







The **Prototype Pattern** is a **creational design pattern** where you create new objects by **cloning (copying)** an existing object — called the prototype — instead of creating them from scratch.

### ****Definition****

“Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.”  
— Gang of Four (GoF)

### ****Key Idea****

* You **already have an object** that is properly initialized.
* Instead of calling new and setting up everything again, you **clone** the object.
* This is useful when **object creation is costly** (e.g., reading from DB, network calls, heavy computation).

### ****Real-life Analogy****

Think of **photocopying a document**:

* The original document is your **prototype**.
* The photocopier **clones** it into new copies.
* Each copy can be slightly modified if needed.

### ****When to Use****

* **Performance critical** applications where new object creation is expensive.
* You want to **avoid subclassing** and use an existing object as a template.
* You need to create objects **dynamically at runtime** without knowing their concrete types.

### ****Example****

If you have a **configured game character** and want multiple similar characters:

java

CopyEdit

GameCharacter prototype = new GameCharacter("Knight", 100, 50);

GameCharacter another = prototype.clone();

You get a **new object** that starts with the same properties, but you can change it without affecting the original.

**Use Case**

The **Prototype Pattern** is useful when:

* Creating new objects is **expensive** (in terms of performance, memory, or configuration).
* You want to **create a copy of an existing object** instead of creating it from scratch.
* You need to **maintain object state** during duplication.
* Example scenarios:
  1. **Game development** – Creating multiple similar enemies or items with small variations.
  2. **Document editors** – Copy-pasting shapes, styles, or components.
  3. **Caching** – Store a prototype object and clone it for new requests.
  4. **Configuration objects** – Reuse a pre-configured template object.

**Advantages**

1. **Performance** – Faster than building a new object when initialization is costly.
2. **Reduced Complexity** – No need to repeat initialization code.
3. **Dynamic Object Creation** – You can create objects at runtime without depending on concrete classes.
4. **Easier to Add Variations** – Just clone and tweak a few fields.

**Disadvantages**

1. **Cloning Can Be Tricky** – Especially with **deep copies** (nested objects) vs **shallow copies**.
2. **Complex Object Graphs** – If the object has many references, you must ensure correct copy semantics.
3. **Hidden Dependencies** – If the prototype contains references to mutable shared objects, clones may unintentionally share state.

## ****Example****

java

CopyEdit

// Prototype Interface

public interface Prototype<T> {

T clone();

}

// Concrete Prototype

public class Document implements Prototype<Document> {

private String content;

private String author;

public Document(String content, String author) {

this.content = content;

this.author = author;

}

@Override

public Document clone() {

return new Document(this.content, this.author); // Shallow copy

}

@Override

public String toString() {

return "Document{" + "content='" + content + "', author='" + author + "'}";

}

}

// Client

public class Main {

public static void main(String[] args) {

Document original = new Document("Report", "John");

Document copy = original.clone();

System.out.println(original);

System.out.println(copy);

}

}

If you want, I can **extend this example** to show a **deep copy version** where the prototype contains a list or complex object