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?”

   Answer :

   - Statically typed language mean that before source code is compiled, the type associated with each and every single variable must be known (ex: java, C, C+, C#, etc…)

   - Dynamically typed language mean that perform type checking at run-time and user don’t have to explicitly specify data type of variables. (ex: JavaScript, Python, PHP, Ruby. Etc …)

   - Strongly typed language mean that demand the specifications of the data types(ex: java, Python, Ruby, C, C#, etc…)

   - Loosely typed language mean that is not concern about the data type and has a looser type rules also may perform implicit type conversion at runtime(ex: JavaScript, C, etc..)

   - Java is a Statically type and Strongly type programming 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?”

   Answer

   - Case Sensitive:

     A language is considered case sensitive if it differentiates between uppercase and lowercase letters in identifiers. This means that "identifier," "Identifier," and "IDENTIFIER" are considered distinct and separate identifiers in the language.

     Ex:

```
# Python (case sensitive)
variable = 10
Variable = 20

# The two variables are different due to case sensitivity
print(variable)  # Output: 10
print(Variable)  # Output: 20
```

- A case-insensitive language is a programming language in which variations in the capitalization of letters in identifiers (eg, variable names, function names, and keywords) are not considered significant. In other words, the language treats uppercase and lowercase letters as equivalent, and identifiers with different cases are treated as the same.

  In a case-insensitive language, "identifier," "Identifier," and "IDENTIFIER" would be considered identical, and there would be no distinction between them.

  Ex :
```
-- SQL (case insensitive)
SELECT column1 FROM tableName;      -- [These statements are equivalent
select COLUMN1 from TABLENAME;   -- due to case insensitivity.]
```

- In some programming languages, the case sensitivity of identifiers may depend on the context or options set by the programmer or the environment. In such cases, certain parts of the language could be case sensitive while others are case insensitive.

  Ex:
```
// JavaScript is generally case sensitive
let variable = 10;
let Variable = 20;

console.log(variable);  // Output: 10

// However, when accessing object properties, JavaScript is case insensitive
const obj = { SomeProperty: 'Value' };
console.log(obj.someproperty);  // Output: Value (even though the case is different)
```

- Java is a case-sensitive programming language. This means that it treats uppercase and lowercase letters in identifiers distinctly. For example, `variable` and `Variable` would be considered as two different identifiers in Java.

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

Answer :

In Java, Identity Conversion is a type conversion that occurs when a value is assigned to a variable or parameter of the same type without any explicit cast or conversion operator. In other words, the data type of the value being assigned is the same as the data type of the variable or parameter it is being assigned to. In such cases, the Java compiler automatically performs an identity conversion because the types are already compatible.

- Ex_1:

```
public class IdentityConversionExample {
    public static void main(String[] args) {
        int num = 42;
        int result = num; // Identity conversion from int to int
        System.out.println("Result: " + result); // Output: Result: 42
    }
}
```

Ex_2 :

```
public class IdentityConversionExample {
    public static void main(String[] args) {
        String name = "John";
        String personName = name; // Identity conversion from String to
String
        System.out.println("Person's name: " + personName); // Output:
Person's name: John
    }
}
```

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

Answer :

- In Java, primitive widening conversion is an automatic and implicit process of converting a smaller data type to a larger data type. This conversion is performed when the data types involved are compatible and no data loss occurs. Widening conversion is also known as "promotion" as the value is promoted to a wider type to accommodate the larger range of values.

  The Java language has a set of predefined rules for primitive widening conversions. These conversions are safe because they don't result in any loss of data. The following diagram illustrates the hierarchy of data types in Java, from smaller to larger:
  byte -> short -> int -> long -> float -> double

  Ex 1: Widening from `byte` to `int`

```
byte smallNumber = 10;
int largerNumber = smallNumber; // Widening conversion from byte to int
```

  Ex 2: Widening from `short` to `double`

```
short smallValue = 1000;
double largeValue = smallValue; // Widening conversion from short to double
```

Ex 3: Widening from `float` to `double`

```
float smallFloat = 3.14f;
double largerDouble = smallFloat; // Widening conversion from float to double
```

Ex 4: Widening from `char` to `int`

```
char character = 'A';
int charCode = character; // Widening conversion from char to int (ASCII value of 'A' is 65)
```

In all these examples, a smaller data type is automatically widened to a larger data type, and there is no loss of data or precision.

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

Answer

In Java, both run-time constants and compile-time constants are used for values that remain constant throughout the execution of a program. However, there are fundamental differences between them in terms of when their values are determined and how they are stored in the compiled bytecode.

