

Java Assignment – Theory Questions

Q1 Explain the difference between primitive and reference data types with examples.

Ans Primitive data types are basic building blocks of data in Java. They are predefined by the language and store simple values directly in memory. Ex: int, float, double etc.

Characteristics:

- Stored in stack memory.
- Fast and memory-efficient.
- Holds only one value at a time.
- Does not support methods.

Primitive Datatypes:

Datatype	Size
boolean	1 bit
byte	1 byte
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes
char	2 bytes

Example:

```
public class PrimitiveDatatypes {  
    public static void main(String[] args) {  
        int num = 10;  
        char letter = 'J';  
        boolean isJava = true;  
        System.out.println("Number: " + num);  
        System.out.println("Letter: " + letter);  
        System.out.println("Java is fun: " + isJava);  
    }  
}
```

Output:

```
PS C:\Users\Admin\Desktop\Java Assignment> javac PrimitiveDatatypes.java
PS C:\Users\Admin\Desktop\Java Assignment> java PrimitiveDatatypes
Number: 10
Letter: J
Java is fun: true
PS C:\Users\Admin\Desktop\Java Assignment> |
```

Reference Datatypes:

Reference data types are used to **store objects**. They hold a **reference (memory address)** to the actual data rather than the value itself.

Characteristics:

- Stored in heap memory.
- Slower than primitive types.
- Holds multiple values (as objects).
- Can have methods and behaviors.

Examples of Reference Data Types:

- Objects (Instances of Classes)
- Arrays
- Strings
- User-defined types (like interfaces and enums)

Example:

```
public class ReferenceDatatypes {
    String message = "Hello, Java!"; // Reference type (String object)

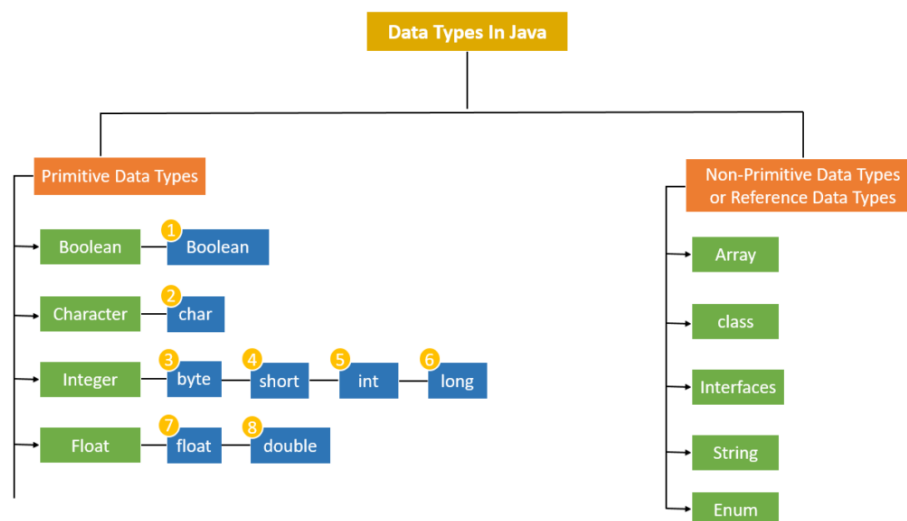
    public static void main(String[] args) {
        ReferenceDatatypes obj = new ReferenceDatatypes();
        System.out.println(obj.message);
    }
}
```

Output:

```
PS C:\Users\Admin\Desktop\Java Assignment> javac ReferenceDatatypes.java
PS C:\Users\Admin\Desktop\Java Assignment> java ReferenceDatatypes
Hello, Java!
PS C:\Users\Admin\Desktop\Java Assignment> |
```

Difference between Primitive and Reference Datatypes

Parameter	Primitive Types	Reference Types
Storage	Store the actual value directly. (Stack Memory)	Store memory addresses to objects. (Heap Memory)
Memory Efficiency	Highly memory-efficient due to direct storage.	Less memory-efficient due to storing references.
Access Speed	Faster access due to direct value storage.	Slightly slower access due to indirection via references.
Usage	Used for simple values (e.g., numbers).	Used for complex objects, arrays, and custom classes.
Sharing Data	Values are independent, not shared.	Can share data among different parts of the program.



