Output: true (both
variables refer to the same object)
    }
}
```

In this example, we have a class MyClass. When we assign an instance of MyClass to another variable of the same type, it is an identity conversion. The sameObj now references the same object as obj.

In both examples, the identity conversion is possible because the source and target types are the same, or they are compatible with each other. The Java compiler allows identity conversions without any explicit casting, as it does not involve any data loss or potential errors.

4. Primitive Widening Conversion, also known as Implicit Widening Conversion or Widening Conversion, is a type of type conversion in Java where a value of a narrower data type is automatically and safely promoted to a value of a wider data type without any explicit casting. This conversion is safe because it does not lead to any loss of data or precision. Java supports primitive widening conversions for numeric types when the target type can represent all possible values of the source type.

   The following are the widening conversion rules for the numeric data types in Java:

   From byte to short, int, long, float, or double.
   From short to int, long, float, or double.
   From char to int, long, float, or double.
   From int to long, float, or double.
   From long to float or double.
   From float to double.

   Example 1: From int to long

```java
public class WideningConversionExample {
    public static void main(String[] args) {
        int intValue = 42;
        long longValue = intValue; // Widening conversion from int to long
        System.out.println(longValue);
    }
}
```

   In this example, we have an int variable intValue with the value 42. We then assign this value to a long variable longValue. Since long can represent a wider range of values than int, there's no loss of data or precision, and Java performs the widening conversion automatically.

Example 2: From char to int

```java
public class WideningConversionExample {
    public static void main(String[] args) {
        char charValue = 'A';
        int intValue = charValue; // Widening conversion from char to int
        System.out.println(intValue);
    }
}
```

In this example, we have a char variable charValue with the value 'A'. We then assign this value to an int variable intValue. Since int can represent a wider range of values than char, there's no loss of data or precision, and Java performs the widening conversion automatically.

5. In Java, both run-time constants and compile-time constants represent values that are fixed and cannot be changed during the execution of the program. However, the main difference between them lies in when their values are determined:

   Compile-time Constants
   Compile-time constants are constants whose values are known and can be evaluated by the Java compiler during the compilation phase itself. These constants are resolved at compile-time, and their values are directly substituted into the code wherever they are used. As a result, no computation is required at runtime to determine their values.
   To be considered a compile-time constant, a variable must meet the following criteria:

   The variable must be declared as final.
   The variable must be initialized with a constant expression.

   Example of Compile-time Constant:

```
public class CompileTimeConstantExample {
        public static final int MY_CONSTANT = 42; // Compile-time constant
        public static final String GREETING = "Hello"; // Compile-time constant

    public static void main(String[] args) {
            int sum = MY_CONSTANT + 10; // The value of MY_CONSTANT is known at
                                                    compile-time.
        System.out.println(GREETING + " World!"); // GREETING is known at compile-
                                                                    time.
                }
        }
```

   In this example, MY_CONSTANT and GREETING are both compile-time constants because they are declared as final and initialized with constant expressions. The compiler

knows their values at compile-time and can directly substitute them into the code.

Run-time Constants:
Run-time constants, on the other hand, are constants whose values can only be determined at runtime, during the execution of the program. These constants are typically computed or initialized within methods, constructors, or static initializers.

Example of Run-time Constant:

```java
import java.util.Random;

public class RunTimeConstantExample {
    public static final int RANDOM_CONSTANT;

    static {
        // The value of RANDOM_CONSTANT is determined at runtime.
        RANDOM_CONSTANT = new Random().nextInt(100);
    }

    public static void main(String[] args) {
        System.out.println("Random Constant: " + RANDOM_CONSTANT);
    }
}
```

In this example, RANDOM_CONSTANT is a run-time constant because its value is determined at runtime within the static initializer block. The Java compiler cannot evaluate its value during compilation because it depends on the result of the nextInt() method from the Random class.

In summary, compile-time constants have their values known and evaluated by the Java compiler during compilation, whereas run-time constants have their values determined during the execution of the program at runtime.

6.

Implicit (Automatic) Narrowing Primitive Conversions:
Implicit narrowing primitive conversions, also known as automatic narrowing conversions, occur when a value of a wider data type is assigned to a variable of a narrower data type without any explicit casting. These conversions can result in potential loss of data or precision because the target type cannot represent all possible values of the source type.

Examples of implicit narrowing primitive conversions:

