Q1:-
Statically Typed Language:
1. Data types of variables are determined and checked at **compile time**.
2. Variables must be explicitly declared with their data types before use.
3. Type checking ensures only compatible operations are performed on variables.
4. Types are fixed and cannot be changed during runtime.
5. Examples: Java, C, C++, Swift.

Dynamically Typed Language:
1. Data types of variables are determined and checked at **runtime** (during execution).
2. Variables do not require explicit type declarations; types are inferred from assigned values.
3. Offers flexibility but may lead to type-related errors during runtime.
4. Examples: Python, JavaScript, Ruby, PHP.

Strongly Typed Language:
1. Type safety is strictly enforced, and type conversions are not performed implicitly.
2. Data types are fixed and cannot be implicitly converted into other types without explicit typecasting.
3. Early detection of type-related errors during compilation.
4. Applies to both statically and dynamically typed languages.
5. Examples: Java, C#, Swift, Haskell.

Loosely Typed Language (Weakly Typed Language):
1. The type system is more flexible, allowing automatic type conversions.
2. Variables can change their data type without explicit typecasting.
3. May lead to unexpected results if not handled carefully.
4. Emphasizes ease of use over strict type safety.
5. Examples: JavaScript, PHP.

Java falls into the category of:
- A **statically typed** language because data types are checked at compile time.
- A **strongly typed** language because it enforces strict type safety and doesn't allow implicit type conversions.

##
Q2:-
1. Case Sensitive:
- Case sensitivity means that the programming language distinguishes between uppercase and lowercase characters in identifiers.
- Identifiers with different cases are treated as distinct entities.
- For example, in a case-sensitive language:
  - `variable` and `Variable` are considered two different variables.
  - `functionName` and `FunctionName` are considered two different function names.

*Example of Case Sensitive Programming Language:
- Python is a case-sensitive language. In Python, `variable` and `Variable` are treated as different variables.

2. Case Insensitive:
- Case insensitivity means that the programming language treats uppercase and lowercase characters in identifiers as equivalent.
- Identifiers written with different cases are considered the same entity.

- For example, in a case-insensitive language:
  - `variable` and `Variable` are considered the same variable.
  - `functionName` and `FunctionName` are considered the same function name.

*Example of Case Insensitive Programming Language:
- SQL (Structured Query Language) is case-insensitive for identifiers. In SQL, `SELECT`, `Select`, and `select` are all treated as the same keyword.

3. Case Sensitive-Insensitive (Partial Case Insensitive):
- Some programming languages exhibit partial case sensitivity, where the treatment of case sensitivity depends on specific contexts.
- For instance, the language may be case-sensitive for variable names but case-insensitive for keywords.
- In this scenario:
  - `variable` and `Variable` would be considered distinct variables.
  - `if`, `IF`, and `If` would be considered the same keyword.

*Example of Case Sensitive-Insensitive Programming Language:
- JavaScript is an example of a partially case-sensitive language. Variable names are case-sensitive, but JavaScript keywords like `if`, `for`, `while`, etc., are case-insensitive.

**Java in Relation to These Terms:
- Java is a **case-sensitive** programming language. It differentiates between uppercase and lowercase characters in identifiers.
- For example, `variable` and `Variable` would be considered two different variables in Java, and `if`, `IF`, and `If` would be treated as different keywords in Java.

##
Q3:-
In Java, identity conversion is a type conversion where a value is explicitly or implicitly converted to its own type without any modification. It is a trivial conversion that ensures type compatibility, and no data loss or change in representation occurs during this process. Identity conversion is applied when the target type is the same as the source type, making the conversion unnecessary.

