Assignment 2 H.A.Chamal A. Hettiarachchi

Q1 ----------

1. Statically Typed Language: A statically typed language is one in which variable types are explicitly declared at compile time and checked before the program is executed. This means that the data types of variables must be known and defined at the time of writing the code. Once a variable is assigned a data type, it cannot be changed during runtime. The compiler enforces type checking to ensure that only compatible operations can be performed on variables. Examples of statically typed languages include Java, C, C++, and Swift.

2. Dynamically Typed Language: In contrast, a dynamically typed language is one where variable types are determined at runtime, not at compile time. The type of a variable can change during the execution of the program, and the type checking is performed during runtime. This flexibility allows for more concise code but may lead to potential type-related errors during runtime. Examples of dynamically typed languages include Python, JavaScript, Ruby, and PHP.

3. Strongly Typed Language: A strongly typed language enforces strict type rules and does not allow implicit type conversions unless explicitly specified by the programmer. This means that the language is less tolerant of mixing different data types, and conversions between types must be done explicitly. The goal of strong typing is to catch type-related errors early during the development process, ensuring more robust and reliable code.

4. Loosely Typed Language: On the other hand, a loosely typed language, also known as weakly typed language, is more permissive with type rules. It allows automatic type conversions or type coercion without explicit instructions from the programmer. This can sometimes lead to unexpected behavior and may require more careful handling of data types. Loosely typed languages tend to be more flexible but can be prone to subtle bugs related to type conversions.

Q2 ----------

1. Case Sensitive:A programming language is considered case sensitive if it differentiates between uppercase and lowercase characters in identifiers. This means that "abc" and "ABC" are treated as two distinctidentifiers. For example, if you define a variable named "myVariable" in a case-sensitive language, you cannot access its value using "myvariable" or "MYVARIABLE"; you must use "myVariable" exactly as it was defined.

Ex : C, C#, C++, Java, Python, Ruby, Swift

2. Case Insensitive:
A programming language is considered case insensitive if it treats uppercase and lowercase characters in identifiers as equivalent, effectively ignoring the distinction between them. This means that "abc" and "ABC" would be considered the same identifier. So, if you define a variable named "myVariable" in a case-insensitive language, you can access its value using "myVariable," "myvariable," "MYVARIABLE," or any other combination of cases.

Ex : MySQL, Pascal

3. Case Sensitive-Insensitive (Mixed):
In some programming languages, the treatment of identifiers may depend on the platform or the context. For example, file systems on certain operating systems might be case insensitive (e.g., Windows), while the programming language itself could be case sensitive. In such cases, the language is referred to as "Case Sensitive-Insensitive."

Example (JavaScript - Case Sensitive, File System - Case Insensitive):

** Java is a case-sensitive programming language. It strictly differentiates between uppercase and lowercase characters in identifiers. For example, "myVariable" and "myvariable" are considered as two distinct identifiers in Java.

Q3 ----------

* Identity conversion is a conversion from a type to that same type is permitted for any type.

Furthurmore,

1. When the type of the expression matches exactly with the target type.
2. When converting a reference type to a type that it directly extends or implements.

Example for Identity Conversion with Primitive Types

```
public class Demo1 {

public static void main(String[] args) {
   int num = 12;
   double numDouble = num; // Identity conversion from int to double

   System.out.println("num: " + num);              // Output: num: 12
   System.out.println("numDouble: " + numDouble);   // Output: numDouble: 12.0
   }
}
```

Example for Identity Conversion with Primitive Types

```
public class Demo2 {
public static void main(String[] args) {

   Object obj = "Hello, World!"; // String is a subclass of Object
   String str = (String) obj;    // Identity conversion from Object to String

   System.out.println("obj: " + obj);   // Output: obj: Hello, World!
   System.out.println("str: " + str);   // Output: str: Hello, World!
}
}
```

Q4 ----------

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

byte -----> short-----><u>int</u>-----><u>long</u>

int -----> float
int -----> double
long-----> float
long-----> double
char -----> int

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

Example for Widening Primitive Conversion

```
class Test {
   public static void main(String[] args) {
   int big = 1234567890;
```

```
    float approx = big;
    System.out.println(big - (int)approx);
    }
}
```

Q5 ----------

Compile-time Constant:

A compile-time constant is a constant whose value can be determined by the Java compiler at compile time, before the program execution starts. The value of such a constant is known during the compilation phase and remains fixed throughout the program's execution. Compile-time constants are usually literals, such as numeric literals or string literals, or expressions that only contain other compile-time constants and operations that can be evaluated at compile time.

Compile-time constants are determined at compile time and are usually literals or expressions that can be evaluated at compile time. They are typically declared as final and should have fixed, known values.

Example :

```
public class Demo3 {
   public static final int MAX_VALUE = 120;

   public static void main(String[] args) {
      int x = 50;
      int y = MAX_VALUE + x;

      System.out.println("y: " + y); // Output: y: 170
   }
}
```

Run-time Constant:

A Run-time constants are constants that have their values determined and assigned during program execution or runtime.

Their values may depend on user input or other runtime conditions.

Example :

```
import java.util.Scanner;

public class Demo4 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a value: ");
        int userInput = scanner.nextInt();

        final int MULTIPLIER = 5; // Run-time constant (value assigned during runtime)
        int result = userInput * MULTIPLIER;

        System.out.println("Result: " + result);
    }
}
```

Q6 ----------

Implicit (Automatic) Narrowing Primitive Conversions:

Implicit narrowing conversions occur automatically by the Java compiler when a value of a larger data type is assigned to a variable of a smaller data type.