```
int intValue = 1000;
byte byteValue = intValue; // Implicit narrowing conversion from int to byte
```

In this example, the intValue (int) is automatically narrowed to byteValue (byte) without any explicit casting. However, the value 1000 cannot be represented in a byte (which has a range of -128 to 127), so data loss occurs, and byteValue will be -24.

Explicit Narrowing Conversions (Casting):

Explicit narrowing primitive conversions, also known as casting, occur when a value of a wider data type is explicitly converted to a variable of a narrower data type using parentheses and the desired target type. This type of conversion informs the compiler that the programmer is aware of the potential data loss and explicitly allows it.

Examples of explicit narrowing primitive conversions:

```
int intValue = 1000;
byte byteValue = (byte) intValue; // Explicit narrowing conversion (casting) from int to byte
```

In this example, we use explicit casting to convert the intValue to a byte by using (byte) before the intValue. The value 1000 will be truncated to fit into a byte, and byteValue will be -24, similar to the implicit conversion example.

Conditions for Implicit Narrowing Primitive Conversions:
For an implicit narrowing primitive conversion to occur without any explicit casting, the following conditions must be met:

- The value being assigned to the narrower data type must be within the valid range of the target type.
- Should be a constant.

If the value being assigned exceeds the range of the target type, or if the target type is larger than the source type, an explicit narrowing conversion (casting) is required to inform the compiler that data loss may occur intentionally.

7. In Java, when a value of a data type with a larger size, such as long (64 bits), is assigned to a data type with a smaller size, such as float (32 bits), an implicit narrowing primitive conversion occurs. This conversion involves potential data loss, as float has a smaller range of representable values compared to long.

   The seeming discrepancy arises from the fact that the float data type in Java uses some of its bits to represent the exponent and some to represent the mantissa (fractional part). As a result, it can represent a wide range of values, but with limited precision.

   Here's how it works:

   The long data type uses all its 64 bits to represent the whole number value without any loss of precision.

The float data type uses 1 bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa (fractional part).

When you assign a long value to a float variable, the value is converted to a floating-point representation. This conversion involves adjusting the exponent and mantissa to fit into the float format.

If the long value has a large number of significant digits (large magnitude), it may lose precision when converted to a float, as the float can only represent a subset of values within its 32-bit representation.

Due to the limited precision of the float data type, some long values may not be accurately represented in float, resulting in potential rounding errors or loss of precision.

Here's an example to illustrate the discrepancy:

```
public class LongToFloatConversionExample {
        public static void main(String[] args) {
                long longValue = 1234567890123456789L;
                float floatValue = longValue; // Implicit narrowing conversion from long to
                                                                                float

                System.out.println("longValue: " + longValue);
                 System.out.println("floatValue: " + floatValue);
                 }
                }
```

In this example, longValue is a large long value (1234567890123456789L). When we assign it to the float variable floatValue, the implicit narrowing conversion takes place. The floatValue will not accurately represent the exact longValue due to the limited precision of the float data type. As a result, the output will show a discrepancy:

