1. **Statically Typed Language**

   In statically typed programming languages, type checking occurs at compile time. Java is a statically typed programming language.

   Dynamically Typed Language

   In dynamically typed programming languages, type checking take place at run time or execution time. Java is also a dynamically typed programming language.

   Strongly Typed Language

   Strongly typed languages, that variable have a well-defined type and that there are strict rules about the type of variable. Java is a strongly typed programming language because it demands the declaration of every variable with a data type.

   Loosely Typed Language

   Loosely typed languages don't care about the type of variables. We do not have to specify the variable type in advance. Java is not a loosely typed language.

2. **Case Sensitive**

   If a programming language is case sensitive, it means that it distinguishes between uppercase and lowercase letters. Case sensitive programming languages include C, C#, C++, Java, Python, Ruby and Swift.

   Case Insensitive

   If a programming language is case insensitive, it has ability to ignore the difference between upper- and lower-case version of a letter. Some examples of these programming languages include Ada, Fortran, SQL, and Pascal.

3. **Identity Conversion**

   In Java, identity conversion is a type conversion that is the simplest and safest type of conversion. It occurs when a value is assigned to a variable of the same type without any explicit casting or conversion. In other words, when the sources value's type matches exactly with the target variable's type, the Java compiler performs an identity conversion.

   Ex 1: Integer to int

   ```
   int number = 42;                    // Declaration of an int variable
   Integer integerObject = number; // Identity conversion from int to Integer
   ```

   In this example, we have an int variable named number with value 42. The wrapper class Integer wraps the primitive int type. When we assign the int variable number to an Integer object integerObject, an identity conversion takes place because the two types of matches. The java compiler automatically converts the int primitive to the corresponding Integer object.

Ex 2: String to String

```
String name = "John";       // Declaration of a String variable
String newName = name; // Identity conversion from String to String
```

In this example, we have a String variable named name with the value John. When we assign the String variable name to another String variable newname, an identity conversion occurs because both variables have the same type (String). The java compiler recognizes this and performs the assignment without any need for explicit casting or conversion.

4. **Primitive widening conversion**

Primitive widening conversion, also known as a automatic type conversion, is a process in Java where a value of a smaller data type is automatically converted to a value of a larger data type without any data loss. This type of conversion is performed implicitly by the Java compiler when certain conditions are met. The goal is to ensure that no data is lost during the conversion and that the value can be represented by the larger data type.

$$\begin{tikzcd}
 & \text{char} \arrow[d] & & & & \\
\text{byte} \arrow[r] & \text{short} \arrow[r] & \text{int} \arrow[r] & \text{long} \arrow[r] & \text{float} \arrow[r] & \text{double}
\end{tikzcd}$$

The arrows indicate the allowed widening conversions. For example, a byte can be automatically converted to a short, an int, a long, a float, or a double, but it cannot be converted to a char or a smaller data type.

Ex: byte to int

```
byte b = 50;          // Declare a byte variable and assign a value
int i = b;            // Widening conversion from byte to int
```

The value 50 is stored in the byte variable b. Since int can represent a larger range of values than byte, the Java compiler performs a widening conversion automatically when we assign b to the int variable i.

5. **Run time constant.**

Run time constants are constants whose values are determined only during the execution of the program, ex, at run time. They cannot be determined at compile time, so their values are evaluated or provided when the program is running.

Ex:
final int y = getRandomNumber(); // 'y' is a run-time constant, its value is determined at runtime

In this example, y is declared as a final variable, but its value is assigned by calling the method getRandomNumber() at runtime. The compiler does not know the value of y during compilation, so it cannot replace it directly in the bytecode.

Compile time constant.

Compile time constants are constants whose value can be determined by the compiler during the compilation phase of the program. These constants are resolved at compile time and replaced with their actual values in the bytecode. They are usually literals or expressions composed of literals, and their values are known and fixed before the program is executed.

Ex:
final int x = 10;   // 'x' is a compile-time constant, its value is known during compilation
final String message = "Hello, World!";   // 'message' is a compile-time constant

In the example, x and message are compile time constants. The compiler knows their values during the compilation process and directly substitutes them in the bytecode. These constants are often declared using the final keyword to indicate that their values cannot be changed during runtime.

