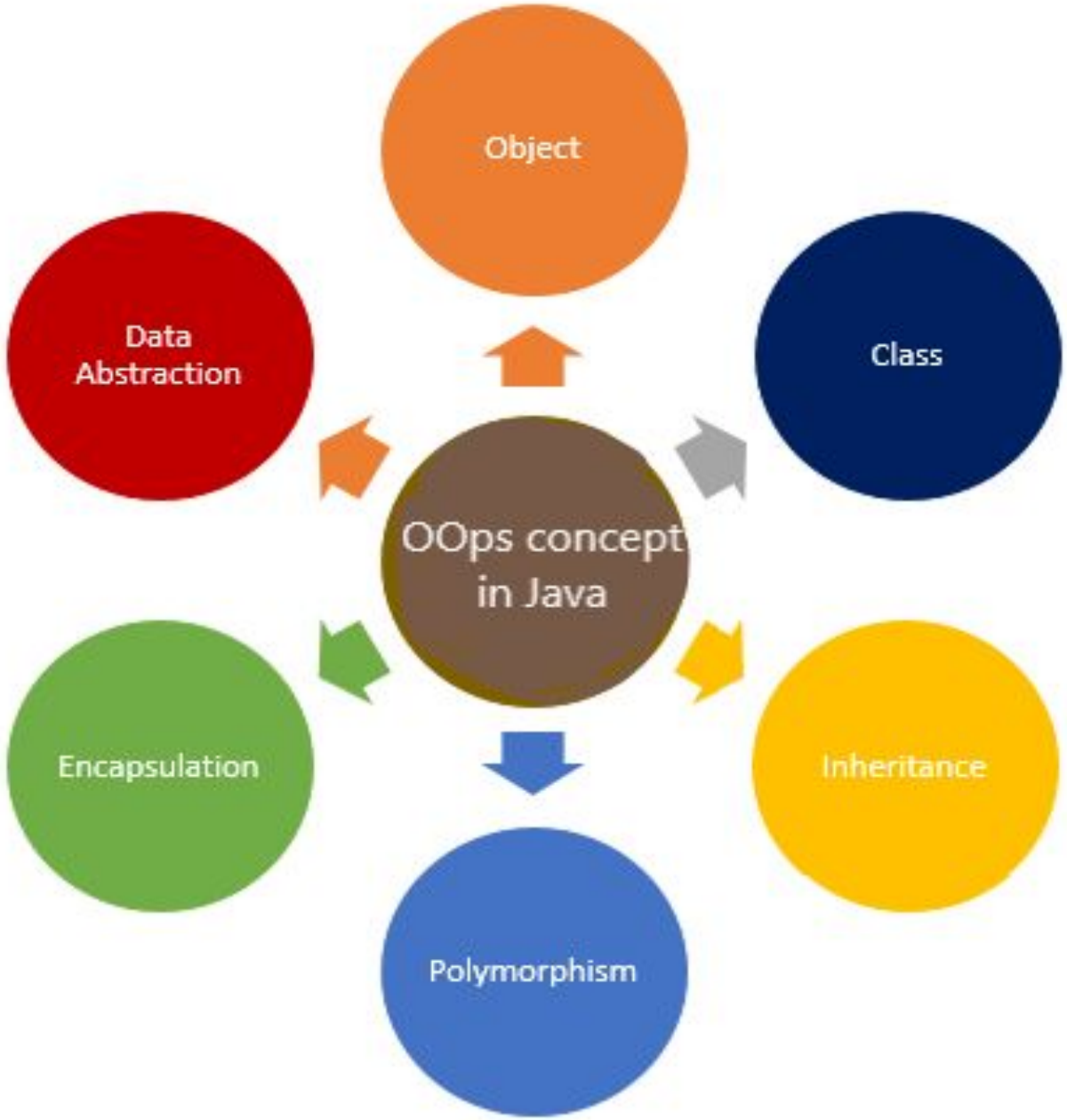
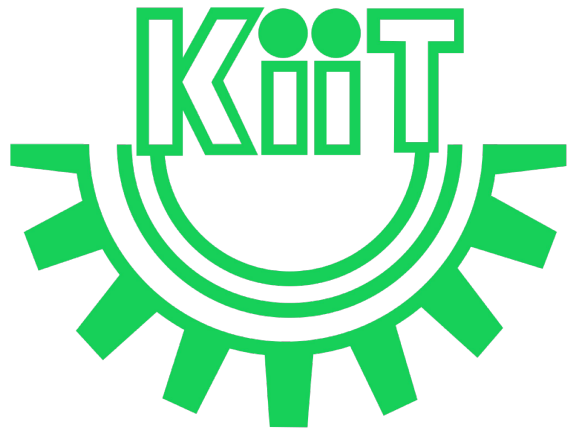


CS20004: Object Oriented Programming using Java

Lec-17

In this Discussion . . .