Q2 Explain the concept of encapsulation with a suitable example.

Ans Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It refers to the bundling of data (variables) and methods (functions) that operate on that data into a single unit (class) while restricting direct access to some of the object's details. This ensures data security, modularity, and better control over the data. **In Java, Encapsulation is implemented using:**

1. Private variables – to restrict direct access.
2. Public getter and setter methods – to control access to the data.
3. A class that encapsulates the data – all related variables and methods are inside a single class.

Example: Let's consider an example where we create a BankAccount class that encapsulates the account balance and provides methods to deposit and withdraw money securely.

```
class BankAccount {  
    private String accountHolder;  
    private double balance;  
  
    public BankAccount(String accountHolder, double balance) {  
        this.accountHolder = accountHolder;  
        this.balance = balance;  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
  
    public void deposit(double amount) {  
        if (amount > 0) {  
            balance += amount;  
            System.out.println("Deposited: " + amount);  
        } else {
```

```
        System.out.println("Invalid deposit amount.");
    }
}
```

```
public void withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
        balance -= amount;
        System.out.println("Withdrawn: " + amount);
    } else {
        System.out.println("Invalid withdrawal amount or insufficient funds.");
    }
}
```

```
public void displayAccountInfo() {
    System.out.println("Account Holder: " + accountHolder + ", Balance: " + balance);
}
}
```

```
public class Encapsulation {
    public static void main(String[] args) {
        BankAccount myAccount = new BankAccount("John Doe", 5000);
        myAccount.displayAccountInfo();
        myAccount.deposit(1500);
        myAccount.withdraw(2000);
        myAccount.displayAccountInfo();
    }
}
```

Output:

```
PS C:\Users\Admin\Desktop\Java Assignment> javac Encapsulation.java
PS C:\Users\Admin\Desktop\Java Assignment> java Encapsulation
Account Holder: John Doe, Balance: 5000.0
Deposited: 1500.0
Withdrawn: 2000.0
Account Holder: John Doe, Balance: 4500.0
PS C:\Users\Admin\Desktop\Java Assignment> |
```

Encapsulation Benefits:

- **Security:** Prevents direct modification of balance.
- **Data Hiding:** The variables `accountHolder` and `balance` are **private** so they **cannot be accessed directly** from outside the class.
- **Controlled Access:** The methods `getBalance()`, `deposit()`, and `withdraw()` are **public** and allow access to modify and retrieve account balance **safely**.
- **Control:** Only valid operations (positive deposit, sufficient balance for withdrawal) are allowed.
- **Code Organization:** The `BankAccount` class keeps all related functions together.

Q3 Explain the concept of interfaces and abstract classes with examples.

Ans Interface: An interface is a completely abstract type that only contains abstract methods (before Java 8) and default/static methods (from Java 8 onwards). It enforces full abstraction, meaning classes that implement an interface must define all its methods.

- The interface in Java is a mechanism to achieve abstraction.
- By default, variables in an interface are public, static, and final.
- It is used to achieve abstraction and multiple inheritances in Java.
- It is also used to achieve loose coupling.
- In other words, interfaces primarily define methods that other classes must implement.
- An interface in Java defines a set of behaviours that a class can implement, usually representing an IS-A relationship, but not always in every scenario.
- We can create an interface using **interface** keyword.