- Compile-time constant:

- A compile-time constant is a value that can be determined by the compiler at the time of compilation. These values are substituted directly into the code during the compilation process, which improves performance and eliminates the need to look up the value at runtime.
- Compile-time constants are declared using the `final` keyword and are typically simple primitive types (ex., `int`, `double`, `char`, etc.), strings, or expressions involving other compile-time constants.
- The value of a compile-time constant must be known at compile time, and it cannot depend on runtime computation or variables.
- Once the value of a compile-time constant is determined at compile time, it becomes a part of the bytecode, and the variable itself is not used during runtime.

ex :

```
public class ConstantsExample {
    public static final int MY_CONSTANT = 42; // This is a compile-time constant.

    public static void main(String[] args) {
        int result = MY_CONSTANT * 2; // The value 42 is substituted at compile time.
        System.out.println("Result: " + result);
    }
}
```

◆

happen during the runtime of the program. Unlike compile-time constants, the value of run-time constants may not be known until the program is executed and variables are assigned values.

- Run-time constants are declared using the `final` keyword, just like compile-time constants, but their values can be determined during program execution based on certain conditions or dynamic calculations.

Ex:

```
public class RuntimeConstantsExample {
    public static final int MY_CONSTANT;

    static {
        // The value of MY_CONSTANT is determined at runtime during the static initialization block.
        // In this example, we assume that the value is read from a configuration file or computed based on some logic.
        MY_CONSTANT = readFromConfigFileOrCalculateValue();
    }

    public static void main(String[] args) {
        int result = MY_CONSTANT * 2; // The value is determined at runtime.
        System.out.println("Result: " + result);
    }
}
```

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?

Answer :

In programming languages like C, C++, and Java, data type conversions are necessary when you want to assign a value of one data type to a variable of another data type. Conversions can be broadly categorized into two types: Implicit (Automatic) Narrowing Primitive Conversions and Explicit Narrowing Conversions (Casting).

Implicit (Automatic) Narrowing Primitive Conversions: Implicit narrowing conversions happen automatically by the compiler when a value of a larger data type is assigned to a variable of a smaller data type. This conversion is considered narrowing because it may result in a loss of information. For example, converting a larger range floating-point number to a smaller range integer can result in truncation of the fractional part.

Ex:

```
double doubleValue = 3.14159;
int intValue = doubleValue; // Implicit narrowing conversion from double to int
```

Conditions for Implicit Narrowing Primitive Conversion: An implicit narrowing primitive conversion can occur if the following conditions are met:

1. The target data type has a smaller range and precision than the source data type.
2. The source value is within the range of the target data type.
3. No precision loss occurs during the conversion. For example, converting from a floating-point type to an integer type may truncate the fractional part, which is considered a loss of precision.

If any of these conditions are not met, the compiler will generate an error or a warning, indicating the potential loss of data.

In summary, implicit narrowing primitive conversions happen automatically by the compiler when converting a value of a larger data type to a variable of a smaller data type. However, you need to be careful with these conversions as they can lead to data loss, and it's important to ensure that the source value fits within the target data type's range and precision. If there's a risk of data loss, explicit narrowing conversions (casting) should be used instead to indicate your intention and handle the conversion explicitly.

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?”

Answer :

The process of assigning a long data type (64 bits) to a float data type (32 bits) in Java involves a type conversion known as "narrowing primitive conversion" or "type casting." This conversion can result in potential loss of precision or data truncation because a float can represent a smaller range of values with less precision compared to a long.
Here's how the conversion works:

1. Data truncation: When you assign a long value to a float, the compiler will truncate the least significant bits of the long value to fit it into the float's 32-bit representation. This means that if the long value has bits in positions beyond the 32nd bit, they will be discarded during the conversion.
2. Precision loss: A float data type can represent fractional values but with reduced precision compared to a long. This means that some values in the long range may not be accurately represented in a float, leading to potential rounding errors or loss of information after the decimal point.

To explicitly perform the conversion, you can use a type cast like this:

```
long longValue = 123456789012345678L;
float floatValue = (float) longValue; // Explicit type casting from long to float
```

However, it's essential to be aware of the limitations of this conversion. If the long value is too large to be accurately represented in a float, you may encounter unexpected behavior due to precision loss. Additionally, if the long value contains significant bits beyond the 32nd bit, those bits will be lost during the conversion.

As a best practice, when performing this kind of conversion, you should only do so when you are confident that the values you are converting will not lose essential information due to the differences in data representation between the two types. Always test and validate the results of such conversions to ensure they meet your requirements.

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?

Answer :

In Java, the default data types for integer literals and floating-point literals are `int` and `double`, respectively. This design decision was made by the Java language designers to balance between convenience, efficiency, and safety.

1. Convenience: Using `int` as the default data type for integer literals is convenient because most integer values used in everyday programming fit within the range of the `int` data type. By defaulting to `int`, Java allows programmers to write code more succinctly without explicitly specifying the data type for most integer literals.

2. Efficiency: The `int` data type is usually the most efficient integer type on most platforms, as it generally matches the size of the native integer representation of the underlying hardware. Using `int` as the default data type allows Java to take advantage of the hardware's native integer operations, optimizing performance for most integer calculations.