These conversions are considered safe by the compiler as long as the value being assigned fits within the range of the target data type.

No explicit casting (type conversion) is needed for implicit narrowing conversions.

Example:

```
public class Demo5 {
    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 (data loss occurred which is the byte representation of 1000 due to the overflow.)
    }
}
```

Explicit Narrowing Conversions (Casting):

Explicit narrowing conversions, also known as casting, are performed when you manually convert a value of a larger data type to a variable of a smaller data type using a cast operator. Casting is a way to tell the compiler that you are aware of the potential data loss and explicitly want to perform the conversion.

Example:

```
public class Demo6 {
    public static void main(String[] args) {
        double largerValue = 123.45;
        int smallerValue = (int) largerValue; // Explicit narrowing conversion (casting) from double to int

        System.out.println("largerValue: " + largerValue);   // Output: largerValue: 123.45
        System.out.println("smallerValue: " + smallerValue); // Output: smallerValue: 123 (decimal part truncated)
    }
}
```

Conditions for Implicit Narrowing Primitive Conversion:

For an implicit narrowing conversion to occur safely without a compile-time error, the following conditions must be met:

* The value being assigned must fit within the range of the target data type. If the value exceeds the target data type's range, data loss may occur,

* The target data type should be smaller (in terms of the number of bits) than the source data type. This ensures that data loss, if any, will be limited to the least significant bits.

Q7 ----------

When it is assigned a long value to a float variable, Java will perform a widening primitive conversion followed by a narrowing primitive conversion.

This is because the float type can represent a broader range of values, including fractional numbers. The process involves two steps:

Widening Conversion (Long to Float):
The long value is widened to a float value to allow the representation of fractional parts.
This widening conversion occurs automatically without the need for explicit casting.

Narrowing Conversion (Float Precision Loss):
As float has limited precision compared to long, the value obtained after the widening conversion may lose some precision. The result is stored in the float variable.

Due to the potential loss of precision in the float type, it is generally not recommended to perform this type of conversion when high precision is required,
such as in financial calculations or when dealing with large integer values.

Example :

```
public class LongToFloatExample {
    public static void main(String[] args) {
        long myLongValue = 9223372036854775807L; // Maximum value for long (64 bits)
        float myFloatValue = myLongValue; // Implicit conversion from long to float

        System.out.println("myLongValue: " + myLongValue); // myLongValue:
9223372036854775807
        System.out.println("myFloatValue: " + myFloatValue); // myFloatValue: 9.223372E18
    }
}
```

Q8 ----------

Performance:
On most platforms, performing arithmetic operations using int is generally faster than using other integer data types such as short or byte. Similarly, using double for floating-point operations is typically more performant than using float. By making int and double the defaults, Java aimed to prioritize the most commonly used and efficient data types for arithmetic operations.

Widening and Narrowing Conversions:
Java supports automatic widening conversions for integer literals. This means that if you have a small integer literal like 10, it can be assigned to an int,
which is the default data type, without requiring an explicit cast. On the other hand, floating-point literals are by default treated as double because double
provides a larger range and precision than float. This helps to minimize data loss and improve overall accuracy when working with floating-point numbers.

Backward Compatibility:
When Java was first introduced, it was designed to be compatible with the C and C++ programming languages, which also use int for integer literals and double for floating-point literals. Keeping this default behavior allowed developers who were already familiar with C and C++ to transition more easily to Java.

Least Astonishment Principle:

The Java language strives to follow the principle of least astonishment, which suggests that the language's behavior should be intuitive and not surprise developers with unexpected outcomes. By using int and double as the defaults for integer and floating-point literals, respectively, the language aims to provide a more predictable experience for developers who might be accustomed to similar defaults in other programming languages.

Q9 ----------

In each of these conversions, the value of the smaller data type is safely and automatically converted to the larger data type. No data loss occurs if the value of the source type is within the range of the target type.

The reason why these specific types are involved in implicit narrowing conversion is mainly due to the following factors:

Size and Range:
The types byte, char, short, and int represent signed and unsigned integers with varying sizes and ranges. Here are their sizes and ranges:

byte: 8 bits, range -128 to 127

char: 16 bits, range 0 to 65,535

short: 16 bits, range -32,768 to 32,767

int: 32 bits, range -2^31 to 2^31-1

byte, char, and short have smaller sizes and ranges compared to int. When a value of a larger type (int) needs to be converted to a smaller type (byte, char, or short), there is a possibility of data loss because the larger range of int may not fit within the smaller range of the target type. However, Java allows implicit narrowing conversions only for byte, char, and short because they have a narrower range compared to int, reducing the likelihood of data loss in most common scenarios.

Safety and Error Prevention: Implicit narrowing conversions are generally more prone to potential loss of information than widening conversions.
By restricting implicit narrowing to a limited set of types, Java aims to improve code safety and prevent potential bugs caused by unintended data loss.

Q10 ----------

Regarding the conversion from short to char, it is not classified as a widening or narrowing primitive conversion. char represents their magnitude by 16bit and short represents their magnitude by 15 bits. if it is occurred a widening or narrowing primitive conversion, this ll'be occured as follows,
short (15bit magnitude) ------(Widening Primitive Conversion)------> it (31 bit magnitude) ------(Narrowing Primitive Conversion)------> char (16 bit magnitude)

According to above, most probably, they may result in data loss.

therefore short to char represents widening primitive conversion as follows

short (15bit magnitude) ------(Widening Primitive Conversion)------> char (16 bit magnitude)