Factory Method Pattern (auf Deutsch: **Fabrikmethode-Entwurfsmuster**) wird verwendet, , um die **Erzeugung von Objekten** zu kapseln und die Flexibilität bei der Erstellung von Objekten zu erhöhen.

Anstatt die **Klasse eines Objekts direkt** im Code zu verwenden (zum Beispiel mit `new`), delegiert das Factory Method Pattern die Verantwortung für die Objekterstellung an Unterklassen oder eine spezielle Methode. Dies fördert das Prinzip der **Abstraktion**, indem die konkrete Logik zur Erzeugung von Objekten von der Klasse getrennt wird, die das Objekt verwendet.

Was ist das Ziel des Factory Method Patterns?

Das Ziel des **Factory Method Patterns** ist es, den **Erstellungsprozess von Objekten** so flexibel zu gestalten, dass der Code nicht direkt von konkreten Klassen abhängig ist. Dies ist nützlich, wenn:

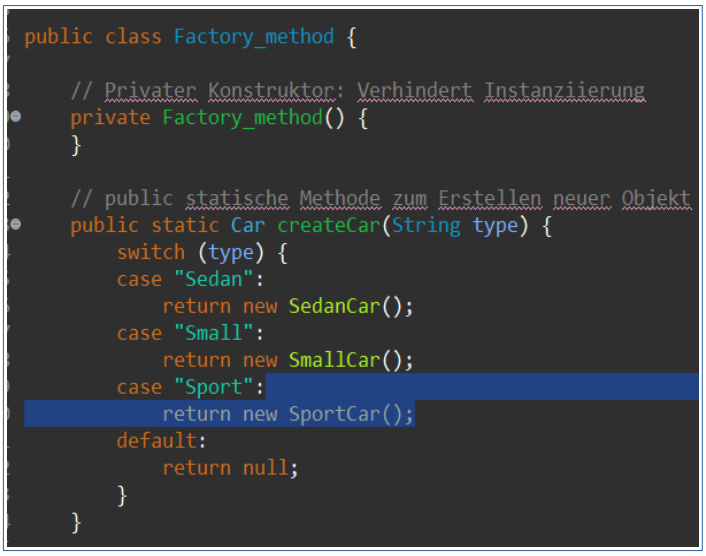
- Der genaue Typ des zu erstellenden Objekts **zur Laufzeit** festgelegt werden muss.
- Du möchtest, dass der Code **offen für Erweiterungen** ist, ohne dass der bestehende Code geändert werden muss.
- Du eine einfache Möglichkeit brauchst, **neue Objekttypen** hinzuzufügen, ohne den bestehenden Code zu ändern.

Wie funktioniert das Factory Method Pattern?

1. **Abstraktion**: Du erstellst eine **abstrakte Klasse** oder **Schnittstelle** für das zu erstellende Objekt.
2. **Fabrikmethode**: Du definierst eine **abstrakte Methode** oder eine **Factory-Methode**, die den **Erstellungsprozess** kapselt. Unterklassen oder konkrete Implementierungen entscheiden, wie das Objekt erstellt wird.
3. **Flexibilität**: Der aufrufende Code arbeitet mit der **abstrakten Klasse oder Schnittstelle** und weiß nicht, welche konkrete Implementierung tatsächlich erstellt wird.

Fabrikmethode hat:

- Klasse ist public
- einem privaten Konstruktor
- öffentlichen statische Methode zum Erstellen neuer Objekt



```
public class Factory_method {

    // Privater Konstruktor: Verhindert Instanziierung
    private Factory_method() {
    }

    // public statische Methode zum Erstellen neuer Objekt
    public static Car createCar(String type) {
        switch (type) {
        case "Sedan":
            return new SedanCar();
        case "Small":
            return new SmallCar();
        case "Sport":
            return new SportCar();
        default:
            return null;
        }
    }
```

Vorteile des Factory Method Patterns:

1. **Trennung von Objekterstellung und Geschäftslogik**: Das Factory Method Pattern trennt den Code, der das Objekt verwendet, von der **konkreten Objekterstellung**. Dies macht den Code flexibler und weniger abhängig von den konkreten Klassen.
2. **Erweiterbarkeit**: Wenn du neue Klassen hinzufügen möchtest, die dieselbe Abstraktion verwenden (wie neue Tiere), kannst du einfach neue Fabrikklassen erstellen, ohne den bestehenden Code zu ändern.
3. **Flexibilität**: Die Entscheidung, welche Klasse instanziiert werden soll, kann **dynamisch** getroffen werden (z. B. basierend auf Benutzerinput oder Konfigurationsdateien).
4. **Vermeidung von Abhängigkeiten**: Der Code, der Objekte erstellt, ist **nicht direkt** von den konkreten Klassen abhängig. Stattdessen hängt er nur von der **Abstraktion** ab.
5. **Besser testbar**: Da der Code nur mit abstrakten Schnittstellen arbeitet, kannst du leicht **Mock-Objekte** oder **Dummy-Objekte** in Tests verwenden.

Zusammenfassung:

Das **Factory Method Pattern** ist ein kreatives Entwurfsmuster, das verwendet wird, um die **Erstellung von Objekten zu kapseln**. Es trennt die **Logik der Objekterstellung** von der Verwendung der Objekte und erlaubt es, den Code flexibler und erweiterbarer zu gestalten. Es fördert die **Abstraktion** und macht den Code weniger abhängig von konkreten Implementierungen.

- Es eignet sich gut für Szenarien, in denen die **Art des zu erstellenden Objekts zur Laufzeit** entschieden werden muss oder wenn der Code leicht erweiterbar sein soll.
- Der Hauptvorteil des Patterns liegt darin, dass es den **Erstellungsprozess von Objekten abstrahiert** und dem Entwickler erlaubt, neue Klassen hinzuzufügen, ohne den bestehenden Code zu ändern.

Beispiel:

```
}

// oder Interface
abstract class Car{
    public abstract int getPrice();
}

class SedanCar extends Car {
    @Override
    public int getPrice() {
        return 100000;
    }
}

class SmallCar extends Car {
    @Override
    public int getPrice() {
        return 50000;
    }
}

class SportCar extends Car {
    @Override
    public int getPrice() {
        return 250000;
    }
}

```

```java
public class Factory_method {

    // Privater Konstruktor: Verhindert Instanziierung
    private Factory_method() {
    }

    // public statische Methode zum Erstellen neuer Objekt
    public static Car createCar(String type) {
        switch (type) {
        case "Sedan":
            return new SedanCar();
        case "Small":
            return new SmallCar();
        case "Sport":
            return new SportCar();
        default:
            return null;
        }
    }

    public static void main(String[] args) {

        Car sedan = Factory_method.createCar("Sedan");
        Car small = Factory_method.createCar("Small");
        Car sport = Factory_method.createCar("Sport");

        System.out.println("Price of Sedan Car: $" + sedan.getPrice());
        System.out.println("Price of Small Car: $" + small.getPrice());
        System.out.println("Price of Sport Car: $" + sport.getPrice());

    }

}
```