Advantages of Interfaces

- Without bothering about the implementation part, we can achieve the security of the implementation.
- In Java, multiple inheritances are not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

Example:

```
interface Vehicle {  
    void start();  
    default void stop() {  
        System.out.println("Vehicle is stopping...");  
    }  
}  
  
class Car implements Vehicle {  
    @Override  
    public void start() {  
        System.out.println("Car engine started.");  
    }  
}
```

```

    }
}
class Bike implements Vehicle {
    @Override
    public void start() {
        System.out.println("Bike engine started.");
    }
}

```

```

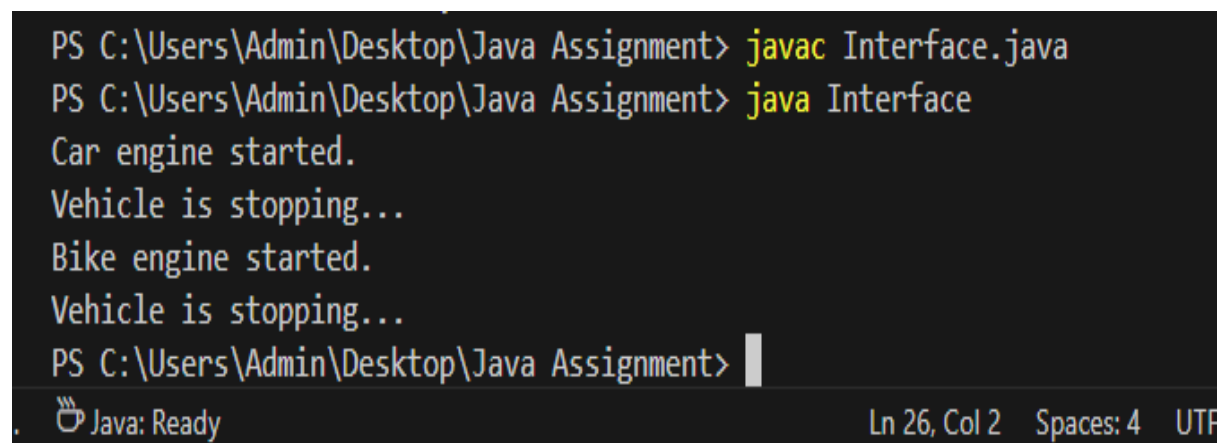
public class Interface {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        Vehicle myBike = new Bike();

        myCar.start();
        myCar.stop();

        myBike.start();
        myBike.stop();
    }
}

```

Output:



```

PS C:\Users\Admin\Desktop\Java Assignment> javac Interface.java
PS C:\Users\Admin\Desktop\Java Assignment> java Interface
Car engine started.
Vehicle is stopping...
Bike engine started.
Vehicle is stopping...
PS C:\Users\Admin\Desktop\Java Assignment>

```

Java: Ready Ln 26, Col 2 Spaces: 4 UTF

Abstract Class:

An abstract class is a class that cannot be instantiated and may contain both abstract methods (without a body) and concrete methods (with implementations). It provides partial implementation, meaning child classes must override abstract methods to define specific behavior. Java abstract class is a class that cannot be instantiated by itself, it needs to be subclassed by another class to use its properties. An abstract class is declared using the “abstract” keyword in its class definition.

Example:

```
abstract class Animal {  
    String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
    // Abstract method (must be implemented by subclasses)  
    abstract void makeSound();  
    // Concrete method (shared behavior)  
    public void sleep() {  
        System.out.println(name + " is sleeping.");  
    }  
}  
  
// Concrete subclass of Animal  
class Dog extends Animal {  
    public Dog(String name) {  
        super(name);  
    }  
    // Implementing abstract method  
    @Override  
    public void makeSound() {  
        System.out.println(name + " barks: Woof! Woof!");  
    }  
}
```

```
}  
}
```

// Concrete subclass of Animal

```
class Cat extends Animal {  
    public Cat(String name) {  
        super(name);  
    }  
}
```

