

PROGRAMMING FUNDAMENTALS – ASSIGNMENT 2

Q1. 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

Statically typed programming languages do type checking at *compile-time*.

In a statically typed language, the data types of variables are explicitly declared and known at compile time. The compiler performs type checking and ensures that the data types are used correctly throughout the program. Once a variable is declared with a specific data type, its type cannot change during runtime.

The main advantage here is that all kinds of checking can be done by the compiler, and therefore a lot of trivial bugs are caught at a very early stage.

Examples: C, C++, Java, Rust, Go, Scala

Dynamically Typed Language

Dynamically typed programming languages do type checking at *run-time*.

In dynamically typed languages, variable data types are determined implicitly during runtime, and they can be changed throughout the program's execution. Type checking occurs at runtime, making the code more flexible, as variables can hold values of different types at different stages in the program. Unlike statically typed languages, where types are known at compile time, dynamically typed languages offer greater flexibility but may be prone to runtime errors if a variable's type is not handled appropriately.

Examples: Perl, Ruby, Python, PHP, JavaScript, Erlang

Strongly Typed Language

Strongly typed language has stricter typing rules at compile time, which implies that errors and exceptions are more likely to happen during compilation.

Strongly typed languages prioritize safety and consistency by enforcing strict data type usage. This approach helps minimize errors and unexpected behavior, as it ensures that operations are carried out with compatible data types. If a particular operation requires different data types, the programmer must explicitly convert the data types to perform the operation safely.

Examples: java, C++, swift

Loosely Typed Language

In a loosely typed language, also referred to as a weakly typed language, the type system is more permissive, allowing implicit type conversions and the mixing of data types without the need for explicit conversions.

Loosely typed languages provide greater flexibility to the programmer, as they allow variables and expressions of different data types to be used together without strict restrictions. For example, you can perform operations between variables of different types without explicitly converting them.

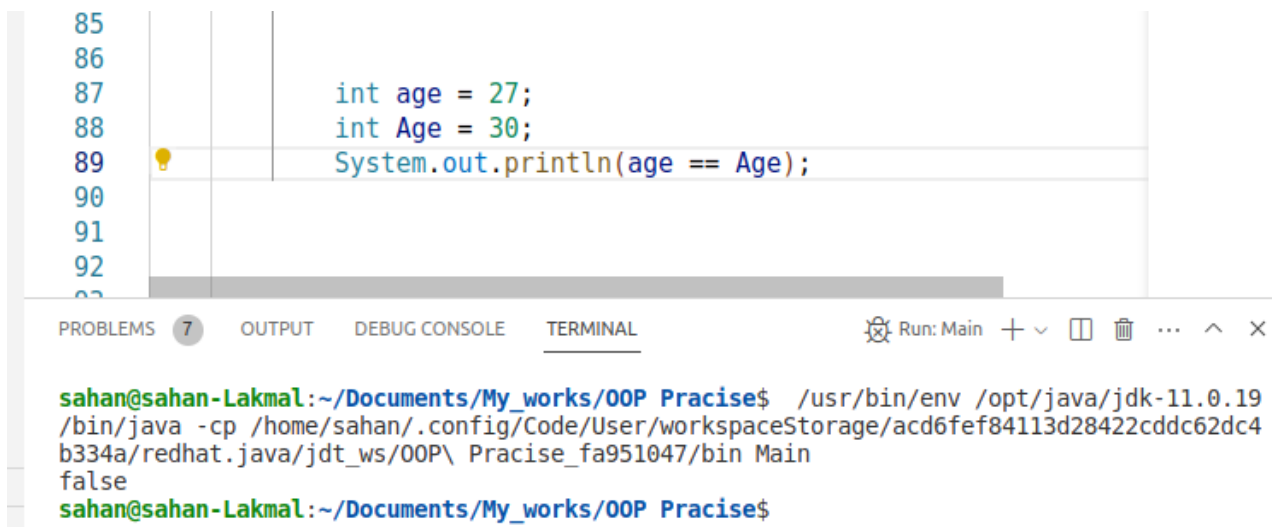
However, this flexibility comes with potential risks. Loosely typed languages may lead to subtle errors and bugs, as the compiler or interpreter might perform implicit conversions that the programmer did not intend. This behavior can sometimes result in unexpected outcomes, making it crucial for developers to be cautious and aware of potential type-related issues.

