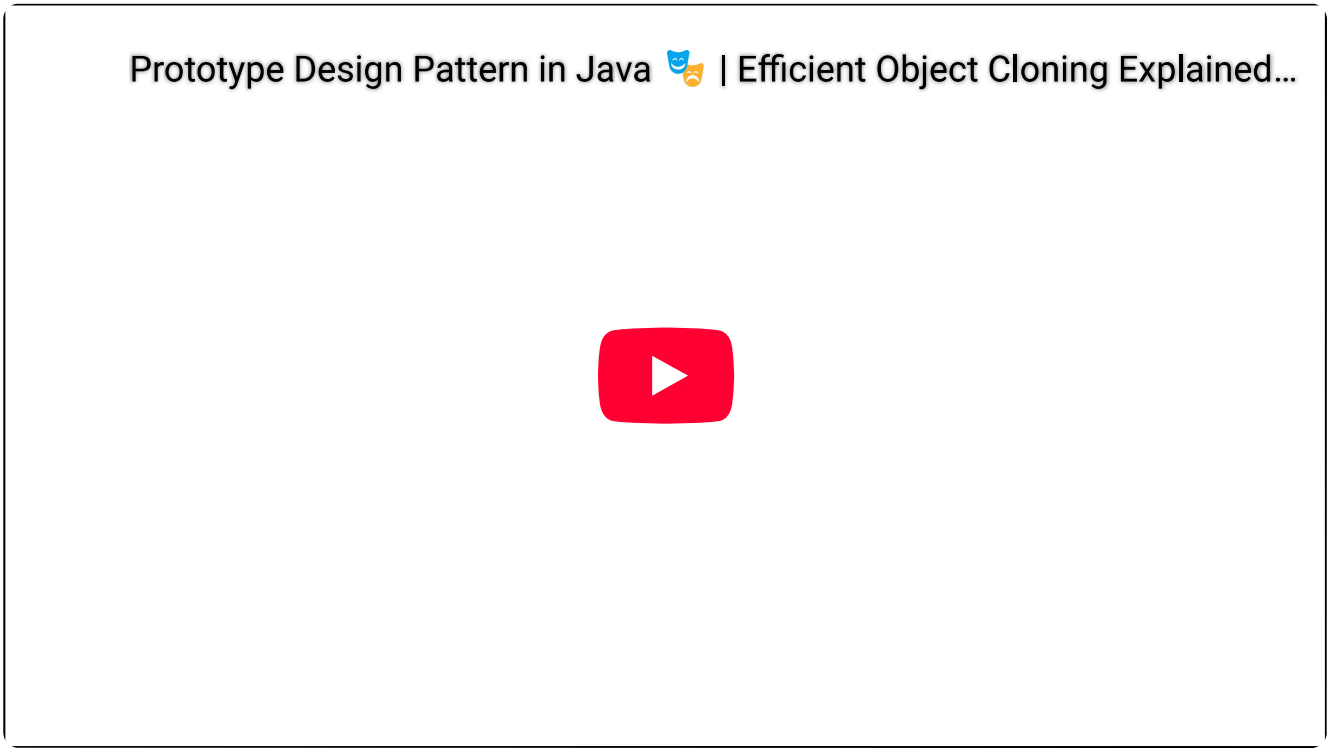
Search articles...

Prototype Design Pattern



Topic Tags:

System Design LLD

Prototype Design Pattern: Making Cloning Objects Easy

Let's start by understanding the Prototype Design Pattern. In real life, when you create something, sometimes you don't need to start from scratch every time. Instead, you clone an existing thing and make a few small changes. It's like having a cookie cutter 🍪 to make several cookies in the same shape instead of baking them one by one from scratch.

This is exactly what the Prototype Design Pattern does in programming. It allows you to create a new object by cloning an existing prototype and making small modifications to it. This pattern is particularly useful when creating objects that are very similar, saving time and effort. ⏳ 🔄
Now, let's look at a scenario to see how this pattern works in practice! 🎮 💡

Solving a Scenario with the Traditional Method 🎮

Let's imagine we're working on a video game where players can create custom characters. Each character has a name, health, attack power, and level. However, some players want to create characters that are very similar to others but with a few small changes (e.g., a different name or level). 🤔

Without the Prototype pattern, we might approach this problem like this:

Traditional Approach 👷

In the traditional approach, we create a new character object every time, manually setting all the attributes even if most of them stay the same. ⚙️