Two examples of identity conversion in Java:

*Example 1: Assigning a value to a variable of the same type:

***java

```
public class IdentityConversionExample1 {
    public static void main(String[] args) {
        int number = 42;
        int anotherNumber = number; // Identity conversion: Assigning 'number' to 'anotherNumber'
        System.out.println("Value of anotherNumber: " + anotherNumber); // Output: Value of anotherNumber: 42
    }
}
```

In this example, an integer variable `number` is assigned the value `42`. Then, `anotherNumber` is assigned the value of `number`, which is an example of identity conversion. Both `number` and

`anotherNumber` are of the same type (`int`), so the conversion is not necessary. The value is directly copied from `number` to `anotherNumber`.

*Example 2: Passing an argument to a method with the same type parameter:

***java

```
public class IdentityConversionExample2 {
    public static void main(String[] args) {
        String message = "Hello, Java!";
        printMessage(message); // Identity conversion: Passing 'message' as an argument
    }

    public static void printMessage(String msg) {
        System.out.println("Message: " + msg); // Output: Message: Hello, Java!
    }
}
```

In this example, a `String` variable `message` is initialized with the value `"Hello, Java!"`. The `printMessage` method takes a `String` parameter `msg`, and we call this method, passing `message` as an argument. The argument passing is an example of identity conversion, as both the argument (`message`) and the parameter (`msg`) are of the same type (`String`). The conversion is not required, as the types match, and the value of `message` is directly passed to the method.

In both examples, identity conversion is utilized to ensure type compatibility without any change in the value or type representation. It demonstrates the efficiency and convenience of handling same-type conversions without the need for any actual conversion process.

##
Q4:-
Primitive widening conversion in Java is an automatic type conversion that occurs when a data value of a smaller primitive type is assigned to a variable of a larger primitive type. In this process, the value is widened to fit into the larger data type without any loss of data. Java performs primitive widening conversions implicitly, as they involve widening the range of values and, therefore, do not result in any data loss or precision issues.

The primitive data types in Java and their sizes, in increasing order of size, are as follows:

1. byte (1 byte)
2. short (2 bytes)
3. int (4 bytes)
4. long (8 bytes)
5. float (4 bytes)
6. double (8 bytes)

The implicit primitive widening conversions are as follows:

byte → short → int → long → float → double

Now, let's demonstrate primitive widening conversions with examples and diagrams:

*Example 1: byte to short

***java
public class PrimitiveWideningExample {
    public static void main(String[] args) {
        byte b = 100;
        short s = b; // Widening conversion: byte to short
        System.out.println("Value of s: " + s); // Output: Value of s: 100
    }
}

*Example 2: int to long

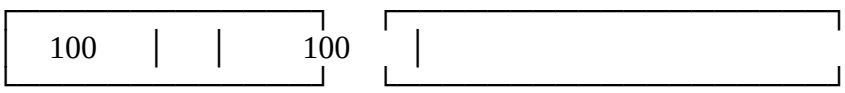
***java
public class PrimitiveWideningExample {
    public static void main(String[] args) {
        int num = 42;
        long bigNum = num; // Widening conversion: int to long
        System.out.println("Value of bigNum: " + bigNum); // Output: Value of bigNum: 42
    }
}

In both examples, we can observe the following:

1. The source variable (`b` in Example 1 and `num` in Example 2) has a smaller data type (byte and int, respectively).
2. The target variable (`s` in Example 1 and `bigNum` in Example 2) has a larger data type (short and long, respectively).
3. The assignment statements (`short s = b;` and `long bigNum = num;`) perform primitive widening conversions automatically.

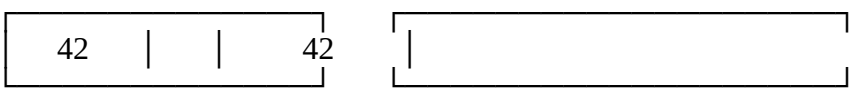
*Diagrammatic Representation:
The diagrams below illustrate how primitive widening conversions work:

***
Example 1: byte to short
byte (8 bits) → short (16 bits)



(b) (s)

Example 2: int to long
int (32 bits) → long (64 bits)



(num) (bigNum)

In Example 1, the byte value `100` is widened to a short without any loss of data. Similarly, in Example 2, the int value `42` is widened to a long without any data loss. These conversions ensure that the values fit safely into the larger data types without compromising precision.