- Interfaces
  - Nested Interfaces
- Dynamic method lookup
- References



Interface in Java

keyboard interface glide class ways since angry
another certain class cake class java new
declared implementations java since new since will
baker new glide keyboard
extend java new angry new high glide certain using class
methods
since java glide example interface multiple
able class interface extends bird glide
assume glide class canary provide java

Nested Interface

- An interface, that is declared within another interface or class, is known as a nested interface.
- The nested interfaces are used to group related interfaces so that they can be easy to maintain.
- The nested interface must be referred to by the outer interface or class. It can't be accessed directly.
- Nested interfaces are declared static.

Important Points considering Nested Interfaces

- To declare a nested interface there are certain rules. These rules apply depending upon where the nested interface declaration occurs inside another interface or class.
    - Nested interfaces are implicitly static regardless of where they are declared (inside class or another interface).
    - Nested interface declared inside another interface is implicitly public.
    - Nested interface declared inside a class can accept any access modifiers.
    - Nested interface can be implemented by any class (package level, nested or inner) if the access modifiers permit visibility.

Syntax

Syntax of nested interface which is declared within the interface	**Syntax of nested interface which is declared within the class**

interface interface_name
{
…
interface nested_interface_name
{
…
}
}
	class class_name
	{
	…
	interface nested_interface_name
	{
	…
	}
	}

Example:- Nested Interface within another Interface

```
interface Showable
{
	void show();
	interface Message
	{
		void msg();
	}
}
class TestNestedInterface1 implements Showable.Message
{
	public void msg()
	{
		System.out.println("Inside the nested interface");
	}
	public static void main(String args[])
	{
		Showable.Message message=new TestNestedInterface1();//upcasting here as we are accessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like the almirah inside the room; we cannot access the almirah directly because we must enter the room first.
		message.msg();
	}
}
```

Example:- Nested Interface within another Interface

- As said earlier, when an interface is declared within another interface it is implicitly public and static.
- **Therefore, adding those modifiers is considered redundant and makes no difference.**
- Given that rule, we can see that the following two nested interface declarations are exactly equivalent:

```
interface A
{
  interface NestedA { void aMethod(); }
  interface NestedAA { void aaMethod(); }
}
```

Example:- Nested Interface within another Interface

- Above nested interface declaration for **NestedA** and **NestedAA** is equivalent to the following declaration because modifiers public and static are implicit.
- However, the compiler would not complain if an implicit modifier is mentioned explicitly by programmers but this is not considered as a good practice and you should avoid doing that.
- And so, if a modifier comes more than one time in a declaration, it would be a compile time error.

```
interface A
{
  public static interface NestedA { void aMethod(); }
  public static interface NestedAA { void aaMethod(); }
}
```

Example:- Nested Interface within another Interface: Saving, Compilation, and Running

