1. Differences Between Primitive and Reference Data Types

- **Primitive Data Types:**
  - **Definition:** Basic data types provided by the programming language.
  - **Examples:** `int`, `char`, `boolean`, `byte`, `short`, `long`, `float`, `double`.
  - **Size:** Fixed size (e.g., `int` is 4 bytes).
  - **Storage:** Stored directly in the memory location.
  - **Default Values:** Have default values (e.g., `int` defaults to `0`).
  - **Usage:** Typically used for simple, single values.
- **Reference Data Types:**
  - **Definition:** Refer to objects or arrays.
  - **Examples:** Arrays, Strings, Classes (e.g., `String`, `ArrayList`, `CustomClass`).
  - **Size:** Variable size, depending on the object.
  - **Storage:** Stores the reference (or memory address) where the actual data is located.
  - **Default Values:** Default to `null`.
  - **Usage:** Used for storing complex objects and collections of data.

2. Scope of a Variable

- **Local Variable:**
  - **Definition:** Declared within a method, constructor, or block of code.
  - **Scope:** Accessible only within the method or block where it's declared.
  - **Lifetime:** Exists only during the execution of the method or block.
- **Global Variable (Instance/Static Variable):**
  - **Definition:** Declared outside of methods but inside the class.
  - **Scope:** Accessible by any method within the class.
  - **Lifetime:** Exists as long as the object or class is in memory.
  - **Types:**
    - **Instance Variable:** Each object of the class has its own copy.
    - **Static Variable:** Shared among all objects of the class.

3. Why is Initialization of Variables Required?

Initialization of variables is required to:

- **Avoid Unpredictable Behavior:** Uninitialized variables contain garbage values, leading to errors.
- **Ensure Proper Functionality:** Provides a known starting value to the variable.
- **Compiler Requirement:** Some programming languages, like Java, require variables to be initialized before use to avoid compilation errors.

JavaAssignment

Section 1

4. Differences Between Static, Instance, and Local Variables

- **Static Variable:**
  - **Definition:** Declared with the *static* keyword.
  - **Scope:** Belongs to the class, not any specific object.
  - **Lifetime:** Exists for the entire duration of the program.
  - **Access:** Shared among all objects of the class.
- **Instance Variable:**
  - **Definition:** Declared within a class but outside of any method, constructor, or block.
  - **Scope:** Each object of the class has its own copy.
  - **Lifetime:** Exists as long as the object exists.
  - **Access:** Accessed through object references.
- **Local Variable:**
  - **Definition:** Declared within a method, constructor, or block.
  - **Scope:** Limited to the block where it is declared.
  - **Lifetime:** Exists only during the execution of the block/method.

5. Differences Between Widening and Narrowing Casting in Java

- **Widening Casting (Implicit):**
  - **Definition:** Conversion of a smaller data type to a larger data type.
  - **Examples:** *int* to *long*, *float* to *double*.
  - **Automatic:** Performed automatically by the compiler.
  - **Data Loss:** No data loss occurs.
  - **Syntax:** No special syntax needed.

Example:

```
int num = 10;
long longNum = num; // Widening casting
```

  -
- **Narrowing Casting (Explicit):**
  - **Definition:** Conversion of a larger data type to a smaller data type.
  - **Examples:** *double* to *float*, *long* to *int*.
  - **Manual:** Must be explicitly performed by the programmer.
  - **Data Loss:** Possible data loss or precision loss.

- **Syntax:** Requires a cast operator.

Example:
java
```
double num = 10.5;
int intNum = (int) num; // Narrowing casting
```

6. Filling in the Missing Values in the Table

TYPE	SIZE (IN BYTES)	DEFAULT	RANGE
boolean	1 bit	false	true, false
char	2	'\u0000'	'\u0000' to '\uffff'
byte	1	0	-128 to 127
short	2	0	-32,768 to 32,767
int	4	0	-2,147,483,648 to 2,147,483,647
long	8	0L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	0.0f	-3.4E+38 to 3.4E+38
double	8	0.0d	-1.8E+308 to 1.8E+308

7. Define Class as Used in OOP

A **class** in Object-Oriented Programming (OOP) is a blueprint or template that defines the attributes (fields) and behaviors (methods) that objects created from the class will have. A class encapsulates data and functions, allowing the creation of multiple instances (objects) with similar properties and methods.

8. Importance of Classes in Java Programming

- **Encapsulation:** Classes allow bundling of data (attributes) and methods (functions) that operate on the data into a single unit, providing a clear structure and enhancing data protection.
- **Reusability:** Classes can be reused across different programs, promoting code reuse and modularity.
- **Inheritance:** Classes support inheritance, allowing new classes to inherit attributes and methods from existing classes, promoting code reuse and extending functionality.
- **Abstraction:** Classes enable abstraction, hiding the complex implementation details and exposing only the necessary components to the user.
- **Modularity:** Classes help in organizing the code into logical units, making it easier to manage, maintain, and debug.