1.
Dynamically-typed languages perform type checking at runtime, while statically typed languages perform type checking at compile time.

This means that scripts written in dynamically-typed languages (like Groovy) can compile even if they contain errors that will prevent the script from running properly (if at all).
If a script written in a statically-typed language (such as Java) contains errors, it will fail to compile until the errors have been fixed. Statically-typed languages require you to declare the data types of your variables before you use them, while dynamically-typed languages do not. Consider the two following code examples:

```
//Java example
int num;
num =8;

//Groovy example
num=8
```

Both above examples do the same thing: create a variable called num and assign it the value 8. The difference lies in the first line of the Java example, int num;, which defines num's data type as int. Java is statically-typed, so it expects its variables to be declared before they can be assigned values. Groovy is dynamically-typed and determines its variables' data types based on their values, so this line is not required.

Dynamically-typed languages are more flexible and can save you time and space when writing scripts. However, this can lead to issues at runtime. For example:

```
// Groovy example
number = 8
numbr = (number+12) / 2  //note typo error
```

The code above should create the variable number with a value of 5, then change its value to 10 by adding 12 to it and dividing it by 2. However, number is misspelled at the beginning of the second line. Because Groovy does not require you to declare your variables, it creates a new variable called numbr and assigns it the value number should have. This code will compile just fine, but may produce an error later on when the script tries to do something with number assuming its value is 10.

Weakly-typed languages make conversions between unrelated types *implicitly*;
whereas, strongly-typed languages do not allow implicit conversions between unrelated types.

Python is a strongly-typed language:

```
var = 21; #type assigned as int at runtime.
var = var + "dot"; #type-error, string and int cannot be concatenated. print(var);
```

Output of above code:

```
Traceback (most recent call last):
  File "/home/nuwan/hello.py", line 2, in <module>
    var = var + "dot";    #type-error, string and int cannot be concatenated.
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

JavaScript is a weakly-typed language:

```
value = 21; v
alue = value + "dot";
console.log(value); /* This code will run without any error. As Javascript is a weakly-typed language, it allows implicit conversion between unrelated types. */
```

Output of above code:

```
21dot
```

According to above characteristics of given categories, Java is statically and strongly typed language.

2.

Case Sensitive: A programming language is considered case sensitive if it distinguishes between uppercase and lowercase letters when identifying or referencing elements in the code. This means that "identifier," "Identifier," and "IDENTIFIER" are treated as three distinct entities. For example:

```
# Case-sensitive Python code
variable = 42
Variable = "Hello"
print(variable)  # Output: 42
print(Variable)  # Output: Hello
```

Case Insensitive: A programming language is considered case insensitive if it does not differentiate between uppercase and lowercase letters when identifying or referencing

elements in the code. This means that "identifier," "Identifier," and "IDENTIFIER" are treated as the same entity. For example:

```
' Case-insensitive VBScript code
Dim myVariable
myvariable = 42
MYVARIABLE = "Hello"
MsgBox myVariable  ' Output: Hello
MsgBox MYVARIABLE  ' Output: Hello
```

Case Sensitive-Insensitive (Mixed): In some programming languages, the case sensitivity can be mixed or configurable. It means that some parts of the language might be case-sensitive, while others are case-insensitive. For instance, the language may have case-sensitive variable names but case-insensitive keywords. An example could be:

Java is a case-sensitive programming language. It means that Java differentiates between uppercase and lowercase letters when it comes to identifiers such as variable names, method names, class names, etc. For example:

```
// Case-sensitive Java code
int myVariable = 42;
int MyVariable = 100;


System.out.println(myVariable);  // Output: 42
System.out.println(MyVariable);  // Output: 100
```

In Java, the variables `myVariable` and `MyVariable` are considered as two different variables. If you attempt to access a variable with the wrong case, you'll get an error or unintended results.

3.
A conversion from a type to that same type is permitted for any type. In other words, identity conversion happens when you assign a value to a variable of the same data type as the value itself. In this scenario, no actual conversion is required because the target variable's data type is already the same as the data type of the value being assigned.

Example 1: Identity Conversion with int

```
public class IdentityConversionExample {
    public static void main(String[] args) {
```

