PROGRAMMING FUNDAMENTALS ASSIGNMENT – 2

1. Elucidate the following concepts: 'Statically Typed Language', 'Dynamically Typed Language', 'Strongly Typed Language', and 'Loosely Typed Language'? Also, into which of these categories would Java fall?"

- Statically Typed Language – Computer Language which Type checking during the compiling stage.
  Ex : java,C,C++,Rust,Go,Scala

- Dynamically Typed Language – Computer Language which Type checking during run time.
  Ex : Perl, Ruby, Python, PHP, JavaScript, Erlang

- Strongly Typed Language – The language that consider the data types.
  Ex : java,C++,swift

- Loose Typed Language – The language that not consider the data types.
  Ex : PHP,Perl

- Java falls to Dynamically Typed Language and Strongly Types Language.

2. "Could you clarify the meanings of 'Case Sensitive', 'Case Insensitive', and 'Case Sensitive-Insensitive' as they relate to programming languages with some examples? Furthermore, how would you classify Java in relation to these terms?"

- Case-sensitive Language - A case-sensitive language is a programming language in which the distinction between uppercase and lowercase letters is significant.
  Ex : java,C,C++,C#,Phython,Ruby,javaScript

- Case-Insensitive Language - A case-insensitive language is a programming language in which the distinction between uppercase and lowercase letters is not significant .
  Ex : VB.NET

- Case sensitive - Insensitive Language - Some programming languages provide options to treat identifiers as either case sensitive or case insensitive based on a specific setting or compiler flag. In such languages, you can choose whether the distinction between uppercase and lowercase letters matters (case sensitive) or doesn't matter (case insensitive) when writing code.

- Java is a Case-sensitive Language

3. Explain the concept of Identity Conversion in Java? Please provide two examples to substantiate your explanation.

In Java, identity conversion is the simplest form of type conversion. It occurs when a value of a certain type is assigned to a variable of the same type (or a compatible type).

An identity conversion is a safe and straightforward operation because it involves converting a value to the exact same type or to a type that is considered compatible without any loss of precision or risk of data loss.

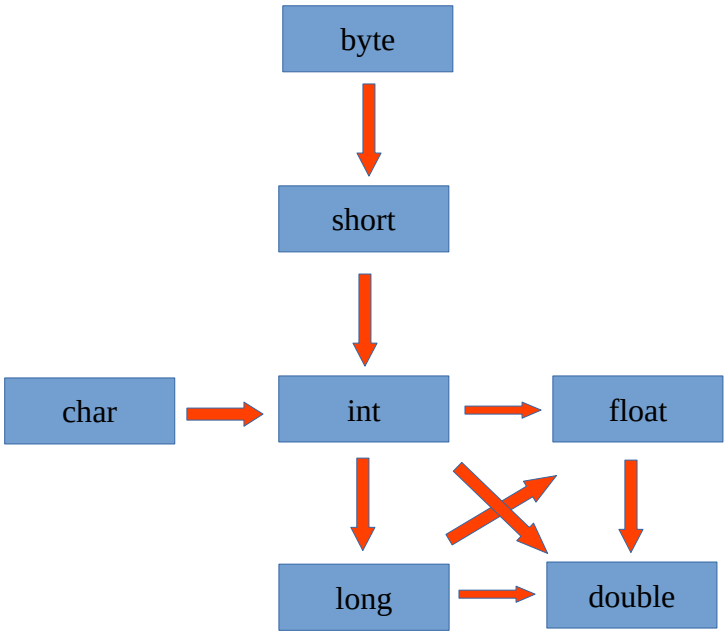
```java
String name1 = "Sahan";
String name2 = name1;
//In this example te variable "name1" is assigned the string of sahan
//The variable "name2" is assigned the vslue of "name1"
//Since both variables are type of string.It is an identity conversion.


int num1 = 35;
int num2 = num1;
//In this example te variable "num1" is assigned the integer of 35
//The variable "num2" is assigned the vslue of "num1"
//Since both variables are type of integers.It is an identity conversion.
```

4. Explain the concept of Primitive Widening Conversion in Java with examples and diagrams.

The type conversion in Java where a value of a smaller data type is assigned to a variable of a larger data type is called primitive widening conversion.

Primitive widening conversions in Java involve automatically converting a smaller data type to a larger data type without any loss of information or need for explicit casting.

This type conversion is happening towards the direction of arrows in the below diagram.



5. Explain the the difference between run-time constant and Compile-time constant in java with examples.

Compile-time constant:

A compile-time constant is the constants that are already known at compile-time, and their values are determined before the program runs.

```
final int MATH_MARKS = 85;
```

In the given example, MATH_MARKS represents a compile-time constant. The value 80 is already known to the compiler during the compilation phase. Consequently, the Java compiler will replace all instances of MATH_MARKS with the value 80 in the compiled code. This "baking" of the constant value into the code eliminates any runtime overhead for accessing it, as the value is directly substituted during compilation.

Run-time constant:

A run-time constant is a value that is known only at runtime, These constants are determined or calculated during program execution, and they cannot be evaluated by the compiler during the compilation phase.

```
final int RANDOM_NUMBER = new Random().nextInt(5);
```

In this particular instance, RANDOM_NUMBER serves as a run-time constant. Its value is determined at runtime when the program executes the new Random().nextInt(5) statement. Due to the fact that the value is only known during runtime, the Java compiler cannot replace it with a specific value during compilation. As a result, RANDOM_NUMBER remains dynamic throughout program execution and cannot be statically evaluated during the compilation phase.

6. Explain the difference between Implicit (Automatic) Narrowing Primitive Conversions and Explicit Narrowing Conversions (Casting) and what conditions must be met for an implicit narrowing primitive conversion to occur?