Examples: PHP, Perl

Q2. "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 programming language, the differentiation between uppercase and lowercase letters holds significant importance. This means that identifiers, such as variable names, function names, class names, etc., must precisely match the case used during their declaration for the language to recognize them correctly.



The screenshot shows an IDE with a Java file. The code is as follows:

```
85  
86  
87     int age = 27;  
88     int Age = 30;  
89     System.out.println(age == Age);  
90  
91  
92
```

The IDE's output window shows the following command and output:

```
sahan@sahan-Lakmal:~/Documents/My_works/OOP Pracise$ /usr/bin/env /opt/java/jdk-11.0.19  
/bin/java -cp /home/sahan/.config/Code/User/workspaceStorage/acd6fef84113d28422cddc62dc4  
b334a/redhat.java/jdt_ws/OOP\ Pracise_fa951047/bin Main  
false  
sahan@sahan-Lakmal:~/Documents/My_works/OOP Pracise$
```

Case Insensitive

In a case-insensitive language, uppercase and lowercase letters do not carry any significance, and identifiers are considered equivalent regardless of the case used. This means that if a variable is declared as "myVariable," it would be treated the same way whether referred to as "myvariable," "MyVariable," or "MYVARIABLE" in a case-insensitive language. The language's parser or interpreter would not distinguish between different letter cases in identifiers.

Example (VB.NET - Case Insensitive):

Dim name As String = "John"

Dim NAME As String = "Jane" ' These are the same variable due to case insensitivity

Case Sensitive – Insensitive

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:

```
90
91     String name = "Sahan";
92     String Name = "sahan";
93
94     System.out.println(name.toLowerCase() == Name);
95
96
97
98
99
100 }
101 }
```

PROBLEMS 7 OUTPUT DEBUG CONSOLE TERMINAL Run: Main + v [] [] ... ^ x

```
false
sahan@sahan-Lakmal:~/Documents/My_works/OOP Practise$ cd /home/sahan/Documents/My_works/OOP\ Practise ; /usr/bin/env /opt/java/jdk-11.0.19/bin/java -cp /home/sahan/.config/Code/User/workspaceStorage/acd6fef84113d28422cddc62dc4b334a/redhat.java/jdt_ws/OOP\ Practise_fa951047/bin Main
false
```

JavaScript:

```
>> var name = "Sahan";
    var Name = "sahan";

    console.log(name.toLowerCase() == Name);

true

← undefined
```

Java's Classification

Java is classified as a case-sensitive programming language, meaning that in Java, identifiers like variable names, method names, and class names must be written using the correct letter case for the language to identify and interpret them accurately.

Q3. Explain the concept of Identity Conversion in Java? Please provide two examples to substantiate your explanation.

Identity conversion in Java refers to a type of conversion that occurs when a value is assigned to a variable having the same data type as the value. In simpler terms, no casting or type conversion is necessary as the types are already compatible. It involves a direct and straightforward assignment of a value to a variable with a matching data type.

```
int age = 30; //"age" is an integer variable
int years = age; // Identity conversion

/*
 * In this example, the variable "age" is assigned the value 30
 * The variable "years" is assigned the value of "age."
 * Since both variables are of type int, it is an identity
 * conversion.
 */

double pi = 3.142; // "pi" is a double variable
double approxPi = pi; // Identity conversion

/*
 * In this example, the variable "pi" is assigned the value 3.14159,
 * The variable "approxPi" is assigned the value of "pi."
 * Both variables are of type double, so it is an identity
 * conversion,
 */
```

Q4. Explain the concept of Primitive Widening Conversion in Java with examples and diagrams.