Java

```java
public class Character {
  private String name;
  private int health;
  private int attackPower;
  private int level;
  public Character(String name, int health, int attackPower, int level) {
    this.name = name;
    this.health = health;
    this.attackPower = attackPower;
    this.level = level;
  }
  public void showCharacterInfo() {
    System.out.println("Character [Name=" + name + ", Health=" + health
        + ", AttackPower=" + attackPower + ", Level=" + level + "]");
  }
}
public class CharacterFactory {
  // Creating a new character each time with similar attributes
  public Character createCharacterWithNewName(String name) {
    // Creating a new character with the same attributes, just changing th
    return new Character(name, 100, 50, 1); // Default attributes for simp
  }
  public Character createCharacterWithNewLevel(int level) {
    // Creating a new character with the same attributes, just changing th
    // level
    return new Character(
        "DefaultName", 100, 50, level); // Default name and attributes
  }
  public Character createCharacterWithNewAttackPower(int attackPower) {
```

```
30      // Creating a new character with the same attributes, just changing th
31      // attack power
32      return new Character(
33          "DefaultName", 100, attackPower, 1); // Default name and level
34    }
35  }
```

Issues with This Approach ⚠️

• Code duplication:
Every time we want to create a character with a small change, we repeat the same code over and over again, modifying just one or two values. 🔄 📝

• Inefficient:
If we have a large number of characters with only slight differences, we end up writing many similar methods, which leads to a lot of repetitive work. 📐❌

• Hard to maintain:
If we need to modify the creation logic (e.g., adding a new property like “armor”), we would need to update all the methods where we create characters. That’s messy! 🧹😣

Interviewer's Questions: Can We Do Better?

An interviewer might ask:
• What if we need to create many characters with similar attributes?
• Can we avoid writing so much repetitive code?
• How do we make the system scalable without adding new methods every time we need a slight change?

We realize that this method is getting ugly as we scale. We need a better solution to create characters without duplicating code and making the system harder to manage.

The Ugly Code

Here’s what the code starts looking like as we try to add more variations:

Java

```
1  public class CharacterFactory {
2    // Too many methods for every small change
3    public Character createCharacterWithNewName(String name) {
4      return new Character(name, 100, 50, 1);
```

```
  }
  public Character createCharacterWithNewLevel(int level) {
    return new Character("DefaultName", 100, 50, level);
  }
  public Character createCharacterWithNewAttackPower(int attackPower) {
    return new Character("DefaultName", 100, attackPower, 1);
  }
  public Character createCharacterWithNewHealth(int health) {
    return new Character("DefaultName", health, 50, 1);
  }
  // More and more methods for every possible variation...
}
```

As you can see, this approach quickly becomes hard to maintain and scalable. We end up creating a bunch of methods for every small change, which makes the code harder to read and manage. 😅

Introducing the Prototype Pattern: Our Savior

Now, let's introduce our savior: the Prototype Design Pattern! 🎉

Why is it Called Prototype?

> *The Prototype pattern is named so because it allows you to create a new object by cloning an existing prototype and modifying only what's necessary. It's like using a template (prototype) to create multiple similar objects with small variations.*

Solving the Problem with the Prototype Pattern

In the Prototype Pattern, instead of manually creating new objects each time by setting each attribute, we can clone an existing object (the prototype) and modify only the properties that need to change. This allows us to create similar objects quickly and efficiently.
Cloning the Character Object

Here's the correct code implementation:

Java

```
public class Character implements Cloneable {
  private String name;
  private int health;
  private int attackPower;
```

```
  private int level;
  public Character(String name, int health, int attackPower, int level) {
    this.name = name;
    this.health = health;
    this.attackPower = attackPower;
    this.level = level;
  }
  @Override
  public Character clone() throws CloneNotSupportedException {
    return (Character) super.clone(); // Shallow copy of the character obj
  }
  public void showCharacterInfo() {
    System.out.println("Character [Name=" + name + ", Health=" + health
        + ", AttackPower=" + attackPower + ", Level=" + level + "]");
  }
}
```

Explanation of the Code

• **Cloneable Interface:**
The Character class implements the Cloneable interface. This is necessary because Java's Object class provides a clone() method that can be used to clone objects, but this method only works if the class explicitly implements Cloneable.

Overriding the clone() Method:
The clone() method is overridden to allow cloning of the Character object. The super.clone() method performs a shallow copy of the object.

Java

```
@Override
public Character clone() throws CloneNotSupportedException {
    return (Character) super.clone();  // Shallow copy of the character obje
}
```

• Default Constructor:
The constructor initializes the attributes like name, health, attackPower, and level.

