**day22\_107856406\_dsdipt\_sudipto\_30july2025**

**Employee Code:** 107856406

**Login ID:** dsdipt

**Email :** dsdipt@amazon.com

**Name:** Sudipto Das

**Date:** 30 July 2025 (Day 22)

### ***Task 1: What is Good Code vs Bad Code?***

**🟢 Good Code**

* **Readable**
* **Maintainable**
* **Modular**
* **Testable**
* **Performant**
* **Scalable**
* **DRY** – Don’t Repeat Yourself. No copy-pasta code spam.

Clean code is code that humans can read, understand, and change.

**🔴 Bad Code**

* **Hardcoded Junk** – Magic numbers, hardcoded values all over.
* **No Naming Sense** – a, b1, foo123 — what do they even mean??
* **No Comments** – Or worse, useless comments: // increment i by 1
* **Tight Coupling** – Everything depends on everything. One tweak? 💥

### ***Task 2: What do you understand by databinding?***

**🔗 What is Data Binding?**

**⚙️ Types of Data Binding:**

**🟢 1. One-way Binding (most common in React):**

* Data flows **one direction**: From the **code → UI**
* UI reflects changes in the code, but not the other way automatically.

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

If name = "eZcoDiN" in code, UI shows "Hello, eZcoDiN"

**🔄 2. Two-way Binding (used in frameworks like Angular or with useState in React):**

* Data flows **both ways**: From code → UI and UI → code
* Useful for forms and user input.

const [name, setName] = useState("");

<input

type="text"

value={name}

onChange={(e) => setName(e.target.value)}

/>

You type in the input field — name updates in real-time. You update name in code — input field updates too. *Synced vibes* 🔁

**🔴 3. No Binding (manual DOM update era 😵):**

Before frameworks, devs had to manually update the DOM when data changed.

**Why It’s Important:**

* Simplifies UI updates
* Reduces boilerplate
* Keeps UI and logic in sync
* Makes forms easier to manage
* Makes your app feel **live** & **dynamic**

### ***Task 3: What do you know about Continuous Development?***

**Continuous Development** is the umbrella term for **constantly improving, testing, and deploying code** — **without stopping the world**.

**🔁 It’s a whole *DevOps loop* — here’s the core flavors:**

1. **Continuous Integration (CI)** Merging code frequently, testing every commit  
    🧪 = "Code gets tested & built *automatically* after every push"
2. **Continuous Delivery (CD)** Pushing tested code to **staging** automatically  
    = "Always ready to go live"
3. **Continuous Deployment (also CD)** Automatically pushing to **production** without manual approval  
    = "Code hits real users as soon as it passes tests"

**Example:**

You push your Spring Boot backend to GitHub →  
 GitHub Actions runs tests →  
 If green → Auto deploy to your cloud (like Heroku, Vercel, or AWS).

**Advantages:**

* Catches bugs early
* Speeds up releases
* Devs get feedback fast
* Less deployment stress

### ***Task 4: What are the conditions for Polymorphism?***

"**Poly**" = many, "**morph**" = forms  
 So... **polymorphism** = **one thing, many forms**

It’s a **core OOP concept** where objects can **take different behaviors depending on the context**.

**2 Types of Polymorphism in Java:**

**1. Compile-Time (Static)**

* **Method Overloading**: Same method name, diff parameters

void show(int a)

void show(String s)