Primitive Widening Conversion, also referred to as Automatic or Implicit Conversion, is a type conversion in Java where a value of a smaller data type is assigned to a variable of a larger data type. In this process, no information is lost, and the Java compiler handles the conversion automatically.

The term "widening" is used because the target data type has a broader range and can hold all potential values of the source data type without sacrificing precision.

In summary, 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.

Java:

1. byte to short, int, long, float, or double
2. short to int, long, float, or double
3. char to int, long, float, or double

4. int to long, float, or double
5. long to float or double
6. float to double

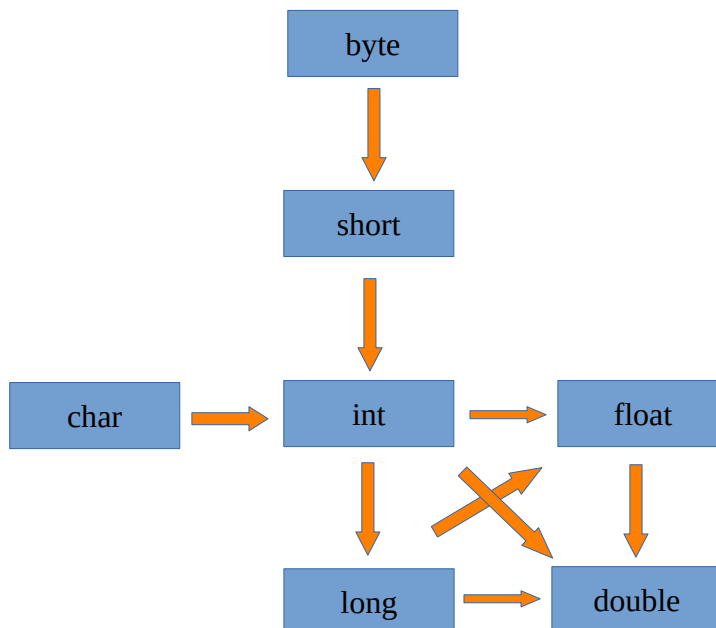
```
int intValue1 = 100;  
long longValue = intValue1; // Implicit conversion from int to long  
  
System.out.println(longValue);
```

```
char charValue = 'A';  
int intValue2 = charValue; // Implicit conversion from char to int  
  
System.out.println(intValue2);
```

Outputs:

```
sahan@sahan-Lakmal:~/Documents/My_works/OOP Practise$ cd /home/sahan/Documents/My_works/OOP\ Practise ; /usr/bin/env /opt/java/jdk-11.0.19/bin/java -cp /home/sahan/.config/Code/User/workspaceStorage/acd6fef84113d28422cd62dc4b334a/redhat.java/jdt_ws/OOP\ Practise_fa951047/bin Main  
100  
65  
sahan@sahan-Lakmal:~/Documents/My_works/OOP Practise$
```

Diagram:



Q5. Explain the the difference between run-time constant and Compile-time constant in java with examples.

Compile-time constant:

A compile-time constant in Java refers to a value that can be evaluated by the Java compiler and replaced with its actual value during the compilation process. These constants are already known at compile-time, and their values are determined before the program runs.

```
final int MAX_SIZE = 10;
```

In the given example, MAX_SIZE represents a compile-time constant. The value 10 is already known to the compiler during the compilation phase. Consequently, the Java compiler will replace all instances of MAX_SIZE with the value 10 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, i.e., during the execution of the program. 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(10);
```

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(10) 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.

Choosing between compile-time and run-time constants depends on the specific requirements of a program. Compile-time constants offer improved efficiency and performance by resolving values during compilation, while run-time constants provide adaptability and the ability to handle dynamic situations during program execution.

Q6. 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, also referred to as automatic narrowing 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.

Examples:

```
byte myByte1 = 10;  
short myShort1 = 100;
```

Above two examples 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.

Let's illustrate this with your example:

```
byte myByte1 = 10;
```

The value `10` is a constant because it is a literal value specified directly in the code.

1. The value `10` falls within the valid range of the `byte` data type, which is `-128` to `127`.
2. The value `10` is known at compile-time because it is directly specified in the source code.

Since all the conditions are met, the Java compiler allows the implicit narrowing primitive conversion, and the `int` literal value `10` is implicitly narrowed to a `byte`, resulting in a valid assignment to the `byte` variable `myByte1`.