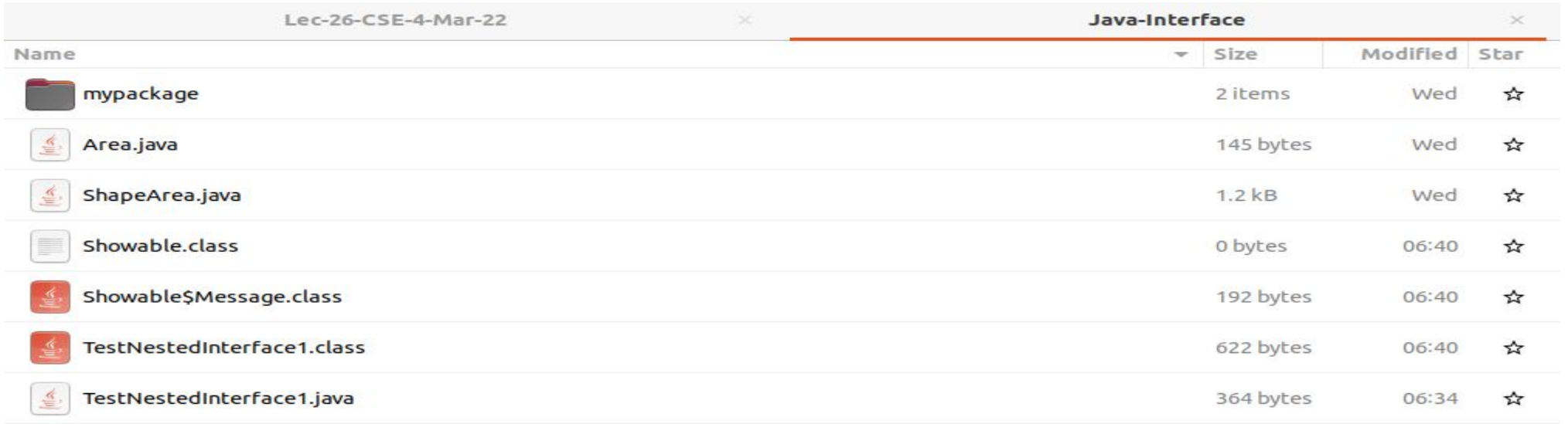
Save the file as: **TestNestedInterface1.java**

Notice:- Here, both the outer and inner interfaces are present saved in the same file with the name of the file equivalent to the java class with the main method.

Example:- Nested Interface within another Interface: Saving, Compilation, and Running

Compiling the file as: javac **TestNestedInterface1.java**

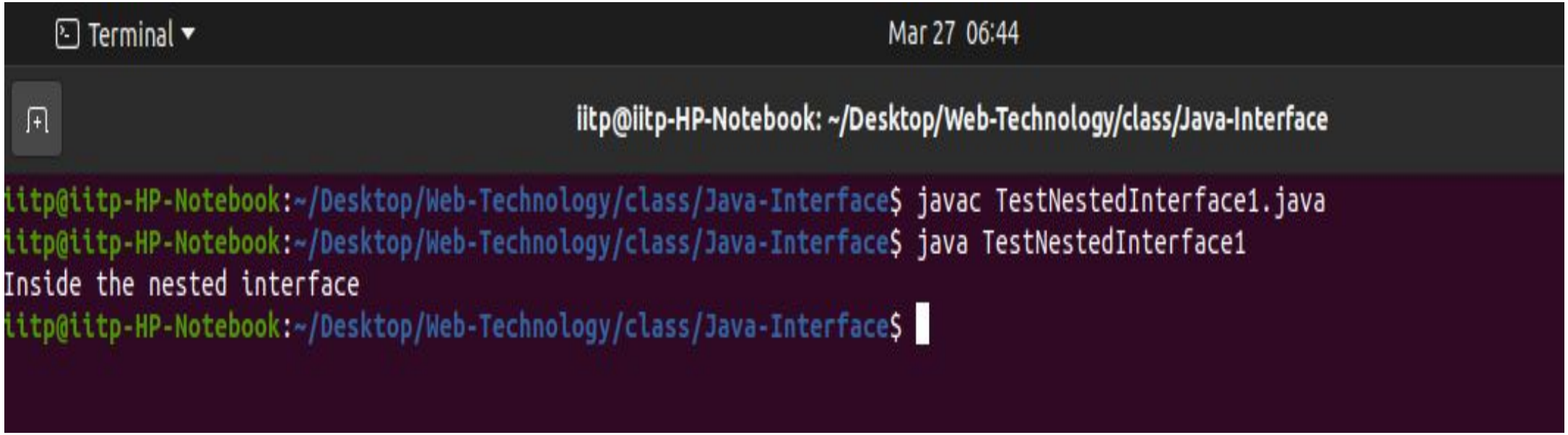
Notice:- On compiling the above file, we are able to get the class files for both the parent and nested interface, along with the class with main method



Example:- Nested Interface within another Interface: Saving, Compilation, and Running

Run the file using: java **TestNestedInterface1**

Notice:- We do not require to run the classes of the generated interfaces, but instead only of the class which uses the interfaces.



