Name: Nusrat Jahan Shamima
ID : IT-22017

1. When to use Interface and when to Abstract class. Develop a story and write codes to explain.

→ Story:

In the magical Kingdom of Creators, every inhabitant can create something - some create art, some music, some software, and some do multiple things.

The Queen wants to build a system to track all creators and their capabilities.

She calls ~~to~~ upon two wise engineers:

Engineer Abstracto (Abstract Class)

Abstracto says:

"All creators have names, and they must register and display their portfolio. These are common behaviors, so let's define a base abstract class 'Creator'.

Engineer Interfacia (Interface)

Interfacia adds:

But wait! Some creators might:

- Draw (Drawable)
- Code (Codeable)
- Sing (Singable)

These are skills, and creators can have multiple.
Let's define these as interfaces!"

The Queen agrees. Some creators will inherit from the base creator class, and implement multiple interfaces depending on their skills.

Code:

```
abstract class Creator {
        string name;
    Creator (string name){
        this.name = name; }
  abstract void register();
  abstract void showPortfolio();
 void intro() {
     System.out.println ("Hi, I'm " +name ", a creator!")
   } }
interface Drawable {
        void draw();
    }.
interface Singable {
        void sing();
   }
```

```
class Artist extends creator implements Drawable {
   Artist (string name) {
            super (name) ; }
  public void draw () {
            system.out.println (name + " is drawing.");
            }
void register () {
     System. out. println (name + " registered as Artist.");
     }
void showPortfolio () {
     System.out.println (name + " 's Art Portfolio.");
    }
  }

class Rockstar extends Creator implements Drawable, Singable {
    Rockstar ( string name) {
                super (name); }
public void draw () {
     System.out.println (name + " is drawing album art.");
          }
public void sing () {.
          System. out. println (name + " is singing live."); }
void register() {
    System.out.println (name + " registered as Rockstar");}
void showPortfolio () {
   System.out.println (name + "'s Music & Art Portfolio.");}
      }
public class Main {
  public static void main (String [] args) {
    Creator artist = new Artist ("Amiya");
    artist.intro() ;
    artist. register() ;
   ((Drawable) artist). draw();
   artist. showPortfolio();
 System.out.println ();
Creator rockstar = new Rockstar ("Rafi");
rockstar. intro() ;
rockstar. register ();
((Drawable) rockstar ). draw();
((singable) rockstar ). sing();
rockstar . showPortfolio () ;
 } }
```