3. Backward Compatibility: Java was designed with a focus on backward compatibility. When Java was initially developed, memory and processing power were more limited than they are today. At that time, using `int` as the default integer data type and `double` as the default floating-point data type provided a reasonable balance between performance and storage requirements.

4. Avoiding Precision Loss: Floating-point numbers have limited precision and are subject to rounding errors when performing certain calculations. Using `double` as the default floating-point data type helps to minimize precision loss in most common calculations, making it suitable for a wide range of applications.

   While this design decision offers many benefits for most use cases, it's important for developers to be aware of potential issues related to overflow and precision errors when dealing with large integers or critical numerical computations. In such cases, programmers can explicitly choose appropriate data types like `long` for larger integers or use specialized libraries for precise decimal arithmetic if needed.

   Java's design philosophy has always been centered around simplicity, portability, and maintainability, and the default data type choices for integer and floating-point literals reflect these principles while providing a balance between practicality and performance.

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

Answer :

Implicit narrowing primitive conversion in Java is a process where a value of a larger data type is automatically converted to a value of a smaller data type without any explicit casting. This kind of conversion can potentially result in a loss of data or precision, as the target data type may not be able to represent all possible values of the source data type.

The reason why implicit narrowing is only allowed among `byte`, `char`, `int`, and `short` data types in Java is primarily related to the size and range

of these data types.

1. `byte`: It is an 8-bit data type with a range of -128 to 127. Since its range is relatively small, it is frequently used for memory-efficient storage of small values.
2. `char`: It is a 16-bit unsigned data type representing a Unicode character in the range 0 to 65535. It is used to store individual characters.
3. `int`: It is a 32-bit data type with a wider range of -2^31 to 2^31-1, allowing it to store larger integer values.
4. `short`: It is a 16-bit data type with a range of -32,768 to 32,767. It is useful for saving memory when the value is known to be within this range.

   The limited set of implicit narrowing conversions is a safety measure taken by Java to avoid potential data loss when converting between different data types. Allowing implicit narrowing for other data types could lead to unintended consequences, such as overflow or unexpected values, which might be hard to detect and debug.

   For other conversions involving larger data types like `long` or floating-point types (`float` and `double`), explicit casting is required to make it clear to the developer that they are intentionally truncating the data and accepting the risk of data loss or precision issues. This makes the code more explicit and easier to understand. For example, the following explicit cast is required when converting from `long` to `int`:

```
long bigValue = 1234567890L;
int smallValue = (int) bigValue; // Explicit casting is required here
```

   In summary, implicit narrowing is limited to `byte`, `char`, `int`, and `short` in Java to ensure type safety and to avoid unintentional data loss when converting between different data types. For other data type conversions, explicit casting is required to make the code more explicit and prevent potential issues related to data loss and precision.

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

Answer :

In programming, "Widening and Narrowing Primitive Conversion"

refers to the process of converting one data type to another, where the destination data type either has a wider range or a narrower range than the source data type. The terms "widening" and "narrowing" indicate whether the conversion may result in a potential loss of information or not.

Widening Primitive Conversion:

- This type of conversion occurs when a data type with a smaller range is converted to a data type with a larger range, and no information is lost during the conversion.
- For example, converting an `int` to a `long` or converting a `float` to a `double` are widening primitive conversions because the destination data types can hold values of a wider range than the source data types.

Narrowing Primitive Conversion:

- This type of conversion occurs when a data type with a larger range is converted to a data type with a smaller range, and there is a possibility of losing information or precision.
- For example, converting a `double` to a `float` or converting an `int` to a `byte` are narrowing primitive conversions because the destination data types have a narrower range than the source data types, and data loss may occur if the value doesn't fit in the smaller type.

  Now, let's discuss why the conversion from `short` to `char` is not classified as Widening and Narrowing Primitive Conversion:

  In most programming languages, including Java, both `short` and `char` are 16-bit numeric data types. Despite this, they serve different purposes:
  - `short`: It is a signed data type that can represent both positive and negative values. It has a range from -32,768 to 32,767.
  - `char`: It is an unsigned data type that represents a single 16-bit Unicode character. It has a range from 0 to 65,535.

  Since both `short` and `char` have the same size (16 bits), converting from `short` to `char` doesn't involve a change in the range or size of the data. Instead, it is considered a conversion between different semantic interpretations of the same bit pattern. The only difference lies in how the values are interpreted:

  - `short` values represent integer numbers, which can be positive or negative.
  - `char` values represent Unicode characters.

For example, if you have a `short` variable with a value of -100, converting it to a `char` will not change the bit pattern; it will be interpreted as a Unicode character with a code point of 65,536 - 100 = 65,436. This means that the conversion is not classified as either a widening or narrowing conversion because the bit pattern remains the same, and the data types have the same size. The classification distinction is based on the range and the possibility of data loss or gain during conversion, which is not applicable in this case.

To summarize, the conversion from `short` to `char` is not considered a Widening and Narrowing Primitive Conversion because both data types have the same size and the same number of representable values; the only difference lies in their semantic interpretation.