Internal code generated by the java compiler for nested interface Message

- The java compiler internally creates a public and static interface as displayed below:

```
public static interface Showable$Message
{
  public abstract void msg();
}
```

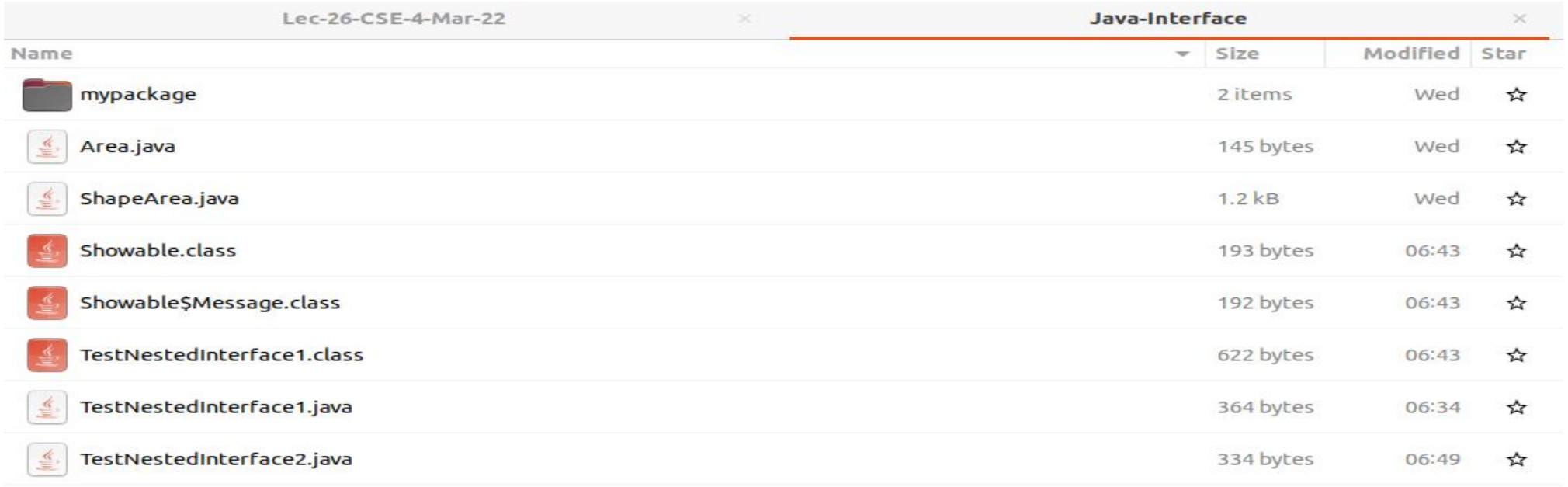
Example:- Nested Interface within a Class

```
class A
{
	interface Message
	{
		void msg();
	}
}

class TestNestedInterface2 implements A.Message
{
	public void msg()
	{
		System.out.println("Hello nested interface");
	}

	public static void main(String args[])
	{
		A.Message message=new TestNestedInterface2();//upcasting here
		message.msg();
	}
}
```

Example:- Nested Interface within another Class: Saving, Compilation, and Running