• showCharacterInfo():

This method displays the character's attributes. After cloning a character, we can modify the cloned character's properties while keeping the rest of the attributes the same.

Cloning the Prototype in the Factory

Now, let's see how we can use this clone() method to solve the problem of creating new characters that are similar to an existing prototype but with some modifications:

Java

```java
public class CharacterFactory {
  private Character prototypeCharacter;
  // Constructor to create a prototype character (default character)
  public CharacterFactory() {
    prototypeCharacter =
        new Character("DefaultName", 100, 50, 1); // Default prototype cha
  }
  // Create a character by cloning the prototype and changing only the req
  // attributes
  public Character createCharacterWithNewName(String name)
      throws CloneNotSupportedException {
    Character clonedCharacter = prototypeCharacter.clone();
    clonedCharacter = new Character(name, clonedCharacter.health,
        clonedCharacter.attackPower, clonedCharacter.level);
    return clonedCharacter;
  }
  public Character createCharacterWithNewLevel(int level)
      throws CloneNotSupportedException {
    Character clonedCharacter = prototypeCharacter.clone();
    clonedCharacter = new Character(clonedCharacter.name,
        clonedCharacter.health, clonedCharacter.attackPower, level);
    return clonedCharacter;
  }
  public Character createCharacterWithNewAttackPower(int attackPower)
      throws CloneNotSupportedException {
    Character clonedCharacter = prototypeCharacter.clone();
    clonedCharacter = new Character(clonedCharacter.name,
        clonedCharacter.health, attackPower, clonedCharacter.level);
    return clonedCharacter;
  }
}
```

Explanation of the Factory Code

• Prototype Object:
In the CharacterFactory constructor, we create a prototype character that serves as the template. This character is used as the base for creating new characters.

Java

```
prototypeCharacter = new Character("DefaultName", 100, 50, 1);  // Default p
```

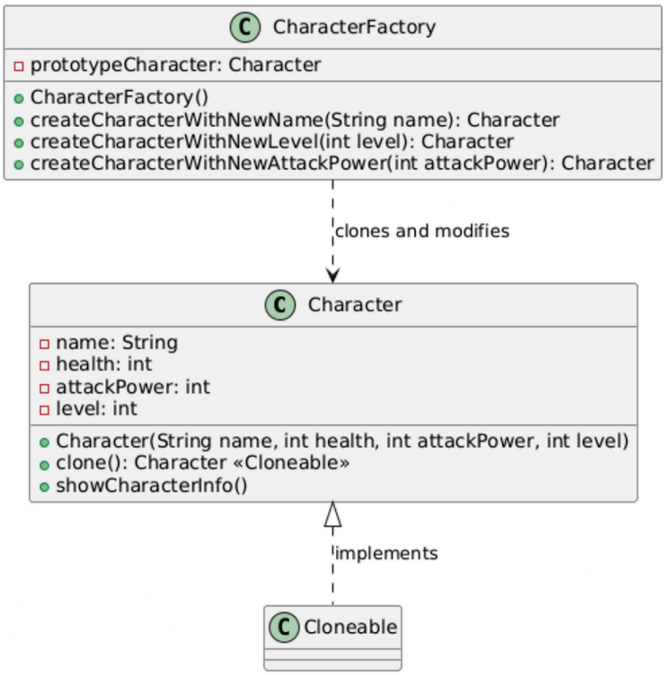
• Cloning and Modifying:
The createCharacterWithNewName, createCharacterWithNewLevel, and createCharacterWithNewAttackPower methods all clone the prototype character using the clone() method. After cloning, we modify only the attribute that needs to change (like name, level, or attackPower), while the rest of the attributes remain the same.

For example, to create a new character with a new name:

Java

```
public Character createCharacterWithNewName(String name)
    throws CloneNotSupportedException {
  Character clonedCharacter = prototypeCharacter.clone();
  clonedCharacter = new Character(name, clonedCharacter.health,
      clonedCharacter.attackPower, clonedCharacter.level);
  return clonedCharacter;
}
```

• Efficiency:
Instead of creating a new character from scratch every time, we are now cloning the prototype and making small modifications. This reduces code duplication and simplifies object creation, especially when we have many variations.



What's Different? 🤔

• Clone the prototype:
Instead of creating new characters from scratch, we clone the prototype character, which already has default values. 🔄✨

• Modify only what's necessary:
After cloning the prototype, we only modify the attributes that need to change (like name, level, or attack power). This means we don't have to repeat the logic for every variation. 🤷‍♂️💡

• No code duplication:
We no longer need to write separate methods for every possible variation. We simply clone the prototype and adjust it as needed. 🖌️🔧

Interviewer's Questions: Can We Do Better?

