Privater Konstruktor

Ein privater Konstruktor ist ein Konstruktor, dessen Zugriff auf die Klasse selbst beschränkt ist. Dies wird durch ein vorangestelltes Unterstrichzeichen (_) im Konstruktornamen erreicht. Ein privater Konstruktor verhindert, dass Instanzen der Klasse von außerhalb des Moduls erstellt werden.

Warum benutzt man einen privaten Konstruktor?

Es gibt zwei grundlegende Gründe, warum man einen privaten Konstruktor benötigt:

1. **Den Bau der Klasse verbieten**:

Betrachten wir zum Beispiel eine `Farbe`-Klasse, die unsere eigenen Farben enthält. Diese Farben sind als statische Variablen definiert, das heißt, sie können direkt über die Klasse selbst aufgerufen werden. In diesem Fall brauchen wir keine Instanzierung der Klasse. Um dies sicherzustellen, geben wir der Klasse einen privaten Konstruktor, sodass keine Instanzen außerhalb der Klasse erstellt werden können.

```
void main(){
    final farbe = Farben.blau;
    }
class Farben {
  Farben._();
  // Die Farben kann aus der Klasse direkt zugegriffen werden
  static const Color rot = Colors.red;
  static const Color gruen = Color.green;
  static const Color blau = Colors.blue;
  static const Color gelb = Colors.yellow ;
  static const Color schwarz = Colors.black;
  static const Color weiss = Colors.white;
}
```

2. **Den Bau der Klasse einrichten und begrenzen:**:

Situationsklärung: Angenommen, ich habe eine `Fahrzeug`-Klasse, die nicht direkt instanziiert werden soll. Stattdessen soll es möglich sein, über diese `Fahrzeug`-Klasse drei Unterklassen (Auto, Bus und Lkw) zu erstellen.

Um die Instanziierung der `Fahrzeug`-Klasse zu verhindern, macht man ihren Konstruktor privat. Um jedoch die Erstellung der drei Objekte zu ermöglichen, kann man drei Factory-Methoden erstellen. Jede dieser Factory-Methoden verwendet den privaten Konstruktor, um eine spezifische Instanz zu bauen. Dadurch kann man zwar keine `Fahrzeug`-Instanz direkt erstellen, aber über die Factory-Methoden kann man `Auto`, `Bus` und `Lkw` instanziieren.

Wichtige Hinweise:

- Als Alternative zum privaten Konstruktor könnte man nicht die `Fahrzeug`-Klasse abstrakt machen, sodass könnte man ihn mit Factory-Methoden nicht nutzen. Daher ist es zwingend notwendig, einen privaten Konstruktor zu verwenden.

- Warum kann der private Konstruktor innerhalb der Factory-Methoden verwendet werden? Weil die privaten Werte innerhalb der gleichen Datei zugänglich sind. Die Factory-Methoden können auf den privaten Konstruktor zugreifen, weil sie in derselben Datei definiert sind.
- Wie kann man spezifische Methoden für jede Factory bereitstellen? Es gibt keinen direkten Weg, um für jede Factory spezifische Methoden zu definieren. Aber man kann einen Trick anwenden: Man erstellt ein `String`-Attribut, das den Typ der Factory angibt, und gibt diesem Attribut bei jeder Factory den entsprechenden Typ. Je nach diesem Typ wird dann die passende Methode ausgewählt.

```
// Wegen des privaten Konstruktors kann ein Objekt nicht direkt erstellt werden
// der Bau des Objekts wird nur durch die Factory-Methode ermöglicht
 class FahrzeugGerichtetBeiFactorie {
  final String name;
  final int geschwindigkeit;
// _typ ist verwended um die Methoden zu spezifizieren
  final String _typ;
  final int sitzplaetze;
  // Privater Konstruktor, kann keine Objekte direkt erstellen
  FahrzeugGerichtetBeiFactorie._(this.name, this.geschwindigkeit, this._typ,
this.sitzplaetze);
  // Factory Methoden, die die Objekte erstellen
  factory FahrzeugGerichtetBeiFactorie.auto({required String name, required int
geschwindigkeit}) {
    return FahrzeugGerichtetBeiFactorie._(name, geschwindigkeit, 'Auto', 5);
  }

  factory FahrzeugGerichtetBeiFactorie.bus(
      {required String name, required int geschwindigkeit, required int
sitzplaetze}) {
    return FahrzeugGerichtetBeiFactorie._(name, geschwindigkeit, 'Bus',
sitzplaetze);
  }

  factory FahrzeugGerichtetBeiFactorie.lkw({required String name, required int
geschwindigkeit}) {
    return FahrzeugGerichtetBeiFactorie._(name, geschwindigkeit, 'LKW', 2);
  }
   // Getters und Methoden geltend für alle Fahrzeuge Typen
  String get info => 'FahrzeugDirectedWithFactorie: name: $name, Geschwindigkeit:
$geschwindigkeit';

  void fahren() {
    // durch die Typen wird die Methode fahren() spezifiziert
    if (_typ == 'Auto') print('Das Auto fährt');
    if (_typ == 'Bus') print('Der Bus fährt');
    if (_typ == 'LKW') print('Der LKW fährt');
  }

  void bremsen() {
    print('Das Fahrzeug bremst');
  }
}
```