**Conditions:**

* Same method name
* Different parameters
* Resolved at compile time

**2. Run-Time (Dynamic)**

* **Method Overriding**: Subclass provides specific implementation

class Animal { void sound() { } }

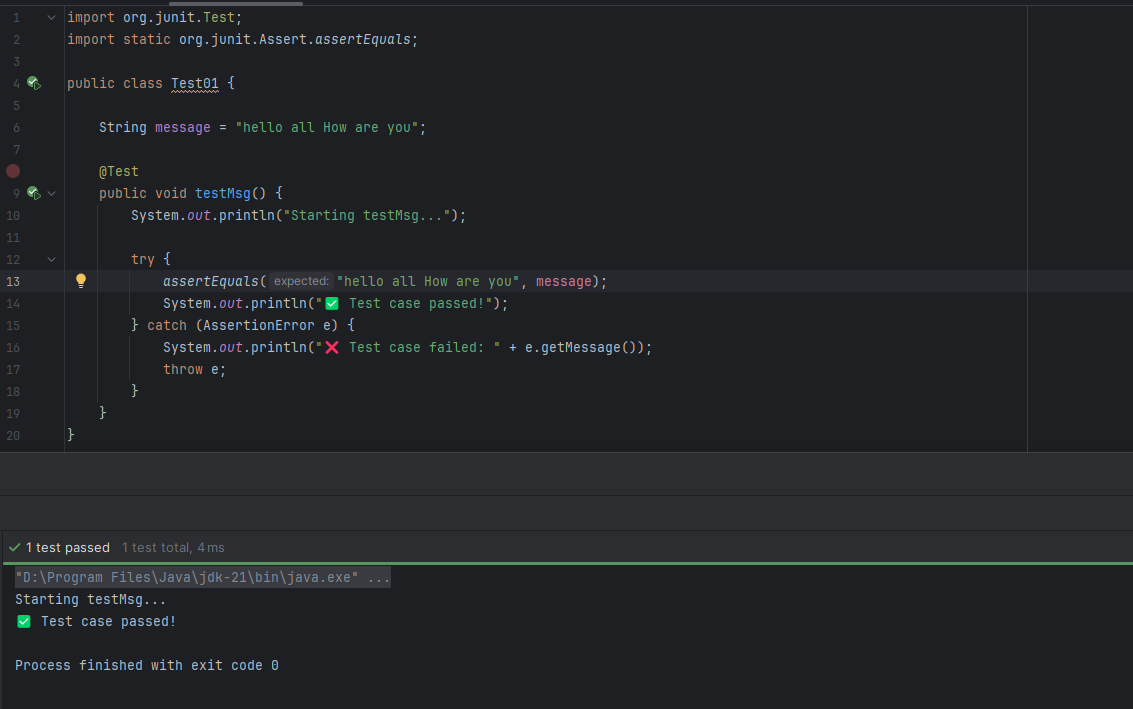
class Dog extends Animal { void sound() { System.out.println("Bark"); } }

**Conditions:**

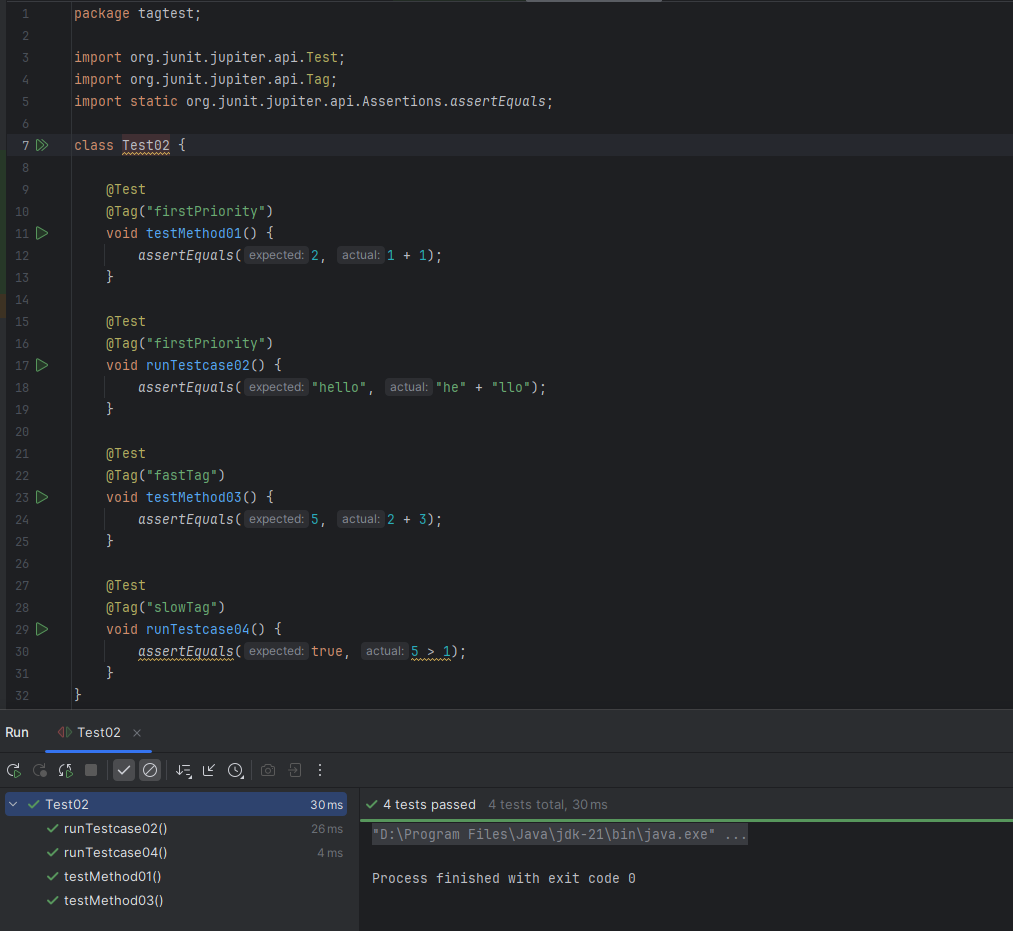
* Inheritance
* Method in child overrides parent
* Object is accessed via parent reference