// Implementing abstract method

@Override

```
public void makeSound() {  
    System.out.println(name + " meows: Meow! Meow!");  
}
```

```
}
```

```
public class AbstractClass {  
    public static void main(String[] args) {  
        Dog myDog = new Dog("Buddy");  
        Cat myCat = new Cat("Whiskers");  
  
        myDog.makeSound();  
        myCat.makeSound();  
  
        myDog.sleep();  
        myCat.sleep();  
    }  
}
```

Output:

```
Buddy barks: Woof! Woof!
Whiskers meows: Meow! Meow!
Buddy is sleeping.
Whiskers is sleeping.
PS C:\Users\Admin\Desktop\Ishikaaaaa>
```

Difference between Abstract Class and Interface:

Parameter	Abstract Class	Interface
Method Type	Can have both abstract and concrete methods	Can only have abstract methods (before Java 8), but default/static methods are allowed from Java 8+
Method Implementation	Can provide partial implementation	Cannot provide implementation (except for default/static methods in Java 8+)
Variables	Can have instance variables (with any access modifier)	Can only have public, static, and final (constants) variables
Constructors	Can have constructors	Cannot have constructors
Multiple Inheritance	Does not support multiple inheritance	Supports multiple inheritance (a class can implement multiple interfaces)
Access Modifiers	Methods can have public, protected, or private access	All methods are public and abstract by default
Usage	Used when a class needs some common behavior with flexibility for subclasses	Used when different classes need to follow the same contract

Q4 Explore Multithreading in Java to perform multiple tasks concurrently.

Ans Multithreading is a technique in Java that allows multiple tasks (threads) to run simultaneously within a program. It improves performance by utilizing CPU efficiently, making applications more responsive and faster.

Key Concepts

- **Thread:** The smallest unit of execution in a program.
- **Multithreading:** Running multiple threads in parallel.
- **Concurrency:** Executing multiple tasks at the same time.
- **Parallelism:** Executing tasks on multiple CPU cores simultaneously.

Java provides two ways to create threads:

1. **Extending Thread class**
2. **Implementing Runnable interface**

Creating a Thread using Thread Class: In this method, we extend the Thread class and override the run() method.

// Extending the Thread class

```
class Task extends Thread {  
    private final String taskName;  
  
    public Task(String taskName) {  
        this.taskName = taskName;  
    }  
  
    @Override  
    public void run() {  
        for (int i = 1; i <= 5; i++) {  
            System.out.println(taskName + " - Count: " + i);  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException e) {
```

```

        System.out.println("Thread interrupted.");
    }
}
}
}

```

```

public class Multithreading {
    public static void main(String[] args) {
        Task task1 = new Task("Thread 1");
        Task task2 = new Task("Thread 2");

        task1.start();
        task2.start();
    }
}

```

```

Thread 1 - Count: 1
Thread 2 - Count: 1
Thread 2 - Count: 2
Thread 1 - Count: 2
Thread 2 - Count: 3
Thread 1 - Count: 3
Thread 2 - Count: 4
Thread 1 - Count: 4
Thread 2 - Count: 5
Thread 1 - Count: 5
PS C:\Users\Admin\Desktop\Java Assignment>

```

Creating a Thread using Runnable Interface

The Runnable interface allows multiple classes to share the same interface while still achieving multithreading.

// Implementing Runnable interface

```

class MyRunnable implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(taskName + " - Count: " + i)

```

```

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}
}

public class RunnableExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        Thread t2 = new Thread(new MyRunnable());
        t1.start();
        t2.start();
    }
}

```

Output:

```

Thread 1 - Count: 1
Thread 2 - Count: 1
Thread 2 - Count: 2
Thread 1 - Count: 2
Thread 2 - Count: 3
Thread 1 - Count: 3
Thread 2 - Count: 4
Thread 1 - Count: 4
Thread 2 - Count: 5
Thread 1 - Count: 5
PS C:\Users\Admin\Desktop\Java Assignment>

```

Runnable is preferred because Java allows extending only one class, but multiple interfaces can be implemented.