```
        int x = 42;
        int y = x;  // Identity conversion, x and y are of the same type (int)

        System.out.println("x: " + x);  // Output: x: 42
        System.out.println("y: " + y);  // Output: y: 42
    }
}
```

In this example, we have an `int` variable `x` initialized with the value `42`. When we assign `x` to another `int` variable `y`, it is an identity conversion because `x` and `y` are of the same data type. The value `42` is copied directly from `x` to `y`.

Example 2: Identity Conversion with double

```
public class IdentityConversionExample {
    public static void main(String[] args) {
        double a = 3.14;
        double b = a;  // Identity conversion, a and b are of the same type (double)

        System.out.println("a: " + a);  // Output: a: 3.14
        System.out.println("b: " + b);  // Output: b: 3.14
    }
}
```

In this example, we have a `double` variable `a` initialized with the value `3.14`. When we assign `a` to another `double` variable `b`, it is an identity conversion because `a` and `b` are of the same data type. The value `3.14` is copied directly from `a` to `b`.

In both above examples, the target variable's data type is the same as the source value's data type, making the conversion an identity conversion. No data loss or change in representation occurs in the process.

4.
Primitive widening conversion in Java is a type conversion that allows the automatic and implicit promotion of smaller data types to larger data types.

19 specific conversions on primitive types are called the widening primitive conversions:

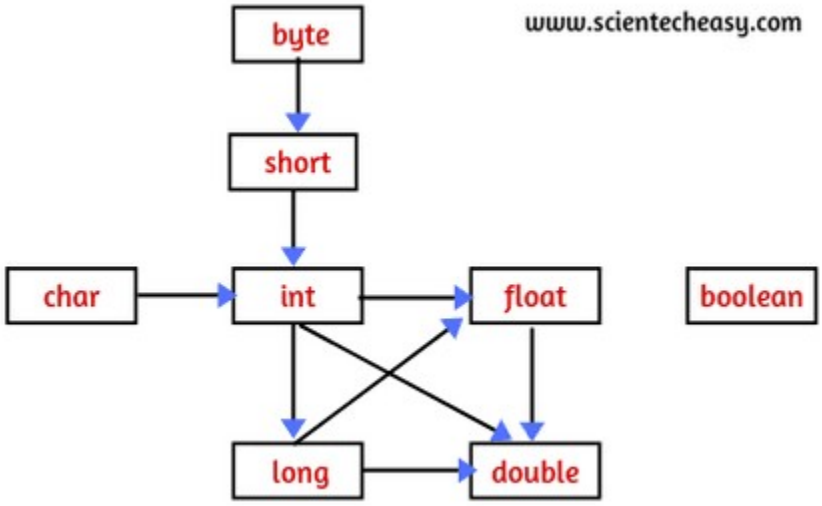
- byte to short, int, long, float, or double
- short to int, long, float, or double

- char to int, long, float, or double
- int to long, float, or double
- long to float or double
- float to double

A widening primitive conversion does not lose information about the overall magnitude of a numeric value in the following cases, where the numeric value is preserved exactly:

- from an integral type to another integral type
- from byte, short, or char to a floating point type
- from int to double
- from float to double in a strictfp expression

The following diagram illustrates the hierarchy of data types in Java, where the data types at the top are larger and can hold a wider range of values than those at the bottom:



Example 1: Widening Conversion from `byte` to `int`

```
public class WideningConversionExample {

    public static void main(String[] args) {
        byte smallNumber = 10;
```

```
        int largerNumber = smallNumber;  // Widening conversion, byte to int

        System.out.println("smallNumber: " + smallNumber);      // Output: smallNumber:
10
        System.out.println("largerNumber: " + largerNumber);  // Output: largerNumber:
10
    }
}
```

In this example, we have a variable `smallNumber` of type `byte` initialized with the value `10`. When we assign `smallNumber` to the variable `largerNumber` of type `int`, a widening conversion takes place. The `byte` value `10` is automatically and implicitly promoted to an `int`, which can accommodate a wider range of values than `byte`.

Example 2: Widening Conversion from `float` to `double`

```
public class WideningConversionExample {

    public static void main(String[] args) {
        float smallerFloat = 3.14f;
        double largerDouble = smallerFloat;  // Widening conversion, float to double