Animal obj = new Dog(); // runtime decides which `sound()` to run

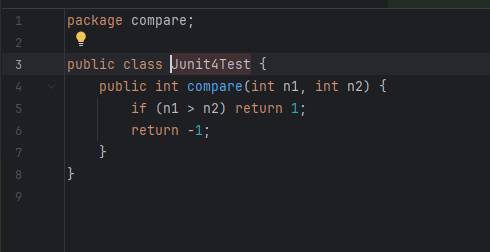
### ***Task 5: JUNIT Code 1***

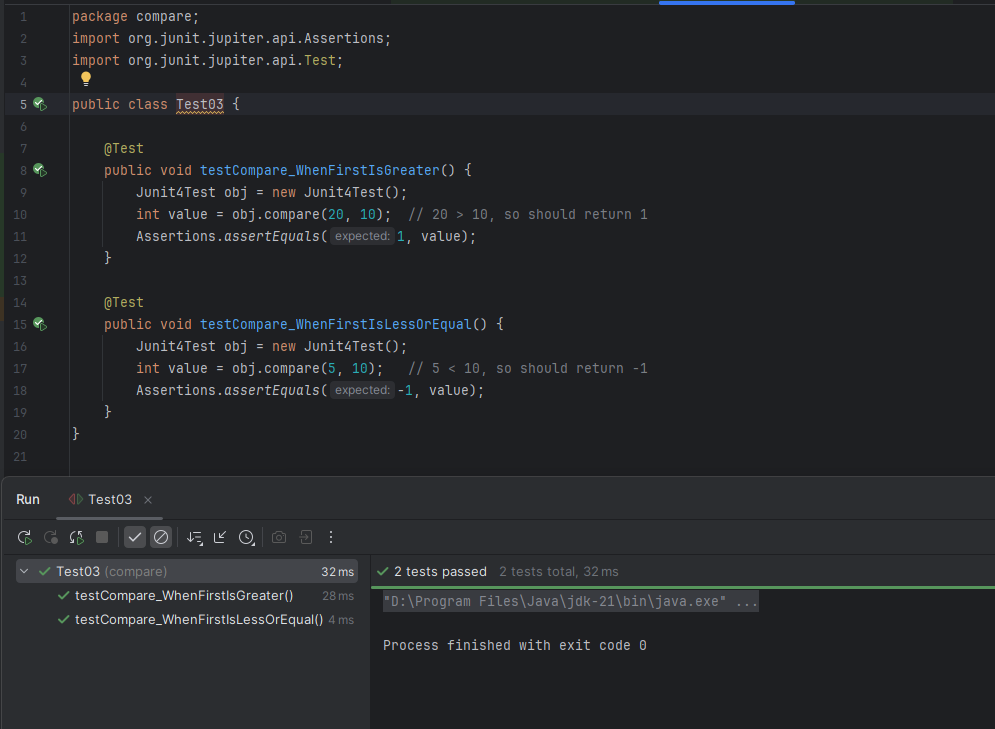


### ***Task 6: JUNIT Code 2 @TAG***



### ***Task 7: JUNIT Code 3 compare***





### ***Task 8: JUNIT Code 4 compare***