Save the file as: **TestNestedInterface2.java**

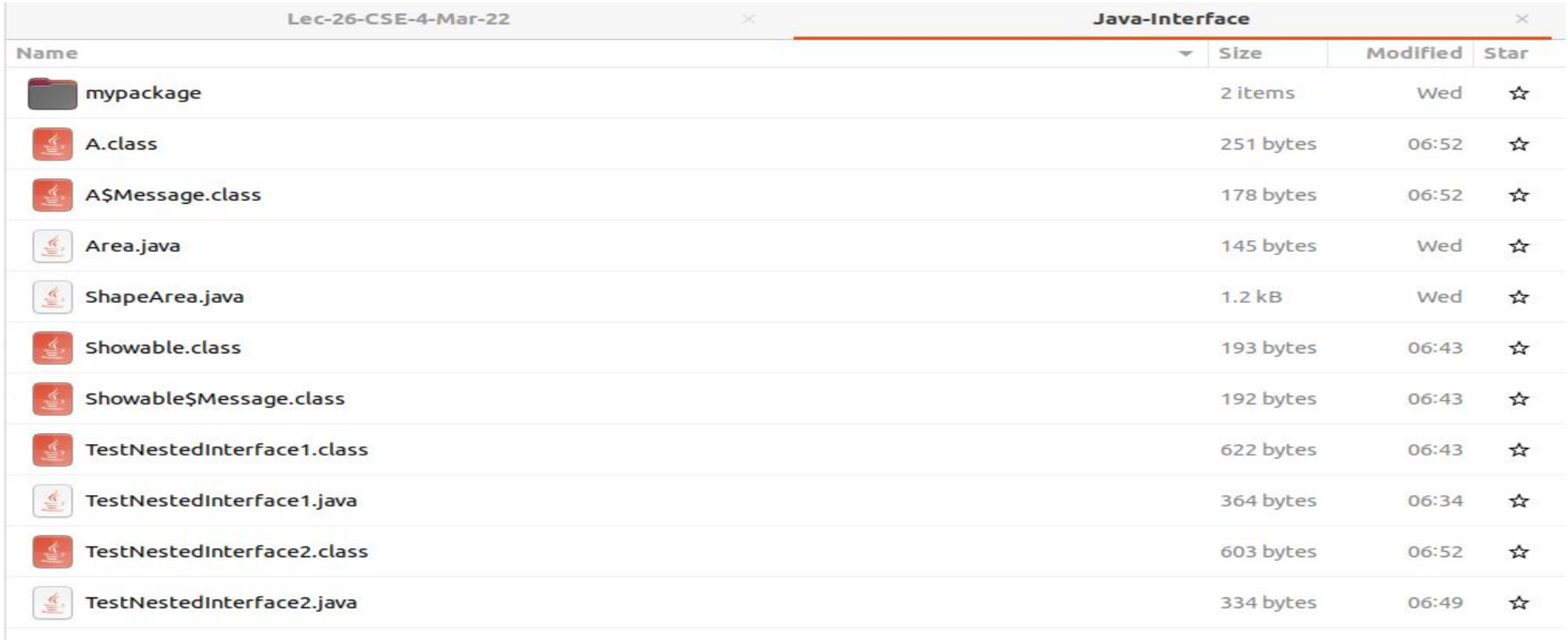
Notice:- Here, both the class along with its inner interfaces are present and saved in the same file with the name of the file equivalent to the java class with the main method.



Example:- Nested Interface within another Class: Saving, Compilation, and Running

Compiling the file as: javac **TestNestedInterface2.java**

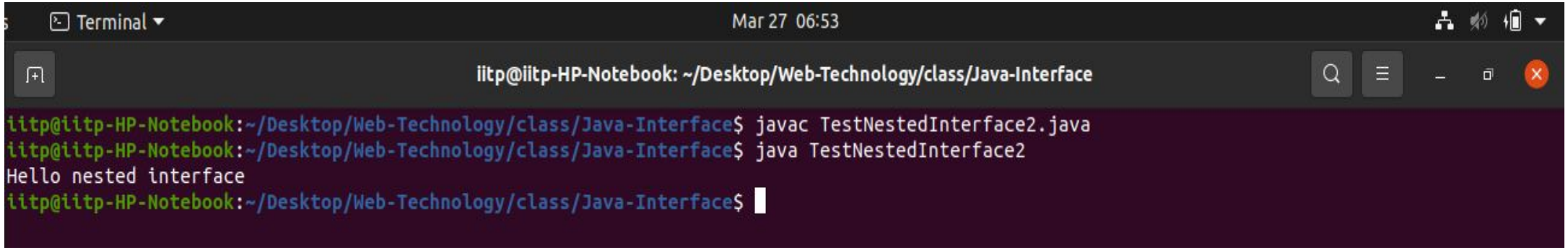
Notice:- On compiling the above file, we are able to get the class files for both the parent class and nested interface, along with the class with main method



Example:- Nested Interface within another Class: Saving, Compilation, and Running

Run the file using: java **TestNestedInterface2**

Notice:- We do not require to run the classes of the generated interfaces or its parent class, but instead only of the class with main method



Can we define a class inside the interface?