1. What if we need to create many characters with similar attributes?
With the traditional approach, we would have had to manually copy and paste code to create each variation of the character, which is inefficient and difficult to maintain as the number of variations grows. 📝⚠️

But with the Prototype Pattern, we solve this problem easily by cloning the prototype. The prototype character is our base template, and we can create as many characters as we need by cloning it and only changing the necessary attributes. 🔄✨ This is much faster and eliminates code duplication. 🚀

For example, we can easily create a large number of characters with different names, levels, or attack powers: 💥👾

Java

```java
CharacterFactory factory = new CharacterFactory();
Character warrior = factory.createCharacterWithNewName("Warrior");
Character mage = factory.createCharacterWithNewName("Mage");
Character knight = factory.createCharacterWithNewLevel(5);
```

Each time we clone the prototype and modify only the parts that are different, making it super easy to create many characters with similar attributes.

2. Can we avoid writing so much repetitive code?
Absolutely! That was the main pain point with the traditional approach, where we had to write multiple methods for each small variation, leading to a lot of repetitive code. 📝❌
With the Prototype Pattern, we only need one method to create a new character by cloning the prototype. Then, we simply adjust the required attributes (like name, level, attack power, etc.). There's no need to create separate methods for every possible variation. 🔄✨

For example, we don't need separate methods like createCharacterWithBlueColor, createCharacterWithRedColor, or createCharacterWithHighAttack. Instead, we clone the prototype and modify the needed properties in a single, efficient method. 🔧 This significantly reduces code repetition and makes it easier to manage. 💡

3. How do we make the system scalable without adding new methods every time we need a slight change?
The beauty of the Prototype Pattern is that we don't need to add new methods every time a slight change is required. As new character variations are needed, we simply clone the prototype and modify only the attributes that are different. This makes our system scalable without bloating the codebase with countless methods. 🛠️🔄

For example:
• To create a new character with a different name, we just clone the prototype and change the name. ✨
• To create a character with a new level, we clone and update the level. 🎮
• If we need to add a new property (like armor), we simply update the prototype character once and all clones will automatically have the same default properties. ⚔️🛡️ Any character that needs a different armor can be cloned and customized without writing new methods. 👷💡

Here's how scalable it is:

Java

```java
CharacterFactory factory = new CharacterFactory();
Character newCharacter = factory.createCharacterWithNewAttackPower(100);  //
```

The system is highly scalable because we are not adding a new method for each change. Instead, we're reusing the prototype and simply adjusting what needs to be different. As a result, the system remains clean, efficient, and easy to maintain as it grows.

Advantages of Using the Prototype Pattern ✨

1. Reduced code duplication:
By cloning an existing object, we avoid writing repetitive code for every variation. 🔄📝

2. Easier maintenance:
If we need to change something about how characters are created (e.g., adding a new attribute), we only need to update the prototype object. 🔧🛠️

3. Scalability:
As we add more character variations, we don't need to create new methods. We just clone and modify the prototype. 📈🔄

4. Cleaner and more flexible code:
This approach makes our codebase cleaner, more modular, and easier to maintain as the number of variations grows. 🖌️⚙️

Real-Life Use Cases of the Prototype Pattern 🌍

Here are some real-life examples where the Prototype Pattern is commonly used:
1. Game Development 🎮:
○ In games, many characters might be based on the same base class but have small variations. The Prototype Pattern allows developers to clone a base character and modify its attributes for different players, enemies, or NPCs.

2. Document Creation 📑:
○ When generating reports or documents, the Prototype Pattern can be used to clone a base template and modify only the sections that need to change (e.g., title, content, or layout).

3. GUI Frameworks 🖥️:

○ In GUI frameworks, components like buttons, labels, and text fields are often cloned from a prototype and customized according to user needs.

4. Configuration Settings ⚙️:
○ A configuration object with default values can be cloned and modified for each user or process, ensuring consistency while minimizing the need for creating objects from scratch.

🎉 Conclusion: Simplifying Object Creation with the Prototype Pattern

The Prototype Design Pattern is a powerful and efficient way to create new objects by cloning existing prototypes and making small modifications. This pattern eliminates the need for repetitive code, makes maintenance easier, and improves the flexibility of your code. Whether you're building game characters, generating documents, or creating configuration settings, the Prototype Pattern can make object creation faster, cleaner, and more efficient. 😊

Now, instead of building objects from scratch every time, you can simply clone a prototype and make quick changes! How cool is that? 🌟