6. **Implicit (Automatic) Narrowing Primitive Conversions**

This occur automatically when a value of a larger data type is assigned to a variable of a smaller data type. In this conversion, the Java compiler automatically truncates the value to fit into the target data type. The conversion is considered implicit because it happens without the need of any explicit code. However, there is a potential risk of data loss in this process because the larger data type may not be fully representable within the smaller data type.

Explicit Narrowing Conversions (Casting)

This is also known as casting, are used when a developer wants to force a conversion from a larger data type to a smaller data type explicitly. This process involves using parentheses along with the target data type to inform the compiler about the intended conversion. Casting allows the developer to override the default behavior of the Java compiler, but it comes with the risk of potential data loss if the value exceeds the representable range of the target data type.

Conditions for Implicit (Automatic) Narrowing Primitive Conversions

- The target data type (small type) must have a smaller range of representable values than the source data type (larger type).
- The value being assigned must be within the range of the target data type. If the value exceeds the range, a compilation error will occur, and an explicit narrowing conversion (casting) would be required.
- No explicit casting is used in the assignment statement. The Java compiler automatically performs the conversion if the above conditions are satisfied.

7. Assigning a long data type, which is 64 bits in Java, to a float data type, which is only 32 bits, involves a type conversion known as a widening primitive conversion. Despite the discrepancy in the number of bits, it is allowed because float and long have different representations and purposes.

   In Java, primitive data types are categorized into two groups: integral types (e.g., byte, short, int, long, char) and floating-point types (e.g., float, double). These two groups use different formats for storing data. The long data type is an integral type and uses 64 bits to represent a wide range of integer values from -2^63 to 2^63-1. It is typically used for large integer values that need a wider range than the int data type.

   On other hand, the float data type is a floating-point type and uses 32 bits to represent a wide range of real numbers. It follows the IEEE 754 standard and provides a larger range of values, including both integers and fractional values. However, the precision of float is limited compared to the double data type.

   When a long value is assigned to a float variable, a widening primitive conversion takes place. This conversion is allowed in Java because the float type has a greater range than the long type, though with a potential loss of precision.

   During the conversion, the long value is first converted to an intermediate value that represent the exact integer. Then, this intermediate value is converted to the float format. The intermediate value must be within the representable range of the float data type; otherwise, there may be data loss due to the limited precision of float.

8. Java’s choice of int and double as the default data types for integer and floating-point literals strikes a balance between common use cases, performance considerations, and backward compatibility, making the language both convenient and efficient for a wide range of applications.

   It’s important to note that while int and double are the default data types for integer and floating-point literals, Java provides other data types and supports explicit casting for cases where more specialized or memory-efficient data types are necessary.

9. Implicit narrowing primitive conversion in Java takes place only among byte, char, int, and short because these data types are small enough to allow for automatic type promotion without significant loss of precision or range. The language designers made this decision to strike a balance between convenience and avoiding potential pitfalls related to precision loss.

On the other hand, implicit narrowing conversion is not allowed for larger data types like long and double. The range and precision difference between these types and the smaller ones (byte, char, short, int) is more significant, leading to potential data loss and unintended consequences if narrowing conversion were allowed. Java prioritizes type safety and avoids allowing implicit narrowing conversion that could result in loss of information or introduce hard-to-find bugs.

To perform a narrowing conversion for other data types, developers need to use explicit casting to indicate that they are aware of the potential data loss and that it is a deliberate decision. By making explicit casting necessary for such conversions, Java promotes safer and more predictable code.

10. Widening Primitive Conversion

    - Occurs when a smaller data type is automatically promoted to a larger data type.
    - No data occurs, and the value can be accurately represented in the larger type.
    - Always allowed in Java without explicit casting.

    Narrowing Primitive Conversion

    - Occurs when a larger data type is automatically converted to a smaller data type.
    - May lead to data loss or truncation, so explicit casting is required to ensure programmer awareness.
    - Requires explicit casting in Java.

The conversion from short to char is not classified as either widening or narrowing because they are two distinct data types of the same size (16 bits) with different interpretations.

short: Represents a 16-bits signed integer (both positive and negative values).
char: Represents a 16-bits Unicode character (only positive values for character representations).

As they have different purposes and interpretations, converting short to char involves potential data loss (for negative short values) and thus requires explicit casting.