         System.out.println("smallerFloat: " + smallerFloat);         // Output: smallerFloat:
3.14
        System.out.println("largerDouble: " + largerDouble);        // Output: largerDouble:
3.14
    }
}
```

In this example, we have a variable `smallerFloat` of type `float` initialized with the value `3.14f`. When we assign `smallerFloat` to the variable `largerDouble` of type `double`, a widening conversion occurs. The `float` value `3.14` is automatically and implicitly promoted to a `double`, which can hold a wider range of values and more precision than `float`.

A widening primitive conversion from int to float, or from long to float, or from long to double, may result in loss of precision - that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value

will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode.

A widening conversion of a signed integer value to an integral type T simply sign-extends the two's-complement representation of the integer value to fill the wider format.

A widening conversion of a char to an integral type T zero-extends the representation of the char value to fill the wider format.

Despite the fact that loss of precision may occur, a widening primitive conversion never results in a run-time exception.

5.

A compile-time constant is computed at the time the code is compiled, while a run-time constant can only be computed while the application is running. A compile-time constant will have the same value each time an application runs, while a run-time constant may change each time.

For example, the following code declares a compile-time constant:

final int MAX_VALUE = 100;

The value of MAX_VALUE is known at compile time, so the compiler can replace the constant name with its value in the code. This makes the code more efficient, because the compiler can do some of the work ahead of time.
A **runtime constant** is a value that is not known at compile time. This means that the value of the constant is not available to the compiler when it is compiling the code. Runtime constants are often used in expressions and declarations, but the value of the constant is not known until the program is running.

For example, the following code declares a runtime constant:

int currentYear = 2023;

The value of currentYear is not known at compile time, so the compiler cannot replace the constant name with its value in the code. Instead, the value of currentYear is determined when the program is running.

The main difference between compile-time constants and runtime constants is that compile-time constants are known at compile time, while runtime constants are not known at compile time. This difference affects how the compiler can optimize the code.

6. Implicit narrowing conversions occur automatically when a value of a larger data type is assigned to a variable of a narrower data type. In this process, the compiler automatically reduces the range and precision of the value to fit into the target data type. Since narrowing conversions may result in data loss or loss of precision, the Java compiler enforces rules to ensure that the conversion is safe.

Example of implicit narrowing conversion:

```
public class ImplicitConversionExample {
    public static void main(String[] args) {
        int largerValue = 1000;
        byte smallerValue = largerValue; // Implicit narrowing conversion from int to byte
        System.out.println("largerValue: " + largerValue);    // Output: largerValue: 1000
        System.out.println("smallerValue: " + smallerValue); // Output: smallerValue: -24 (due to data loss)
    }
}
```

In this example, we have an `int` variable `largerValue` initialized with the value `1000`. When we assign `largerValue` to a `byte` variable `smallerValue`, the implicit narrowing conversion takes place. Since `byte` can only hold values from -128 to 127, the value `1000` is reduced modulo 256 (the range of `byte`) and stored in `smallerValue`. Hence, the output of `smallerValue` is `-24`.

Conditions for an implicit narrowing primitive conversion to occur:

- The target data type (the narrower type) must have a smaller range and precision than the source data type (the larger type).
- The value being assigned must be within the range of the target data type. If the value exceeds the range of the target type, it will be reduced modulo the number of distinct values representable by the target type.

Explicit Narrowing Conversions (Casting): Explicit narrowing conversions, also known as casting, occur when a programmer explicitly instructs the compiler to convert a value of

a larger data type to a variable of a narrower data type. This is achieved by placing the target type in parentheses before the value to be converted.

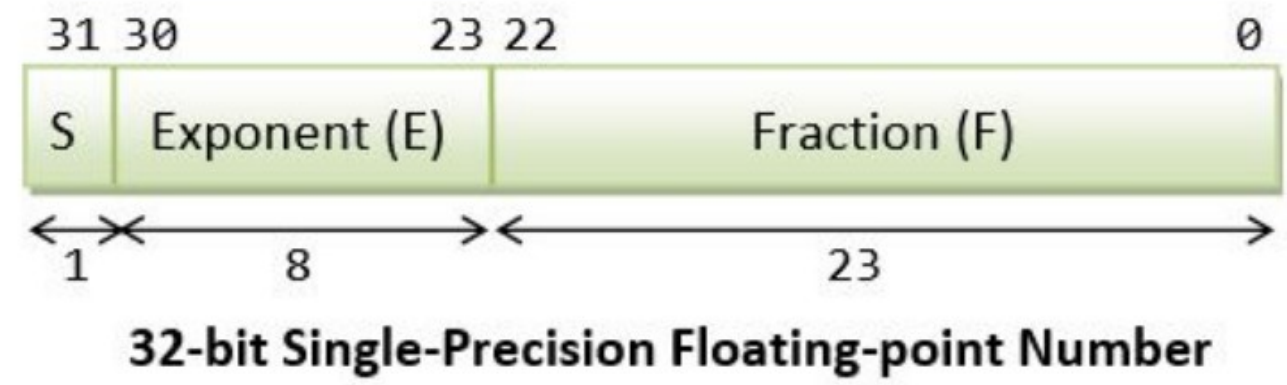
Example of explicit narrowing conversion:

