**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.

**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.

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.

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.

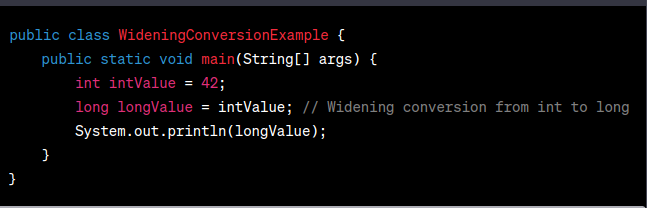
From char to int, long, float, or double.

From int to long, float, or double.

From long to float or double.

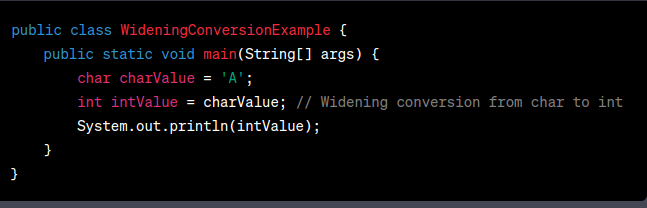
From float to double.

Example 1: From int to long



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



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.