IT 22038

Story: "The Tech Company"

I am a lead software architect at a Tech company. The company has different types of employees.

- Developers
- Designers
- Managers

Everyone in the company is an Employee. They all:

- Have a name and ID.
- Must perform work().
- Can optionally attend meetings.
- Some can code, some can design, and some can manage project.

So:

i) Developer: codes
ii) Designer: designs
iii) Manager: manages

1.10.0

IT-22088 | L

Goal: Model the system so that:

- Common properties like name, id, and work() are shared
- Specialized capabilities like Codeable, Designer, Manageable are added when needed.

☐ Use Abstract class:

- For Employee - since all employees share fields like name and ID, and maybe a default attendMeeting() method.

☐ Use Interfaces:

- For codeable, Designable, Manageable - these are abilities that can be mixed into different employees.

IT22058

1. Capability Interfaces

```
interface Codeable {
    void code(); }
interface Designable {
    void design(); }
interface Manageable {
    void manage(); }
```

2. Abstract Employee Class

```
abstract class Employee {
    String name;
    int id;
Employee (String name, int id) {
    this.name = name;
    this.id = id; }
```

```
abstract void work();
void attendMeeting() {
System.out.println(name + " is attending a
                    meeting."); }
```

3. Specific Employees.

```
class Developer extends Employee implements
 Codeable {
 Developer(String name, int id);
 }
 public void work() {
System.out.println(name + " is working on
                    backend development."); }

 public void code()
        {
        System.out.println(name + " is willing
                            Java code."); }
```

IT-22038

```
class Designer extends Employee implement Designable{
    Designer (String name, int id){
        super (name, id); }
    public void work (){
        System.out.println (name + " is preparing ur
                            mockups."); }
    public void design (){
        System.out.println (name + " is designing a
                            new user interface."); } }
Class Manager extends Employee implement Mangeable{
    Manager ( String name, int id){
        Super (name, id); }
    public void work (){
        system.out.println (name + " is planning prior
                            timelines"); }
```

```
public void manage() {
    System.out.println(name + " is managing a
                        team of " + 10.");
}}

Public class Company {
{ public static void main (String[] args){
Employee dev = new Developer("Ebrahim", 101);
Employee designer = new Designer ("Tushar", 102);
Employee Manager = new Manager ("Charlie", 103);

Employee [] employees = {dev, designer, manager};
for (Employee e: employees) {
    e.displayInfo();
    e.attendMeeting();
    e.work();
    if (e instanceof Codeable) {
        ((Codeable)e).code(); }
```

IT-22038

```
if (e instance Designable) {
    ((Designable e). design();}
if (e instanceof Manageable) {
    ((Manageable e). manage();}
System.out.println();}}}
```

IT-22038

2 Is it true that invoking method in interface are slower than invoking it within the abstract classes. Explain and write a new example.

⇒ Yes, Modern Java optimizes both Interface and abstract class method calls are equally fast in real world scenarios.

Example:

```
public class MethodCallsSpeedTest {
public static void main (String [] args) {
    final int Iterations = 100_000_000;

    // Interface version
    Doable doer = new InterfaceImpl ();
    long startInterface = System.nanoTime();
    for (int i = 0; i < iterations; i++) {
        doer.doSomething (); }
```

```
long endInterface = System.nanoTime();
System.out.println("Interface time" + (endInterface -
        startInterface) / 1000000.0 + "
                ms");

// Abstract class version
Base absDoer = new AbstractImpl();
long startAbstract = System.nanoTime();
for (int i=0 ; i< iterations ; i++) {
  absDoer : doSomething();
}
long endAbstract = System.nanoTime()
System out.println("Abstract class time: " +
        (endAbstract - startAbstract));

// Interface based
interface Double {
    void doSomething(); }
```

```
class InterfaceImpl implements Double {
    public void doSomething () {
        int x = 1+1; } }

// Abstract class Based

abstract class Base {
    abstract void doSomething (); }

class AbstractImp extends Base {
    public void doSomething () {
        int x = 1+1; } }
```

3

Feature	Abstract class	Interface
Purpose	Provides a base for inheritance	Define a constract or capability
Keyword	abstract	interface
Access Modifiers	Private, protected, Public	Methods are Public by default
Fields	Can have instance variable	Only constants
Constructor	Yes	No