##
Q5:-
In Java, both run-time constants and compile-time constants are values that remain constant during the execution of a program. However, there is a fundamental difference in when their values are determined and how they are handled during the compilation and runtime phases.

*Compile-time Constant:
1. A compile-time constant is a value that is known to the compiler at compile time itself, and its value is determined during the compilation phase.
2. These constants are replaced directly with their actual values at compile time, and no memory allocation occurs for them during runtime.
3. Compile-time constants are typically used for optimization purposes, as they eliminate the need to look up the value during program execution.
4. Primitive data types, final variables (assigned with a constant expression), and constants declared with the `static final` keyword are examples of compile-time constants.

*Example of Compile-time Constant:

***java

```
public class CompileTimeConstantExample {
    public static final int NUMBER = 42; // This is a compile-time constant

    public static void main(String[] args) {
        int result = NUMBER * 2; // At compile time, NUMBER is replaced with its value (42)
        System.out.println("Result: " + result); // Output: Result: 84
    }
}
```

In this example, the variable `NUMBER` is declared as a `static final int`, making it a compile-time constant. During compilation, wherever `NUMBER` is used, it is replaced directly with its value (42). So, at runtime, there is no lookup required for `NUMBER`, resulting in optimized code.

*Run-time Constant:

1. A run-time constant is a value whose determination requires computation or method invocation at runtime, and its value is evaluated during program execution.
2. Run-time constants are not directly replaced with their values during compilation, and memory allocation occurs for them during runtime.
3. Constants defined with `final` that are not compile-time constants (e.g., initialized with non-constant expressions or method calls) are considered run-time constants.

*Example of Run-time Constant:

***java

```
public class RunTimeConstantExample {
```

```
    public static final int NUMBER = calculateNumber(); // This is a run-time constant

    public static int calculateNumber() {
        return 20;
    }

    public static void main(String[] args) {
        int result = NUMBER * 2; // At runtime, calculateNumber() is invoked to get the value of
NUMBER (20)
        System.out.println("Result: " + result); // Output: Result: 40
    }
}
```

In this example, the variable `NUMBER` is declared as `static final int`, but its value is determined at runtime by calling the method `calculateNumber()`. During compilation, `NUMBER` is not replaced with its value, and at runtime, the method `calculateNumber()` is invoked to get the actual value of `NUMBER` (20).

In summary, compile-time constants are known to the compiler and replaced with their values at compile time, while run-time constants require computation or method invocation at runtime to determine their values.

##
Q6:-
In Java, primitive type conversions can be categorized into two types: implicit (automatic) narrowing primitive conversions and explicit narrowing conversions (casting).

1. Implicit (Automatic) Narrowing Primitive Conversions:
- Implicit narrowing conversions occur automatically when a value of a larger data type is assigned to a variable of a smaller data type.
- These conversions are safe when the value being converted can be represented within the smaller data type without any loss of data or precision.
- Implicit narrowing conversions happen automatically without the need for explicit casting and are performed when the compiler determines that no data loss will occur.
- Examples of implicit narrowing conversions: int to byte, double to float.

*Example of Implicit Narrowing Conversion:

***java
```
public class ImplicitNarrowingExample {
    public static void main(String[] args) {
        int intValue = 1000;
        byte byteValue = intValue; // Implicit narrowing: int to byte
        System.out.println("Value of byteValue: " + byteValue); // Output: Value of byteValue: -24
    }
}
```

In this example, the int value `1000` is automatically narrowed to a byte value. Since the range of byte is from -128 to 127, the int value `1000` cannot be represented in a byte without loss of data, and the result is truncated (1000 % 256 = -24).

