**Sealed Classses - Java 17**

Sealed Classes - Java 17

Remember when we used to create class hierarchies in Java, and anyone could come along and extend our classes willy-nilly? It was like throwing a party and not knowing who might show up. Java 17 introduces Sealed Classes, giving us more control over our class hierarchies.

Sealed Classes: The Bouncer of the Java World

Sealed Classes are like having a guest list for your class hierarchy party. You get to decide exactly who's allowed to extend your class. Let's see how it works:

1. public sealed class Shape
2. permits Circle, Square, Triangle {
3. // Shape implementation
4. }
6. public final class Circle extends Shape {
7. // Circle implementation
8. }
10. public final class Square extends Shape {
11. // Square implementation
12. }
14. public final class Triangle extends Shape {
15. // Triangle implementation
16. }

In this example, Shape is our sealed class. It's explicitly stating, "Only Circle, Square, and Triangle are allowed to extend me. Everyone else, sorry, you're not on the list."

Breaking It Down

Let's look at the key components:

1. The sealed keyword: This is how we declare a sealed class.
2. The permits clause: This is where we list all the classes allowed to extend our sealed class.
3. The subclasses: Notice they're all marked as final. This isn't mandatory, but it's a common pattern to prevent further extension.

But Wait, There's More: Sealed Interfaces

Interfaces can join the sealed party too! Here's how it looks:

1. public sealed interface Vehicle
2. permits Car, Truck, Motorcycle {
3. // Vehicle interface methods
4. }
6. public final class Car implements Vehicle {
7. // Car implementation
8. }
10. // ... and so on for Truck and Motorcycle

It's the same idea as with classes. We're controlling which classes can implement our interface.

The Plot Thickens: Non-Final Subclasses

What if you want to allow further extension of your subclasses? You have options:

1. public sealed class Shape permits Circle, Polygon {
2. // Shape implementation
3. }
5. public final class Circle extends Shape {
6. // Circle implementation
7. }
9. public sealed class Polygon extends Shape
10. permits Triangle, Square {
11. // Polygon implementation
12. }
14. public final class Triangle extends Polygon {
15. // Triangle implementation
16. }
18. public final class Square extends Polygon {
19. // Square implementation
20. }

Here, Polygon is itself a sealed class, creating a more complex, but still controlled, hierarchy.

Why Should You Care?

1. **API Design**: You can design APIs with more precision, controlling how your classes are used.
2. **Exhaustiveness**: When combined with pattern matching, the compiler can ensure you've handled all possible subclasses.
3. **Clear Intent**: Your code communicates its design more clearly to other developers.
4. **Maintenance**: You can evolve your codebase with confidence, knowing exactly what depends on your base classes.

Use Cases for API Design

Sealed classes shine in several scenarios:

1. **Domain Modeling**: When you have a fixed set of types that model your domain.
2. **State Machines**: Represent a finite set of states in your application.
3. **Algebraic Data Types**: Implement sum types or tagged unions in Java.
4. **Framework Development**: Control how developers can extend and use your framework classes.

The Future is Controlled (in a good way)

Sealed Classes aren't just a neat feature; they're a powerful tool for designing more robust and maintainable APIs. They show us that Java is evolving to give developers more fine-grained control over their code structure.

So, are you ready to start sealing your class hierarchies? Remember, it's not about restricting creativity; it's about clearly expressing your design intentions and creating more robust software.

Now go forth and seal those classes! Your future self (and your team) will thank you when they don't have to deal with unexpected subclasses popping up where they shouldn't.