- Yes, if we define a class inside the interface, the Java compiler creates a static nested class.

```
interface M{
  class A
  {

   }
}
```

Dynamic Method Lookup

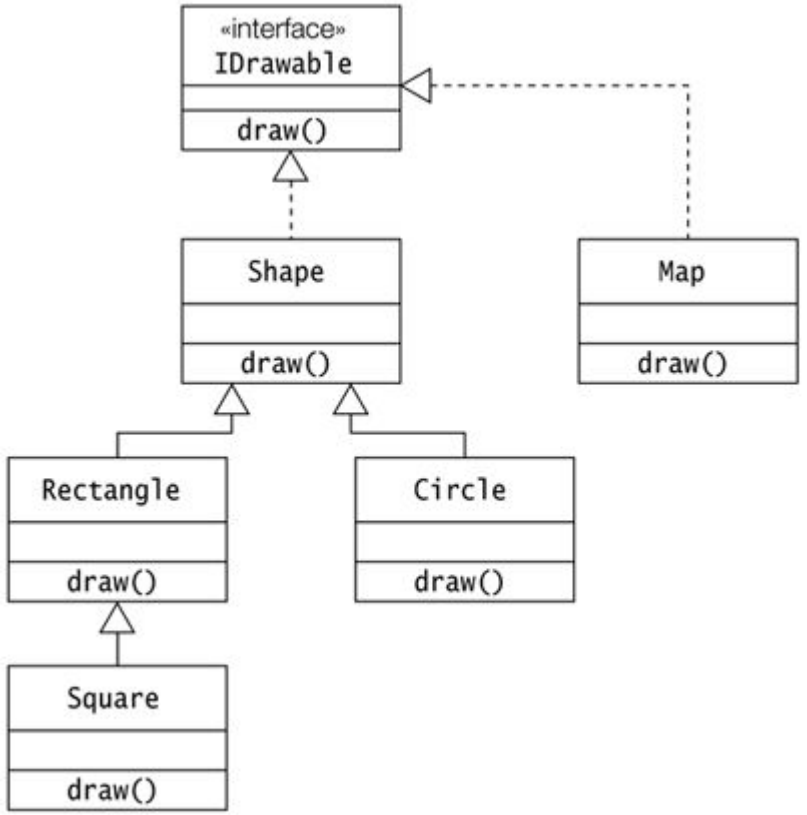
- Which object a reference will actually denote during runtime, cannot always be determined at compile time.
- Polymorphism allows a reference to denote objects of different types at different times during execution.
- A supertype reference exhibits polymorphic behavior, since it can denote objects of its subtypes.

Dynamic Method Lookup

- When a non-private instance method is invoked on an object, the method definition actually executed is determined both by the type of the object at runtime and the method signature.
- Dynamic method lookup is the process of determining which method definition a method signature denotes during runtime, based on the type of the object.
- However, a call to a private instance method is not polymorphic. Such a call can only occur within the class, and gets bound to the private method implementation at compile time.

Dynamic Method Lookup

- The inheritance hierarchy depicted below is implemented in upcoming slides.



Example of Polymorphism & Dynamic Method Lookup

```
interface IDrawable {
    void draw();
}

class Shape implements IDrawable {
    public void draw() { System.out.println("Drawing a Shape.");
}
}

class Circle extends Shape {
    public void draw() { System.out.println("Drawing a Circle."); }
}

class Rectangle extends Shape {
    public void draw() { System.out.println("Drawing a
Rectangle."); }
}
```

```
class Square extends Rectangle {
    public void draw() { System.out.println("Drawing a Square.");
}
}

class Map implements IDrawable {
    public void draw() { System.out.println("Drawing a Map."); }
}

public class PolymorphRefs {
    public static void main(String[] args) {
        Shape[] shapes = {new Circle(), new Rectangle(), new
Square()};   // (1)
        IDrawable[] drawables = {new Shape(), new Rectangle(),
new Map()};// (2)

        System.out.println("Draw shapes:");
        for (int i = 0; i < shapes.length; i++)                    // (3)
            shapes[i].draw();

        System.out.println("Draw drawables:");
        for (int i = 0; i < drawables.length; i++)                  // (4)
            drawables[i].draw();
    }
}
```

Dynamic Method Lookup

- The implementation of the method draw() is overridden in all subclasses of the class Shape.
- The invocation of the draw() method in the two loops at (3) and (4) in the previous slide, relies on the polymorphic behavior of references and dynamic method lookup.

Dynamic Method Lookup

- The array shapes holds Shape references denoting a Circle, a Rectangle and a Square, as shown at (1). At runtime, dynamic lookup determines the draw() implementation to execute, based on the type of the object denoted by each element in the array.
- This is also the case for the elements of the array drawables at (2), which holds IDrawable references that can be assigned any object of a class that implements the IDrawable interface.

