Programming – Fundamentals: Assignment 2 **Isuri Gunaratne**

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?”

1. Statically Typed Language:
A statically typed language is a programming language in which variable types are explicitly declared during compile-time. The data type of each variable is known at compile-time, and the compiler enforces strict type-checking to ensure that only compatible operations are performed on variables. Once a variable is assigned a specific type, it cannot change its type during runtime. Examples of statically typed languages include Java, C, C++, and Swift.

2. Dynamically Typed Language:
A dynamically typed language is a programming language in which variable types are determined at runtime. Unlike statically typed languages, variables do not have predefined types at compile-time. Instead, the type is inferred when a value is assigned to the variable or when a variable is used in an expression. The type of a variable can change during runtime based on the value it holds. Dynamically typed languages provide more flexibility but might lead to runtime errors if incompatible operations are performed. Examples of dynamically typed languages include Python, JavaScript, and Ruby.

3. Strongly Typed Language:
A strongly typed language is a programming language in which strict type-checking is enforced and explicit type conversion (casting) is required for operations between different data types. The language ensures that only well-defined operations are performed on variables and prevents unintended type-related errors. Strongly typed languages do not implicitly convert data types, and the developer is responsible for handling type conversions explicitly. Both statically typed and dynamically typed languages can be strongly typed. Java is an example of a strongly typed language.

4. Loosely Typed Language:
A loosely typed language is a programming language that allows implicit type conversion (coercion) between different data types without requiring explicit casting. In loosely typed languages, variables can change their data type automatically as needed during runtime, based on the context of the operation being performed. This flexibility can make coding easier and reduce the need for explicit type declarations or conversions. Loosely typed languages can be dynamically typed or weakly typed, but they are not strongly typed. An example of a loosely typed language is JavaScript.

Java falls into the following category: Java is a statically typed, strongly typed language. In Java, all variable types must be declared explicitly, and the compiler enforces strict type-checking to ensure type safety.

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: In a case-sensitive language, the distinction between uppercase and lowercase letters is significant. This means that identifiers, such as variable names, function names, and keywords, must be written exactly as they are defined, with the same letter casing.

For example, if you declare a variable with the name "myVariable," referring to it as "myvariable" or "MyVariable" will be treated as a different identifier. Most programming languages, including Java, are case-sensitive.

Case Insensitive: In a case-insensitive language, the distinction between uppercase and lowercase letters is not considered significant. This means that identifiers can be written using any combination of uppercase and lowercase letters, and they will be treated as the same identifier.

For example, "myVariable," "MYVARIABLE," and "MyVariable" would all refer to the same identifier. Few programming languages are case-insensitive; SQL is one example where table names and commands are often case-insensitive.

Case Sensitive-Insensitive: Some programming languages or contexts can have mixed behavior regarding case sensitivity.

For example, a language might be case-sensitive for variable names but case-insensitive for keywords. In this case, variable names would need to be written with the exact casing, while keywords could be written in any combination of uppercase and lowercase letters. However, it's essential to know the specific rules for each language or context to avoid errors.

Java is a case-sensitive language. It requires that variable names, function names, and keywords be written exactly with the correct letter casing.

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

In Java, an Identity Conversion is a type of implicit type conversion (also known as type casting) that occurs when a value is assigned to a variable of the same type, or when a method is invoked with an argument matching the parameter type exactly. In other words, no data loss or modification is required during an identity conversion, as the source and destination types are compatible and interchangeable.

Java supports several implicit conversions, including identity conversion, widening conversion, and narrowing conversion. Identity conversion is the most straightforward, as it only involves assigning a value to the same type of variable or passing an argument that exactly matches the method parameter type.

Here are two examples to illustrate the concept of Identity Conversion in Java:

Example 1: Identity Conversion for Primitives

```
public class IdentityConversionExample {
    public static void main(String[] args) {
        int intValue = 42;

        // Identity Conversion: int to int (no data loss or modification)
        int result = intValue;

        System.out.println("Result: " + result); // Output: Result: 42
    }
}
```

In Java, an Identity Conversion is a type of implicit type conversion (also known as type casting) that occurs when a value is assigned to a variable of the same type, or when a method is invoked with an argument matching the parameter type exactly. In other words, no data loss or modification is required during an identity conversion, as the source and destination types are compatible and interchangeable.

Java supports several implicit conversions, including identity conversion, widening conversion, and narrowing conversion. Identity conversion is the most straightforward, as it only involves assigning a value to the same type of variable or passing an argument that exactly matches the method parameter type.

Here are two examples to illustrate the concept of Identity Conversion in Java:

Example 1: Identity Conversion for Primitives

java

```java
public class IdentityConversionExample {
    public static void main(String[] args) {
        int intValue = 42;

        // Identity Conversion: int to int (no data loss or modification)
        int result = intValue;

        System.out.println("Result: " + result); // Output: Result: 42
    }
}
```

In this example, we have an **`intValue`** of type **`int`**. When we assign **`intValue`** to the **`result`** variable, it is an identity conversion because both variables are of the same type **`int`**. No data loss or modification occurs during this assignment.