[2] ~~Is it~~

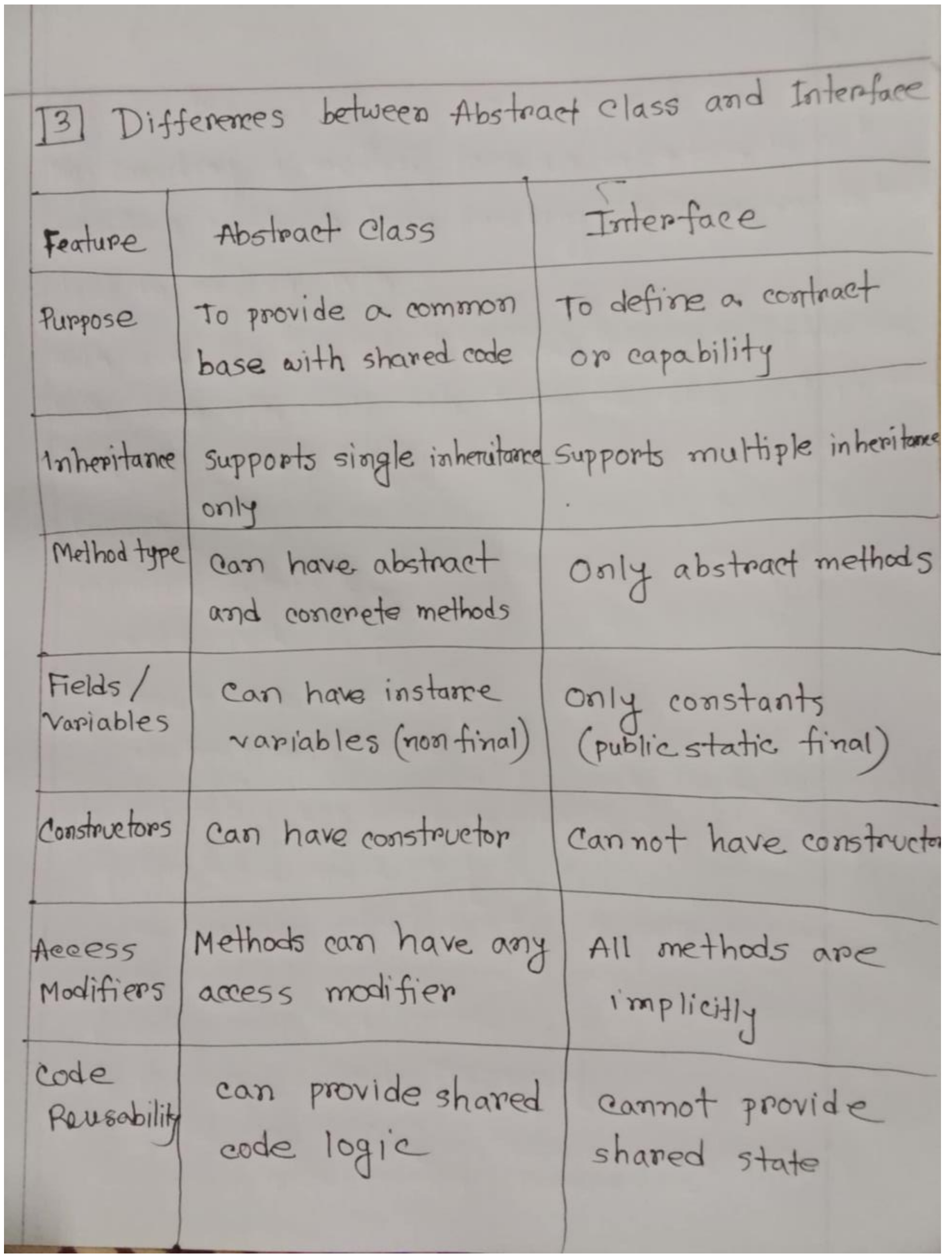
No, invoking a method from an interface is not significantly slower than invoking it from an abstract class in modern JVMs.

Both use dynamic dispatch, and the JVM optimizes both cases efficiently with technique like just-in-time compilation and inlining.

Example:

```
interface Speakable {
    void speak (); }
abstract class Human {
    abstract void speak(); }
class Robot implements Speakable {
  public void speak () {
    System.out.println ("Robot says: Hello!");
    }
 }
class Person extends Human {
    void speak() {
      System.out.println ("Person says: Hi!");
    } }
```

```
public class Main {
 public static void main (String [] args) {
  Speakable robot = new Robot();
  Human person = new Person();
  long start1 = System.nanotime() ;
  robot.speak();
  long ~~star~~ end1 = System.nanotime();
  long start 2 = System.nanoTime();
  person.speak();
  long end2 = System.nanoTime() ;
 System.out.println ("Interface call time : "
                        + (end1 - start1));
 System.out.println ("Abstract class call time: "
                        + (end2 - start 2));
    }
  }
```



[3] Differences between Abstract class and Interface

Feature	Abstract Class	Interface
Purpose	To provide a common base with shared code	To define a contract or capability
Inheritance	Supports single inheritance only	Supports multiple inheritance
Method type	Can have abstract and concrete methods	Only abstract methods
Fields / Variables	Can have instance variables (non final)	Only constants (public static final)
Constructors	Can have constructor	Can not have constructo
Access Modifiers	Methods can have any access modifier	All methods are implicitly
Code Reusability	can provide shared code logic	cannot provide shared state