1.

a. Statically Typed Language: At compilation time, variables' data types are explicitly defined and verified in statically typed languages. This implies that during the writing of the code, each variable's type must be known and specified. Following compilation, the compiler verifies type compatibility and makes sure that only compatible actions are carried out on variables. Java, C, C++, and Swift are a few examples of statically typed languages.

b. A dynamically typed language, in contrast to a statically typed language, does not necessitate the explicit declaration of data types for variables. Instead, as the program is being executed, the data type is decided. Variables can store data of any type, and they can have their types change as the program is run. While allowing for shorter code, this flexibility raises the possibility of type-related runtime mistakes. Languages that use dynamic typing include Python, JavaScript, and Ruby.

c. Strongly typed languages encourage stringent type checking and forbid implicit type conversions unless they are safe and explicit. In other words, it makes sure that only compatible data types are used for operations. To avoid unintentional type-related problems, the compiler or interpreter raises an error whenever a mismatch arises. Strongly typed languages can exist in both statically and dynamically typed languages. Strongly typed languages include Java, C#, and Haskell as examples.

d. Weakly Typed Language: Weakly typed languages, usually referred to as loosely typed languages, are less rigid about type compatibility and permit implicit type conversions. In operations involving various data types, variables can therefore be used without explicit conversions. While this may provide greater flexibility, it may also result in subtle flaws and make it more challenging to comprehend how the code behaves. PHP and certain earlier iterations of JavaScript are two examples of weakly typed languages.

Java is both statically typed and strongly typed.

2.

Case Sensitive: In a case-sensitive programming language, the distinction between uppercase and lowercase characters matters. This means that variables, functions, keywords, and other identifiers must be written with consistent casing for the program to recognize them correctly. For example, "variableName" and "variablename" would be treated as two different identifiers in a case-sensitive language.

Example in Python (case-sensitive):

java

# Correct usage

int variableName = 42;

String variablename = "Hello";

Case Insensitive: In a case-insensitive programming language, the distinction between uppercase and lowercase characters is not relevant. This means that variables, functions, keywords, and other identifiers can be written with different casings, and the program will treat them as the same. For example, "variableName" and "variablename" would be considered equivalent in a case-insensitive language.

Example in SQL (case-insensitive):

-- These queries are considered equivalent in a case-insensitive language

SELECT \* FROM tableName;

select \* from tablename;

Case Sensitive-Insensitive: Some programming languages are case insensitive for keywords yet case sensitive for some aspects (such variable names and function names). It's common to refer to this combination as case sensitive-insensitive.

Java is a case-sensitive language. It differentiates between uppercase and lowercase characters in variable names, function names, and other identifiers.

3.

The easiest and clearest sort of conversion is identity conversion. Without the requirement for casting or conversion, it happens when a value is directly assigned to a variable of the same data type. In other words, the data types of the value and the variable are identical, preventing any data loss or lack of accuracy from happening during the assignment.

Threr are two type of identity conversions in Java.

* Identity conversion with ‘int’

int x = 42;

int y = x; // Identity Conversion - Assigning an int value to an int variable

* Identity conversion with ‘double’

double pi = 3.14159;

double approximation = pi; // Identity Conversion - Assigning a double value to a double variable

4.

Primitive Widening Conversion, sometimes referred to as Implicit Type Conversion or Automatic Type Promotion, is a Java concept where a smaller numeric value's data type is automatically promoted to a bigger data type without any explicit casting or information loss. When various data types are involved in an operation, Java will automatically promote the smaller data type to the bigger one in order to preserve compatibility and avoid data loss.

The following is a list of Java's numeric data types, from smallest to largest:

byte, short, int, long, long, float, double;

ex:

int intValue = 10;

double doubleValue = 3.14;

double result = intValue + doubleValue;

5.

Compile-time Constants: Compile-time constants are constants whose values are known and evaluated by the Java compiler during the compilation phase. These constants are replaced with their actual values in the compiled bytecode, and they do not require any memory allocation at runtime.

final int MY\_CONSTANT = 42;

Run-time Constants: Run-time constants are constants whose values are determined and allocated during the program's runtime. These constants are not known to the compiler during the compilation phase and require memory allocation at runtime.

final int SIZE;

SIZE = 100;

6.

Implicit (Automatic) Narrowing Primitive Conversions:

Implicit narrowing conversions, also known as automatic narrowing conversions, occur when Java automatically converts a value from a larger data type to a smaller data type without the need for explicit casting. This type of conversion is performed when the target data type can safely represent the value of the source data type without any loss of data or precision.

int intValue = 42;

byte byteValue = intValue; // Implicit narrowing conversion - int to byte

Explicit Narrowing Conversions (Casting):

Explicit narrowing conversions, or casting, are performed when a value needs to be explicitly converted from a larger data type to a smaller data type, even though data loss or precision loss may occur. This type of conversion is done using a cast operator and is necessary when the compiler cannot guarantee that the conversion will not lead to data loss.

double doubleValue = 3.14;

int intValue = (int) doubleValue; // Explicit narrowing conversion - double to int

7.