```
public class ExplicitConversionExample {
    public static void main(String[] args) {
        double largerValue = 3.14;
        int smallerValue = (int) largerValue; // Explicit narrowing conversion from double to int
        System.out.println("largerValue: " + largerValue);    // Output: largerValue: 3.14
        System.out.println("smallerValue: " + smallerValue); // Output: smallerValue: 3 (decimal part truncated)
    }
}
```

In this example, we have a `double` variable `largerValue` initialized with the value `3.14`. When we explicitly cast `largerValue` to an `int` data type, the decimal part is truncated, resulting in `smallerValue` being `3`.

7.

In Java, the `long` data type is a 64-bit signed two's complement integer, which can hold values in the range of approximately (-2∧64) to (2∧63 – 1). On the other hand, the `float` data type is a 32-bit single-precision floating-point number according to the IEEE 754 standard. It can represent a broader range of values than `long` but with less precision because bit structure of ‘float’ type is different than ‘long’ type as illustrated below.



The float data type is 32 bits in size and consists of three parts: sign bit, exponent, and mantissa (also called fraction or significand).

Sign Bit (1 bit): The leftmost bit is the sign bit, which represents the sign of the floating-point number. If the sign bit is 0, the number is positive, and if it is 1, the number is negative.

Exponent (8 bits): The next 8 bits represent the exponent of the floating-point number. The exponent is used to scale the value represented by the mantissa. It is stored as an unsigned integer biased by 127. That is, the actual exponent is obtained by subtracting 127 from the stored exponent value. The range of the exponent is from 0 to 255 (2^8 - 1), but two special values are reserved for representing special numbers (NaN and Infinity), making the effective range of the exponent from -126 to 127.

Mantissa (23 bits): The remaining 23 bits represent the mantissa (also called fraction or significand) of the floating-point number. It stores the significant digits of the number in binary fractional form. The implicit leading bit (not stored explicitly) is always 1 for normalized numbers, except for denormalized numbers and special cases like zero and NaN.

Here's an example to illustrate the concept:

```
public class LongToFloatConversionExample {
    public static void main(String[] args) {
        long longValue = 12345678912345L;
        float floatValue = longValue; // Implicit widening conversion from long to float
```