```
int myInt = 12;  
byte myByte2 = myInt; //Showina an compile time error
```

However, if we use `final` keyword make the variable to constant, it should work.

```
final int myInt = 12;  
byte myByte2 = myInt;
```

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.

Example:

```
double myDouble= 1234.57;
int myInt = myDouble;//Implicit narrowing not allowed
```

```
double myDouble= 1234.57;
int myInt = (int) myDouble;//Casting required
```

Output:

```
sahan@sahan-Lakmal:~/Documents/My_works/OOP_Pracise$ cd /home/sahan/Documents/My_works/OOP\ Pracise ; /u:
in/env /opt/java/jdk-11.0.19/bin/java -cp /home/sahan/.config/Code/User/workspaceStorage/acd6fef84113d284:
dc62dc4b334a/redhat.java/jdt_ws/OOP\ Pracise_fa951047/bin Main
1234
```

By using explicit casting (`int`), you are indicating that you are willing to truncate the decimal part and convert the `double` value to an `int`. In this case, the result will be `myInt = 1234`, as the decimal part `0.57` is truncated during the conversion.

Q7. 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 ability to assign a `long` data type (64 bits) into a `float` data type (32 bits) is possible due to the fundamental difference in how these data types represent and store values.

In Java, the `long` data type is a 64-bit signed integer type that can store values within the range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, offering 64 bits of precision. In contrast, the `float` data type is a 32-bit single-precision floating-point type capable of representing a broader range of values, but at the cost of reduced precision.

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`.

Example:

```
long myLong = 123456782;
float myFloat = myLong;

System.out.println(myLong);
System.out.println(myFloat);
```


Output:

```
sanam@sanam-Lakmal:~/documents/my_works/00P_Pracise$ cd /home/sanam/documents/my_works/00P\ P
in/env /opt/java/jdk-11.0.19/bin/java -cp /home/sanam/.config/Code/User/workspaceStorage/acd6fe
dc62dc4b334a/redhat.java/jdt_ws/00P\ Pracise_fa951047/bin Main
123456782
1.23456784E8
```

In this example, we have a `long` value `123456782`, and we assign it to a `float` variable `myFloat`. The output shows that the `myLong` remains the same, but when represented as a `float`, the value becomes `1.23456784E8`. Again, the `float` representation loses precision, and the original `long` value cannot be exactly represented as a `float`.

Q8. 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.

Q9. Why does implicit narrowing primitive conversion only take place among byte , char , int , and short ?

byte, char, int, and short: These data types support implicit primitive conversions because they have a smaller range than long. Implicit conversion from a smaller data type to a larger one is generally safe as no information is lost during the conversion. For example, converting from byte to int or from char to short does not result in a loss of data, and the value can be represented safely.

Implicit conversions from long to smaller data types like byte, char, int, or short are not allowed. This is because narrowing conversions from a larger data type (like long) to a smaller one can result in data loss or truncation, which can lead to unexpected behavior. To ensure type safety, Java requires explicit casting in such cases to indicate that the programmer is aware of the potential data loss.

Here's the relevant excerpt from the JLS:

A widening conversion of an int or a long value to float, or of a long value to double, may result in the 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 (§4.2.4). A widening conversion of an int value to a long value simply sign-extends the 32-bit two's-complement representation of the int value to fill the 64-bit two's-complement representation of the long value.

Notice that it mentions widening conversions of int to long, but it doesn't include conversions from long to int. This confirms that implicit narrowing conversions from long to int, short, or byte are not allowed in Java. These conversions would require explicit casting to avoid the risk of data loss.

Q10. 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.