✅ 1. What is Prototype Design Pattern?

> The **Prototype Pattern** is a creational design pattern that allows you to **clone existing objects** instead of creating new ones from scratch.

It’s especially useful when:

- Object creation is **expensive or complex**
- You need many **similar objects** with slight differences

🎯 Real-Life Analogy

Imagine you're designing avatars for a game. You have a **default warrior template**—each player gets a copy and customizes it.

Instead of building each avatar from scratch, you **clone** the prototype and tweak it.

✅ 2. Prototype Pattern in Java – Step-by-Step

🔹 Step 1: Define a Prototype Interface

```
public interface Prototype extends Cloneable {
    Prototype clone();
}
```

🔹 Step 2: Concrete Class Implements Clone

```
public class Vehicle implements Prototype {
    private String model;
    private Engine engine;  // complex/nested object

    public Vehicle(String model, Engine engine) {
        this.model = model;
        this.engine = engine;
    }

    @Override
    public Vehicle clone() {
        try {
            return (Vehicle) super.clone();  // shallow copy
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }

    public String getModel() {
        return model;
    }

    public Engine getEngine() {
```

```
        return engine;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    @Override
    public String toString() {
        return model + " - " + engine;
    }
}
```

🔹 Step 3: Engine Class

```
public class Engine {
    private String type;

    public Engine(String type) {
        this.type = type;
    }

    public String toString() {
        return "Engine: " + type;
    }

    public void setType(String type) {
        this.type = type;
    }
}
```

🔹 Step 4: Client Code

```
public class PrototypeDemo {
    public static void main(String[] args) {
        Engine dieselEngine = new Engine("Diesel");
        Vehicle original = new Vehicle("SUV", dieselEngine);

        Vehicle clone = original.clone();
        clone.setModel("SUV Clone");
        clone.getEngine().setType("Electric");

        System.out.println("Original: " + original); // Engine: Electric
        System.out.println("Clone   : " + clone);    // Engine: Electric
    }
}
```

> ⚠ Both vehicles now share the same `Engine` object (due to **shallow copy**)

🧠 3. Shallow vs Deep Copy

Aspect	Shallow Copy	Deep Copy
Copy level	Copies top-level object only	Copies entire object graph (nested objects)
Nested objects	Shared references	Separate clones of nested objects
Risk	Modifying one affects the other	Safe isolation
Performance	Faster, less memory	Slower, more memory

🛠️ Shallow Copy in Java (Default `clone()`)

```
java
CopyEdit
Vehicle clone = (Vehicle) original.clone();
```

- Copies `model` string
- Shares reference to `engine`

🛠️ Deep Copy Example

Update `Vehicle`'s `clone()` method:

```
@Override
public Vehicle clone() {
    Engine clonedEngine = new Engine(this.engine.toString());
    return new Vehicle(this.model, clonedEngine);
}
```

Or make `Engine` cloneable:

```
public class Engine implements Cloneable {
    private String type;

    public Engine(String type) {
        this.type = type;
    }

    @Override
    protected Engine clone() {
        return new Engine(this.type);
    }
}
```

Then in `Vehicle`:

```
@Override
public Vehicle clone() {
    return new Vehicle(this.model, this.engine.clone());  // deep copy
}
```

✅ When to Use Prototype Pattern?

Scenario	Suitability
Expensive object creation (e.g., DB configs)	✅
Multiple copies of mostly similar objects	✅
You want to decouple code from constructors	✅
Deep object trees (careful with shared refs)	⚠ Yes, with deep copy logic

---

✅ Summary

- **Prototype pattern** lets you create objects by **copying** an existing one.
- Java supports cloning using `Object.clone()` but it's **shallow by default**.
- For **deep cloning**, you must manually clone all nested objects.
- Helps in reducing cost of creating similar objects.

📝 Bonus Tip (Effective Java – Joshua Bloch)

> "Avoid using `Cloneable` and `Object.clone()` unless you're fully in control. Prefer copy constructors or static factory methods."

✅ So, an alternative clean way is:

```
java
CopyEdit
public Vehicle(Vehicle other) {
    this.model = other.model;
    this.engine = new Engine(other.engine.getType());
}
```