```
        System.out.println("longValue: " + longValue);          // Output: longValue: 12345678912345
        System.out.println("floatValue: " + floatValue);        // Output: floatValue: 1.23456793E13
    }
}
```

In this example, we have a `long` variable `longValue` initialized with the value `12345678912345L`. When we assign `longValue` to a `float` variable `floatValue`, a widening conversion takes place. While the exact integer value `12345678912345` can be represented accurately in a `long`, when assigned to a `float`, it is represented as an approximation of the real value using floating-point notation, which is `1.23456793E13`.

The conversion may lead to some loss of precision as float has a limited number of bits to represent both the integral part and the fractional part of a number. Therefore, when using float, one should be cautious about potential rounding errors when dealing with large integer values or requiring high precision. For more precise numeric operations, the double data type (64-bit) is typically used.

8.

The decision to set `int` as the default data type for integer literals and `double` as the default data type for floating-point literals in Java was made to strike a balance between performance and precision while minimizing the risk of data loss and unexpected results.

Default Data Type for Integer Literals (`int`):

Java was designed to be platform-independent and efficient. Using `int` as the default data type for integer literals helps optimize performance because most processors have native support for integer arithmetic with 32-bit integers. This choice ensures that basic integer arithmetic operations are efficient on most hardware architectures.

Setting `int` as the default also helps maintain consistency with the most commonly used data type for integers and allows developers to write code without explicit type annotations for most integer operations. As a result, most integer literals are automatically treated as `int` unless they exceed the range of `int`, in which case an explicit type annotation (such as `long` or `short`) is required.

For example, in Java, `int number = 42;` automatically assigns the integer literal `42` to an `int` variable, making it a convenient and efficient way to work with most integer values.

Default Data Type for Floating-Point Literals (`double`):

Java's decision to use `double` as the default data type for floating-point literals is driven by the need to provide higher precision for floating-point calculations and to minimize the risk of data loss due to rounding errors.

The `double` data type in Java adheres to the IEEE 754 standard for 64-bit floating-point representation, which provides a higher level of precision than the `float` data type (32-bit floating-point representation). While `float` is useful in situations where memory usage is critical or when a larger range of values is required, `double` is the default choice to strike a balance between performance and precision for most floating-point calculations.

Setting `double` as the default data type ensures that floating-point literals, such as `3.14`, are automatically treated with double precision. If you want to use a `float` literal, you must explicitly annotate it with the `f` suffix, like `float myValue = 3.14f;`.

By having `double` as the default for floating-point literals, Java encourages developers to use higher precision when working with floating-point numbers, reducing the risk of precision-related bugs and ensuring accurate calculations in most scenarios.

9.

Implicit narrowing primitive conversion only takes place among byte, char, int, and short because these are the only primitive data types that are compatible with each other.

The byte data type is the smallest primitive data type in Java. It is 8 bits in size and can store values from -128 to 127. The char data type is 16 bits in size and can store any Unicode character. The int data type is 32 bits in size and can store values from -2147483648 to 2147483647. The short data type is 16 bits in size and can store values from -32768 to 32767.

Any value that can be stored in a byte can also be stored in a char, int, or short. However, not all values that can be stored in a char, int, or short can be stored in a

byte. For example, the value 128 cannot be stored in a byte, but it can be stored in a char, int, or short.

This is why implicit narrowing primitive conversion only takes place among byte, char, int, and short. If a value is implicitly converted to a smaller data type, then no information is lost. When converting from a larger data type to a smaller one, it is essential to ensure that the value can be safely represented within the narrower data type's range.

It is important to note that explicit narrowing primitive conversion can be used to convert values between any of the primitive data types. However, explicit narrowing primitive conversion can result in loss of information if the value being converted is not representable in the smaller data type. If the compiler allowed implicit narrowing conversion to other data types, such as `long` to `int`, or `double` to `float`, there would be a higher risk of data loss and inaccuracies due to the significantly different range and precision of these data types. Therefore, Java restricts implicit narrowing conversions to the subset of data types where it can be performed safely and predictably. For conversions between other data types, explicit narrowing conversion (casting) must be used, allowing the developer to take responsibility for any potential data loss and accuracy issues.

10.

The following conversion combines both widening and narrowing primitive conversions:

- byte to char

First, the byte is converted to an int via widening primitive conversion, and then the resulting int is converted to a char by narrowing primitive conversion

byte (8 bits) → int (32 bits) → char (16 bits)

Now, let's address the specific case of converting from `short` to `char`.

Conversion from `short` to `char` is not classified as either a widening or a narrowing primitive conversion because it is a special case. Both `short` and `char` have the same size (16 bits) in Java, and they represent different types of values:

- `short` represents signed integers with a range of -32,768 to 32,767.

- `char` represents unsigned 16-bit Unicode characters with a range of 0 to 65,535.

Since `short` and `char` have the same size and there is no data loss involved in the conversion, it is considered a special identity conversion. It is not widening because the two types have the same size, and it is not narrowing because no data loss or truncation occurs. As a result, the conversion from `short` to `char` is classified as neither widening nor narrowing, but rather as a special case of identity conversion.