Example 2: Identity Conversion for Objects

```
public class IdentityConversionExample {
    public static void main(String[] args) {
        String strValue = "Hello, World!";

        // Identity Conversion: String to String (no data loss or modification)
        String result = strValue;

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

In this example, we have a **`strValue`** of type **`String`**. When we assign **`strValue`** to the **`result`** variable, it is an identity conversion because both variables are of the same type **`String`**. Just like in the previous example, no data loss or modification occurs during this assignment.

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

In Java, Primitive Widening Conversion (also known as automatic type promotion or widening) is an implicit type conversion that occurs when a value of a smaller data type is assigned to a variable of a larger data type. In other words, when there is no risk of losing data, Java automatically converts smaller data types to larger data types.

Example 1: Widening Conversion from byte to short

```
public class WideningConversionExample {
    public static void main(String[] args) {
        byte byteValue = 100;

        // Widening Conversion: byte to short (no data loss)
        short shortValue = byteValue;

        System.out.println("Short Value: " + shortValue); // Output: Short Value: 100
    }
}
```

Example 2: Widening Conversion from int to long

```
public class WideningConversionExample {
    public static void main(String[] args) {
        int intValue = 1000;

        // Widening Conversion: int to long (no data loss)
        long longValue = intValue;

        System.out.println("Long Value: " + longValue); // Output: Long Value: 1000
    }
}
```

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

Compile-time constants have values known and fixed during compilation, and they are used in constant expressions.

Run-time constants have values calculated during program execution, and they can depend on dynamic factors like user input or system conditions.

Example of a Compile-time Constant:

```
public class CompileTimeConstantExample {

   final int MAX_VALUE = 100;

   public static void main(String[] args) {

      int halfMax = MAX_VALUE / 2; // Compile-time constant expression

      System.out.println(halfMax); // Output: 50

   }

}
```

In this example, `MAX_VALUE` is a compile-time constant with a value of 100. The value of `halfMax` is calculated during compilation because `MAX_VALUE` is a compile-time constant, and the result is directly inserted into the bytecode.

Example of a Run-time Constant:

```
import java.util.Scanner;

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

      final int MAGIC_NUMBER = inputNumber * 5; // Run-time constant
      System.out.println("Magic Number: " + MAGIC_NUMBER);
   }
}
```

In this example, `MAGIC_NUMBER` is a run-time constant. Its value is calculated based on the user's input (during runtime) and then stored in the constant variable. The value of `MAGIC_NUMBER` depends on the user's input and cannot be known 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 conversions are performed automatically by Java when a value of a larger data type is assigned to a variable of a smaller data type. In this case, there is a potential risk of data loss because the target data type may not be able to represent the full range of values of the source data type.

```
Example:
int intValue = 1000;
byte byteValue = intValue; // Implicit narrowing from int to byte
```

Explicit Narrowing Conversions (Casting): Explicit narrowing conversions, also known as casting, involve manually converting a value from a larger data type to a smaller data type. This is achieved by using a cast operator (`(targetType)`) before the value, indicating the intended data type.

```
Example:
double doubleValue = 123.45;
int intValue = (int) doubleValue; // Explicit narrowing from double to int
```

In this example, `doubleValue` is of type `double`, and `intValue` is of type `int`. We explicitly cast `doubleValue` to `int` using `(int)` before the value, which tells Java to convert the `double` value to an `int`. The fractional part is discarded, and `intValue` will be assigned the value 123.

Conditions for Implicit Narrowing Primitive Conversion:

1. The value being assigned must be within the range of the target data type. If the value cannot be represented by the target type, a compilation error will occur.

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

A narrowing primitive conversion may lose information about the overall magnitude of a numeric value and may also lose precision and range.

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?"

In Java, a long data type is a 64-bit signed integer, and a float data type is a 32-bit single-precision floating-point number. Despite the difference in their bit sizes, it is possible to assign a long value to a float variable without explicit casting. This is an example of implicit widening conversion, where a value of a smaller data type is automatically promoted to a larger data type.

The seeming discrepancy arises from the fact that floating-point numbers use a different representation format, known as IEEE 754.

The IEEE 754 standard for floating-point numbers includes a sign bit, exponent bits, and a fraction (also called mantissa) bits. The number of bits allocated to each part determines the range and precision of the floating-point value.

For a 32-bit float, the representation is as follows:

- 1 bit for the sign (positive or negative)
- 8 bits for the exponent
- 23 bits for the fraction

This format allows the float to represent a wide range of values but with limited precision. It sacrifices some precision to accommodate a larger range of values. On the other hand, a 64-bit long is a signed integer with 63 bits for the actual value and 1 bit for the sign. Long integers can represent larger integer values with full precision, but they lack the fractional component. When you assign a long value to a float variable, Java implicitly performs a conversion by preserving the integral value and discarding the fractional part. This can result in a loss of precision if the long value is too large to be represented accurately as a float.

example to illustrate the implicit widening conversion from long to float:

```
public class ConversionExample {

    public static void main(String[] args) {

        long longValue = 1234567890123L;

        float floatValue = longValue; // Implicit widening conversion from long to float

        System.out.println("Long Value: " + longValue);

        System.out.println("Float Value: " + floatValue);

    }

}
```

Output:

```
Long Value: 1234567890123
Float Value: 1.23456788E12
```

As you can see, the `floatValue` has lost some precision compared to the original `longValue`, as the float cannot represent the full precision of the long. This discrepancy is due to the different representations and ranges of the two data types.

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?

The choice of using `int` as the default data type for integer literals and `double` as the default data type for floating-point literals in Java is primarily based on the balance between usability, precision, and backward compatibility.

1. **Usability and Convenience:**
Integers are one of the most commonly used data types in programming, and they are frequently used to represent whole numbers. By making `int` the default data type for integer literals, Java promotes a more intuitive and user-friendly experience for developers. Most of the time, when an integer literal is encountered in the code, it is intended to be stored as an `int`.

Similarly, `double` is the most common choice for floating-point numbers in Java. By making `double` the default data type for floating-point literals, the language encourages the use of the most precise floating-point representation available. Floating-point numbers are often used to represent real-world measurements, calculations, and other precise data, and `double` provides sufficient precision for most applications.

2. **Precision and Range:**
Choosing `int` as the default data type for integer literals (32-bit) allows Java to handle a wide range of whole number values from -2^31 to 2^31-1. This range covers a significant portion of integer values typically encountered in programming tasks without sacrificing performance or memory.

Using `double` as the default data type for floating-point literals (64-bit) provides a balance between precision and memory usage. `double` can accurately represent a wide range of real numbers, from very small to very large, while offering sufficient precision for most practical calculations. It is worth noting that `float` (32-bit) has a smaller range and precision than `double`, and it can lead to potential precision errors in complex calculations.

3. **Backward Compatibility:**
When Java was designed, the choice of `int` and `double` as the default data types for integer and floating-point literals was made to ensure backward compatibility with older programming languages like C and C++. These choices made it easier for developers transitioning from C/C++ to Java to understand and write code, as the default data types were already familiar to them.

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

Implicit narrowing primitive conversion in Java is limited to byte, char, int, and short because these data types represent integer values, and their ranges are all subsets of each other.

1. **byte**: 8-bit signed integer (-128 to 127)
2. **char**: 16-bit unsigned integer (0 to 65535)
3. **short**: 16-bit signed integer (-32,768 to 32,767)
4. **int**: 32-bit signed integer (-2^{31} to $2^{31} - 1$)

As you can see, char and short have the same size (16 bits) and cover the same range, but char represents only non-negative values (0 to 65535) due to being unsigned. On the other hand, byte and int have different sizes (8 and 32 bits, respectively) and cover different ranges.

When narrowing, we convert from a larger data type to a smaller data type. For example, converting from int to short or from int to char would be narrowing because we're reducing the size of the data type and may lose some of the value's range. However, because char and short have the same range, and both are subsets of int, implicit narrowing conversion is allowed between them.

Explicit narrowing conversions (using casting) are required to convert between other data types, such as int to byte, long to int, etc., as these conversions may lead to potential data loss, and the programmer needs to explicitly indicate their intention.

Allowing implicit narrowing conversion only between byte, char, short, and int ensures that the conversions are well-defined, and the programmer is more aware of potential data loss. Java's strict typing system promotes safety and avoids unintentional behavior in numeric conversions.

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

Widening conversion, also known as automatic type promotion, occurs when a value of a smaller data type is assigned to a variable of a larger data type.

Narrowing conversion occurs when a value of a larger data type is assigned to a variable of a smaller data type.

Conversion from short to char: The conversion from `short` to `char` is neither a widening conversion nor a narrowing conversion. It is a special case where both data types have the same size (16 bits), but their interpretation is different.

- `short`: A 16-bit signed integer (-32,768 to 32,767)
- `char`: A 16-bit unsigned integer (0 to 65,535)

In a widening conversion, the target type must have a larger range than the source type, and in a narrowing conversion, the target type must have a smaller range. In the case of `short` and `char`, both data types have the same size but represent different ranges of values. Because of this mismatch in interpretation, explicit casting is required when converting between `short` and `char`, and it is not classified as either a widening or a narrowing conversion.

Example of Conversion from short to char (with explicit casting):

```
short shortValue = 100;
char charValue = (char) shortValue; // Explicit casting between short and char
```

In this example, we explicitly cast `shortValue` to `char` using `(char)` before the value. The value 100 is represented as 'd' in the ASCII character set. The explicit casting is required because `short` and `char` represent different ranges of values, even though they have the same bit size.