2. Explicit Narrowing Conversions (Casting):

- Explicit narrowing conversions, also known as casting, are performed when a value of a larger data type needs to be explicitly converted to a variable of a smaller data type.
- Casting is done using parentheses and specifying the target data type before the value.
- Casting may result in data loss or loss of precision if the value cannot be represented accurately in the target data type.
- Examples of explicit narrowing conversions: int to short, double to int.

*Example of Explicit Narrowing Conversion (Casting):

***java

```
public class ExplicitNarrowingExample {
    public static void main(String[] args) {
        double doubleValue = 123.456;
        int intValue = (int) doubleValue; // Explicit narrowing: double to int
        System.out.println("Value of intValue: " + intValue); // Output: Value of intValue: 123
    }
}
```

In this example, the double value `123.456` is explicitly narrowed to an int value using casting. The fractional part of the double value is discarded, resulting in `intValue` having the value `123`.

*Conditions for Implicit Narrowing Primitive Conversion:

For an implicit narrowing primitive conversion to occur, the following condition must be met:
- The value being converted must be within the valid range of the target data type. If the value exceeds the range of the target data type, a compilation error will occur, and explicit casting is required to perform the conversion.

In summary, implicit narrowing primitive conversions happen automatically when the value can be safely represented within the smaller data type, while explicit narrowing conversions (casting) are used when the value may result in data loss or is outside the range of the target data type.

##
Q7:-
When assigning a 64-bit long data type to a 32-bit float data type in Java, type casting is used. However, due to the smaller size and limited precision of the float data type, there may be a loss of precision during the conversion. It's important to be cautious about potential data loss when performing this type conversion. If precision is crucial, consider using the `double` data type instead, which provides better precision with 64 bits.

##
Q8:-
1. Historical Reasons:
When Java was initially designed, the `int` data type was chosen as the default for integer literals because it strikes a balance between being able to represent a wide range of whole numbers efficiently while also being memory-friendly. Similarly, the `double` data type was selected as the default for floating-point literals because it provides a reasonable compromise between precision and performance for most general-purpose calculations.

2. Performance Considerations:
The `int` and `double` data types are native data types in Java, which means they can be processed more efficiently by the underlying hardware and the Java Virtual Machine (JVM). Operations on `int` and `double` values can be faster compared to other data types because they map directly to the native machine representations for integers and floating-point numbers.

3. Language Design Principles:
Java aims to be a simple and pragmatic language, promoting ease of use and readability. Using `int` for integer literals and `double` for floating-point literals aligns with these principles as they are the most commonly used and widely understood data types for their respective categories.

4. Implicit Narrowing and Widening Conversions:
Java allows implicit narrowing conversions for assigning smaller data types to larger ones (e.g., assigning an `int` to a `long`). However, implicit widening conversions (assigning a larger data type to a smaller one) are not allowed by default to avoid potential loss of data or precision. Choosing `int` and `double` as defaults reduces the need for explicit casting in most scenarios.

##
Q9:-
Implicit narrowing primitive conversion is limited to `byte`, `char`, `int`, and `short` in Java because these integral data types have a smaller range than `int` (32 bits). Restricting implicit narrowing ensures safety by avoiding potential data loss and maintaining predictable code behavior. Explicit casting is required for conversions between larger and smaller data types to handle data loss explicitly.

##
Q10:-
*Widening Primitive Conversion:
It occurs when a value of a smaller data type is assigned to a variable of a larger data type without data loss. Examples include assigning `int` to `long` or `float` to `double`.

*Narrowing Primitive Conversion:
It happens when a value of a larger data type is assigned to a variable of a smaller data type. Data loss may occur. Examples include assigning `long` to `int` or `double` to `float`.

*Conversion from short to char:

The conversion from `short` to `char` is not classified as either widening or narrowing because both data types have the same size (16 bits) and their value ranges overlap (0 to 65535). Converting between `short` and `char` doesn't lead to data loss; thus, it's considered an "identity conversion" rather than a widening or narrowing conversion.