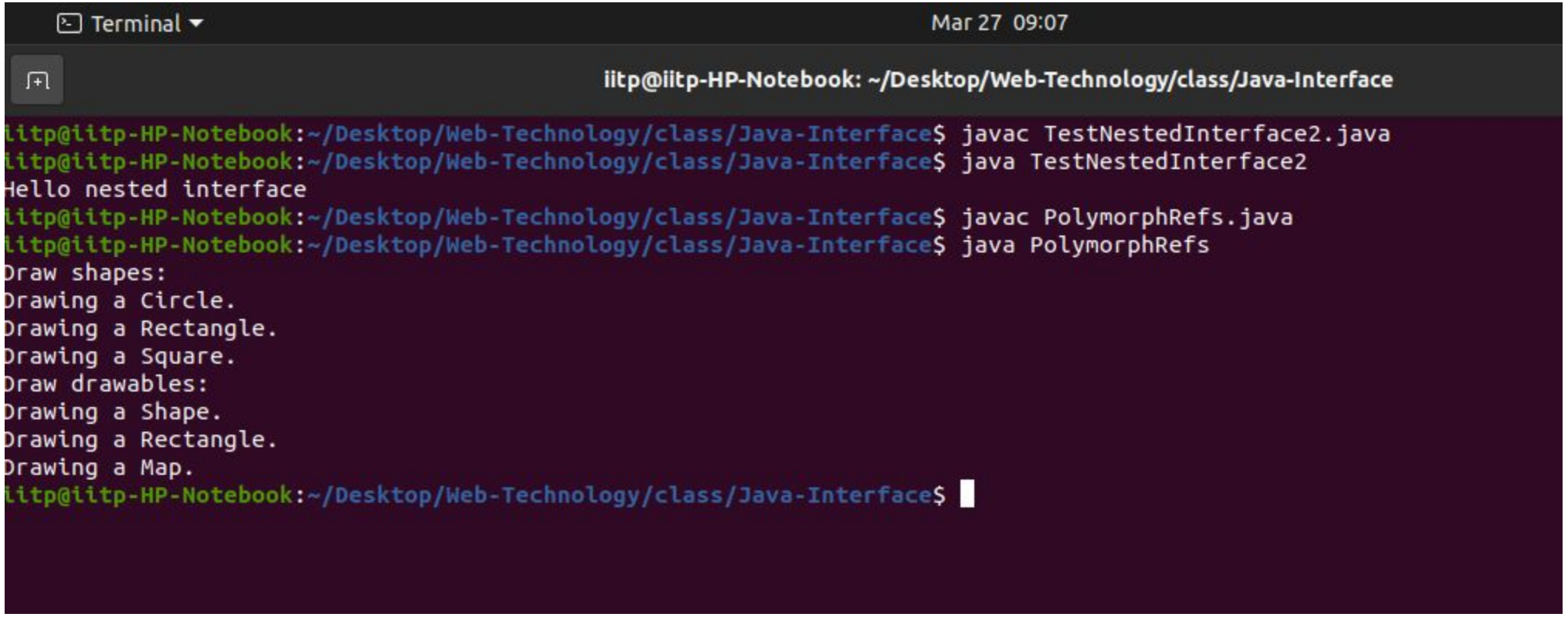
Dynamic Method Lookup

- The first loop will still work without any change if objects of new subclasses of the class Shape are added to the array shapes.
- If they did not override the draw() method, then an inherited version of the method would be executed.
- This polymorphic behavior applies to the array drawables, where the subtype objects are guaranteed to have implemented the IDrawable interface.

Dynamic Method Lookup

- Polymorphism and dynamic method lookup form a powerful programming paradigm that simplifies client definitions, encourages object decoupling, and supports dynamically changing relationships between objects at runtime.

Example of Polymorphism & Dynamic Method Lookup



References

1. https://www.javatpoint.com/nested-interface
2. http://etutorials.org/cert/java+certification/Chapter+6.+Object-oriented+Programming/6.7+Polymorphism+and+Dynamic+Method+Lookup/#:~:text=Dynamic%20method%20lookup%20is%20the,instance%20method%20is%20not%20polymorphic.
3. http://coding-guru.com/polymorphism-java/
4. https://coderanch.com/t/378538/java/Dynamic-method-lookup
5. https://www.baeldung.com/java-inner-interfaces
6. https://cs-fundamentals.com/java-programming/java-static-nested-or-inner-interfaces