Implicit (Automatic) Narrowing Primitive Conversions

Implicit narrowing primitive conversions, take place when a value of a larger data type is assigned to a variable of a smaller data type. During this conversion, the Java compiler automatically truncates or discards some information to fit the larger value into the smaller data type.

```
short myByte = 10;
int myShort = 20;

//In this example the byte and short data ranges are smaller than
//the int data type. But, java allow us to to those conversions
//without any casting.
```

Conditions for Implicit Narrowing Primitive Conversion:

For an implicit narrowing primitive conversion to occur:

1. The value being converted should be a constant.
2. The constant value should be within the valid range of the target data type.
3. The constant value should be known at compile-time, not determined at run-time.

Explicit Narrowing Conversions (Casting)

Explicit narrowing conversions, also known as casting, occur when you manually convert a value from a larger data type to a smaller data type. You do this by using a cast operator (`(type)`) to inform the compiler that you are aware of the potential loss of data, and you still want to perform the conversion.

7. How can a long data type, which is 64 bits in Java, be assigned into a float data type that's only 32 bits? Could you explain this seeming discrepancy?"

The key point to understand here is that `float` uses a different representation format called IEEE 754 floating-point format. In this format, the 32 bits are divided into three parts:

1. The sign bit: 1 bit to represent the sign of the value (positive or negative).
2. The exponent: A group of bits that represents the magnitude of the value.
3. The fraction (mantissa): A group of bits that represents the fractional part of the value.

By using the exponent and mantissa, `float` can represent a wide range of values, including very large and very small numbers, but with reduced precision compared to `long`.

8. Why are int and double set as the default data types for integer literals and floating point literals respectively in Java? Could you elucidate the rationale behind this design decision?

a. Java was designed to be a successor to C and C++, which both use `int` as the default data type for integer literals and `double` as the default data type for floating-point literals.

b. When Java was introduced, the design aimed to be compatible with existing C/C++ code bases. Using the same defaults for integer and floating-point literals helped maintain compatibility with existing code, as well as facilitate the migration of code from C/C++ to Java.

c. The int data type strikes a suitable balance between the range of representable integers and memory usage in Java. Occupying 32 bits of memory, it can effectively represent integer values within a range spanning approximately from -2 billion to +2 billion, making it suitable for a wide array of practical applications.

d. `double` was chosen for floating-point literals because it provides higher precision than `float` (64 bits vs. 32 bits) without a significant performance penalty. While `float` can represent a wider range of values, it has limited precision, which may not be suitable for applications that require high accuracy in calculations. By defaulting to `double`, Java ensures that floating-point calculations are generally more accurate.

e. The default data type selection in Java is designed to minimize the necessity for explicit type annotations in most situations. By defaulting to int for integers and double for floating-point numbers, developers can often use these data types without explicitly specifying them, resulting in less verbosity and improved code readability. These default choices cover a broad range of use

cases, making it convenient for programmers to work with numeric values without the need for constant type declarations.

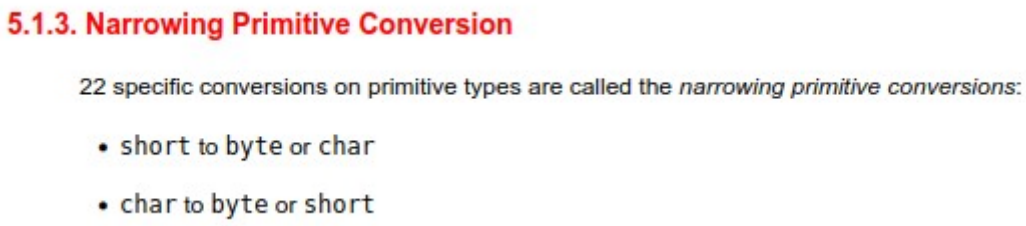
9. Why does implicit narrowing primitive conversion only take place among byte , char , int , and short ?

Implicit narrowing primitive conversion in Java only takes place among `byte`, `char`, `int`, and `short` data types because these data types have a fixed size and their ranges can be represented using a smaller number of bits. Implicit narrowing conversion involves converting a larger data type to a smaller data type, which may result in loss of data if the value is too large to fit into the smaller data type.

Implicit narrowing conversions from `float` to `byte`, `char`, `int`, or `short`, and from `double` to `byte`, `char`, `int`, `short`, or `float`, are not allowed because `float` and `double` have a larger range and precision than the mentioned data types. Implicitly converting from `float` or `double` to a smaller data type could result in a loss of information or precision.

10. Explain “Widening and Narrowing Primitive Conversion”. Why isn't the conversion from short to char classified as Widening and Narrowing Primitive Conversion?

"Widening and Narrowing Primitive Conversion" in Java pertains to the transformation of values between distinct primitive data types. When the target data type has a greater size or encompasses a broader range than the source data type, the conversion is considered "widening." Conversely, when the target data type has a smaller size or a narrower range compared to the source data type, the conversion is categorized as "narrowing."



5.1.3. Narrowing Primitive Conversion

22 specific conversions on primitive types are called the *narrowing primitive conversions*:

- short to byte or char
- char to byte or short

Here we can see both `short` to `char` and `char` to `short` are considered narrowing primitive conversions is because the `char` data type is 16-bit and unsigned, while `short` is 16-bit and signed.

When converting from `short` to `char`, the compiler treats the `short` value as an unsigned 16-bit integer and directly maps it to the `char` type, which can represent a larger range of positive values compared to a `short`.

Conversely, when converting from `char` to `short`, the compiler treats the `char` value as a 16-bit unsigned integer and then performs a signed conversion to fit it into a `short`, which can represent a smaller range of values (both positive and negative) compared to `char`.

Therefore, both `short` to `char` and `char` to `short` are considered narrowing primitive conversions due to the potential loss of information when converting between these two data types.