1. Constructor Pattern

In Java, constructors are a fundamental part of class definition, initializing instance variables with parameters.

```java
public class Person {
    private String name;
    private int age;
    private boolean isDeveloper;

    public Person(String name, int age, boolean isDeveloper) {
        this.name = name;
        this.age = age;
        this.isDeveloper = isDeveloper;
    }

    public void writesCode() {
        System.out.println(isDeveloper ? "this person writes code" : "this person does not write code");
    }

    public static void main(String[] args) {
        Person person1 = new Person("Ana", 32, true);
        Person person2 = new Person("Bob", 36, false);
        person1.writesCode(); // prints: this person writes code
        person2.writesCode(); // prints: this person does not write code
    }
}
```

2. Module Pattern

Java lacks closures, so encapsulation is achieved using private fields and public methods, often with a singleton-like approach.

```java
public class FruitCollection {
    private List<String> objects = new ArrayList<>();

    private FruitCollection() {} // Private constructor for encapsulation

    public static FruitCollection getInstance() {
        return SingletonHolder.INSTANCE;
    }

    private static class SingletonHolder {
        private static final FruitCollection INSTANCE = new FruitCollection();
    }

    public void addObject(String object) {
        objects.add(object);
    }

    public void removeObject(String object) {
        objects.remove(object);
    }

    public List<String> getObjects() {
        return new ArrayList<>(objects); // Return a copy to prevent modification
    }
```

```
    public static void main(String[] args) {
        FruitCollection collection = FruitCollection.getInstance();
        collection.addObject("apple");
        collection.addObject("orange");
        System.out.println(collection.getObjects()); // prints: [apple, orange]
    }
}
```

3. Revealing Module Pattern

In Java, this can be simulated with a class exposing specific methods while hiding implementation details.

```
public class FruitCollection {
    private List<String> objects = new ArrayList<>();

    private void addObject(String object) {
        objects.add(object);
    }

    private void removeObject(String object) {
        objects.remove(object);
    }

    private List<String> getObjects() {
        return new ArrayList<>(objects);
    }

    public static class PublicInterface {
        public static void addName(FruitCollection collection, String object) {
            collection.addObject(object);
        }

        public static void removeName(FruitCollection collection, String object) {
            collection.removeObject(object);
        }

        public static List<String> getNames(FruitCollection collection) {
            return collection.getObjects();
        }
    }

    public static void main(String[] args) {
        FruitCollection collection = new FruitCollection();
        PublicInterface.addName(collection, "Bob");
        PublicInterface.addName(collection, "Alice");
        System.out.println(PublicInterface.getNames(collection)); // prints: [Bob, Alice]
    }
}
```

4. Singleton Pattern

Java's standard singleton pattern uses a static inner class for lazy initialization.

```
public class ConfigurationSingleton {
    private int number = 5;
    private int size = 10;
```

```
    private double randomNumber = Math.random();

    private ConfigurationSingleton() {}

    public static ConfigurationSingleton getInstance() {
        return SingletonHolder.INSTANCE;
    }

    private static class SingletonHolder {
        private static final ConfigurationSingleton INSTANCE = new ConfigurationSingleton();
    }

    public void initialize(int number, int size) {
        this.number = number;
        this.size = size;
    }

    @Override
    public String toString() {
        return "ConfigurationSingleton{number=" + number + ", size=" + size + ", randomNumber=" + randomNumber + "}";
    }

    public static void main(String[] args) {
        ConfigurationSingleton config1 = ConfigurationSingleton.getInstance();
        config1.initialize(8, 10);
        System.out.println(config1); // prints: ConfigurationSingleton{number=8, size=10, randomNumber=someValue}
        ConfigurationSingleton config2 = ConfigurationSingleton.getInstance();
        System.out.println(config2); // prints same instance
    }
}
```

5. Observer Pattern (Publisher/Subscriber)

Java provides the `Observer` and `Observable` classes, or we can implement a custom version.

```
import java.util.ArrayList;
import java.util.List;

public class PublisherSubscriber {
    static class Subscriber {
        private String id;
        private Consumer<Object> callback;

        public Subscriber(String id, Consumer<Object> callback) {
            this.id = id;
            this.callback = callback;
        }

        public void notify(Object data) {
            callback.accept(data);
        }
    }

    private List<Subscriber> subscribers = new ArrayList<>();
```

```
    public void subscribe(String id, Consumer<Object> callback) {
        subscribers.add(new Subscriber(id, callback));
    }

    public void unsubscribe(String id) {
        subscribers.removeIf(sub -> sub.id.equals(id));
    }

    public void publish(Object data) {
        for (Subscriber sub : new ArrayList<>(subscribers)) {
            sub.notify(data);
        }
    }

    public static void main(String[] args) {
        PublisherSubscriber pubSub = new PublisherSubscriber();
        pubSub.subscribe("sub1", data -> System.out.println("Event: " + data));
        pubSub.publish("Hello"); // prints: Event: Hello
        pubSub.unsubscribe("sub1");
    }
}
```

6. Mediator Pattern

A mediator class coordinates interactions between components.

```
import java.util.HashMap;
import java.util.Map;

public class Mediator {
    private Map<String, Step> steps = new HashMap<>();

    public void register(String name, Step step) {
        steps.put(name, step);
    }

    public void next(String currentStep) {
        Step step = steps.get(currentStep);
        if (step != null && step.getNext() != null) {
            step.getNext().run();
        }
    }

    interface Step {
        Runnable getNext();
    }

    public static void main(String[] args) {
        Mediator mediator = new Mediator();
        Step step1 = () -> System.out.println("Step 2");
        mediator.register("step1", () -> step1.getNext());
        mediator.next("step1"); // prints: Step 2
    }
}
```

7. Prototype Pattern

Java uses cloning or prototype instantiation for this pattern.

```
public class Person implements Cloneable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void sayHi() {
        System.out.println("Hi, I am " + name + " and I have " + age);
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public static void main(String[] args) throws CloneNotSupportedException {
        Person person1 = new Person("John Doe", 26);
        Person person2 = (Person) person1.clone();
        person2.sayHi(); // prints: Hi, I am John Doe and I have 26
    }
}
```

8. Command Pattern

Encapsulates requests as objects.

```
interface Command {
    void execute();
}

class Calculator {
    public int add(int x, int y) { return x + y; }
    public int subtract(int x, int y) { return x - y; }
}

class CommandManager {
    private Calculator calculator = new Calculator();

    public void execute(String command, int... args) {
        switch (command) {
            case "add" -> System.out.println(calculator.add(args[0], args[1]));
            case "subtract" -> System.out.println(calculator.subtract(args[0], args[1]));
            default -> System.out.println("Command not found");
        }
    }
}

public class Main {
    public static void main(String[] args) {
```

```
        CommandManager manager = new CommandManager();
        manager.execute("add", 3, 5); // prints: 8
        manager.execute("subtract", 5, 3); // prints: 2
    }
}
```

9. Facade Pattern

Simplifies a complex subsystem.

```
import java.util.List;

public class Facade {
    private DOMSelector selector = new DOMSelector();

    public List<Element> select(String query) {
        return selector.query(query);
    }
}

class DOMSelector {
    public List<Element> query(String query) {
        // Simulated complex logic
        return List.of(new Element());
    }
}

class Element {}

public class Main {
    public static void main(String[] args) {
        Facade facade = new Facade();
        List<Element> elements = facade.select(".parent .child");
        System.out.println(elements.size()); // prints: 1 (simulated)
    }
}
```

These Java implementations adapt the JavaScript patterns to Java's paradigm, ensuring functionality while respecting language constraints.