```
longValue: 1234567890123456789
floatValue: 1.23456792E18
```

As you can see, the floatValue is not the exact same as longValue, and some precision has been lost during the conversion. This is why it is essential to be cautious when performing implicit narrowing conversions, as they can lead to potential data loss or rounding errors. If precision is critical, you may need to use explicit casting and be aware of the limitations of the target data type.

8.

he Java Language Specification (JLS) provides the rationale behind the design decision to set int as the default data type for integer literals and double as the default data type for floating-point literals.

Reference to the Java Language Specification (JLS):

Chapter 3 - "Lexical Structure" (JLS 3.10.1) defines the syntax for integer literals and

floating-point literals. Here are the relevant excerpts:

3.10.1. Integer Literals:

An integer literal is of type long if it is suffixed with an ASCII letter L or l; otherwise, it is of type int. The suffix L is preferred, because the letter l is often hard to distinguish from the digit 1.

3.10.2. Floating-Point Literals:

A floating-point literal is of type float if it is suffixed with an ASCII letter F or f; otherwise, it is of type double and can optionally be suffixed with an ASCII letter D or d (§4.2.3).

Rationale Behind the Design Decision:
The Java Language Specification does not explicitly state the rationale in these sections. However, the design decision can be understood based on historical reasons, practicality, and backward compatibility with C and C++.

Historical Reasons: Java was initially designed as an alternative to C and C++ to be used on small devices. The decision to use int and double as default data types for numeric literals aligns with the practices in C and C++, making it easier for developers familiar with these languages to transition to Java.

Practicality and Usability: Integer literals and floating-point literals are used more frequently in Java code compared to other numeric types. By setting int and double as default data types, the language aims to make common numeric literals more concise and straightforward to write without requiring explicit type declarations in most cases.

Backward Compatibility with C/C++: Java borrowed several syntactical conventions from C and C++ to make the language familiar to developers coming from those backgrounds. Setting int as the default data type for integers and double for floating-point numbers ensures compatibility with C and C++ code, making it easier for developers to adapt to Java.

Performance Considerations: The JVM (Java Virtual Machine) is optimized for handling int and double efficiently. Operations involving these data types are generally faster and more efficient than others. Using these defaults encourages developers to write code that performs well on the JVM, reducing overhead associated with other data types.

Despite these default choices, Java remains a strongly typed language, and developers can always explicitly specify other data types for literals when needed. This explicitness helps improve code clarity, precision, and avoids potential issues related to data type conversions.

In summary, the design decision to set int as the default data type for integer literals and double for floating-point literals in Java was influenced by historical reasons, practicality, usability, backward compatibility with C/C++, and performance considerations in the JVM.

9. Implicit narrowing primitive conversions in Java are limited to certain data types for safety reasons and to avoid potential loss of data. The conversions are allowed among byte, char, int, and short because these data types are considered to be relatively small and of similar size in terms of the number of bits used to represent them.

The explicit rules for implicit narrowing primitive conversions are defined in the Java Language Specification (JLS) in Section 5.1.3 - "Narrowing Primitive Conversion." The relevant excerpt from the specification is as follows:

A narrowing primitive conversion may lose information about the overall magnitude of a numeric value and may also lose precision and range.

[...]

The following 19 specific conversions on primitive types are called the narrowing primitive conversions:

byte to short, char, or int
short to char or int
char to int
int to short or byte
long to int, short, byte, or char
float to int, short, byte, char, or long
double to int, short, byte, char, long, or float
The reason for limiting implicit narrowing conversions to these specific types is to prevent unintended data loss when converting between data types with significantly different ranges. Allowing implicit narrowing conversions between larger data types, such as long and double, and smaller ones, such as int, short, or byte, could lead to potential loss of data or precision, which might not be immediately apparent to the programmer.

By limiting the implicit narrowing conversions to the subset of byte, char, int, and short, the Java language ensures that developers are explicitly aware of possible data loss when narrowing data types. This helps in writing more robust and predictable code.

If a programmer needs to perform an implicit narrowing conversion between larger and smaller data types, they can explicitly use casting to indicate that they are aware of the potential data loss and that it is intentional. This explicitness ensures better code readability and allows the programmer to handle any data loss issues carefully.

10. Widening and Narrowing Primitive Conversion are two types of type conversions used for primitive data types in Java.

    Widening (Implicit) Primitive Conversion:
    Widening conversion, also known as implicit conversion, occurs when a value of a smaller data type is automatically and safely promoted to a value of a larger data type without any explicit casting. The destination data type can represent all possible values of the source data type without any loss of data or precision. Widening conversions do not require explicit casting because they are safe and don't lead to any data loss.

    Example of Widening Conversion:

    ```
    int intValue = 10;
    long longValue = intValue; // Widening conversion from int to long
    ```

    In this example, intValue (int) is automatically promoted to longValue (long) without any

explicit casting. It is safe because long can represent a larger range of values than int.

Narrowing (Explicit) Primitive Conversion:

Narrowing conversion, also known as explicit conversion or casting, occurs when a value of a larger data type is explicitly converted to a value of a smaller data type. This conversion is potentially unsafe because the destination data type may not be able to represent all possible values of the source data type. Narrowing conversions require explicit casting to indicate that the programmer is aware of the potential data loss, and it must be done carefully to avoid loss of precision or unexpected results.

Example of Narrowing Conversion:

```
long longValue = 1234567890123L;
int intValue = (int) longValue; // Narrowing conversion (casting) from long to int
```

In this example, we explicitly use casting (int) to convert the longValue (long) to intValue (int). This is potentially unsafe because int cannot represent the full range of values that a long can, so some data loss may occur.

Now, let's address the question of why the conversion from short to char is not classified as both Widening and Narrowing Primitive Conversion:

short to char conversion:

```
short shortValue = 65;
char charValue = (char) shortValue; // short to char conversion (narrowing with casting)
```

The reason why the conversion from short to char is not classified as either widening or narrowing is because both short and char are 16-bit data types in Java, and they have the same size. Since they are of the same size, there is no data loss in the conversion, and it doesn't fit into the typical classification of widening (promotion to a larger data type) or narrowing (conversion to a smaller data type).

In this specific case, the conversion involves explicit casting ((char)) because Java requires explicit casting for conversions between unrelated data types, even if they are of the same size. This is done to ensure that developers are aware of potential data loss, even though no actual data loss occurs between short and char in this particular conversion.