Diese Methode ist nicht sehr flexibel, da alle Objekte die gleichen Methoden haben und es kompliziert wird, spezifische Methoden zu definieren. Daher erklären wir eine andere Methode, um die Einrichtung und Begrenzung einer Klasse durch einen privaten Konstruktor zu ermöglichen.

Der Bau mittels Unterklassen

Man kann die `Fahrzeug`-Klasse abstrakt machen und die Klassen `Auto`, `Bus` und `Lkw` als separate Klassen erstellen, die alle von der `Fahrzeug`-Klasse erben.

Die `Fahrzeug`-Klasse hat dann drei Factory-Methoden: `Auto`, `Bus` und `Lkw`. Obwohl die Factory-Methoden den privaten Konstruktor der `Fahrzeug`-Klasse nicht direkt aufrufen können, können sie die Konstruktoren der Unterklassen `Auto`, `Bus` und `Lkw` aufrufen. Dies ist möglich, weil diese Unterklassen erstens von der `Fahrzeug`-Klasse erben und daher als `Fahrzeug` betrachtet werden können, und zweitens sind sie nicht abstrakt. In diesem Fall erhält man eine größere Flexibilität mit den Tochterklassen, um verschiedene Attribute und Methoden für jede Tochterklasse (`Auto`, `Bus`, `Lkw`) bauen zu können.

```
// Weil die Klasse abstract ist, kann sie nicht direkt instanziiert werden
// der Bau des Objekts wird nur durch die Factory-Methode und Unterklassen
ermöglicht
abstract class FahrzeugGerichtetBeiUnterKlassen {
  String name;
  int geschwindigkeit;
  // Privater Konstruktor
  FahrzeugGerichtetBeiUnterKlassen._(this.name, this.geschwindigkeit);

  factory FahrzeugGerichtetBeiUnterKlassen.auto({required String name, required
int geschwindigkeit}) {
    return Auto._(name: name, geschwindigkeit: geschwindigkeit);
  }

  factory FahrzeugGerichtetBeiUnterKlassen.bus(
      {required String name, required int geschwindigkeit, required int
sitzplaetze}) {
    return Bus._(name: name, geschwindigkeit: geschwindigkeit, sitzplaetze:
sitzplaetze);
  }

  factory FahrzeugGerichtetBeiUnterKlassen.lkw(
      {required String name, required int geschwindigkeit, required int maxLast})
{
    return LKW._(name: name, geschwindigkeit: geschwindigkeit, maxLast: maxLast);
  }
  // Abstract Methoden, weil sie in den Unterklassen implementiert werden müssen
  void fahren();
}

// Private Klassen, die nur durch die abstrakten Klasse instanziiert werden können
class Auto extends FahrzeugGerichtetBeiUnterKlassen {
  // Auch die Tochterklassen haben private Konstruktoren.
  // weil sie sollen nicht direkt instanziiert werden.
```

```
  Auto._({required name, required geschwindigkeit}) : super._(name,
geschwindigkeit);

  String get info => '$Auto: $name, $geschwindigkeit';
  // Die Methode fahren() wird in der Klasse implementiert
  @override
  void fahren() {
    print('Das Auto fährt');
  }

// man kann spezifische Methoden für die Unterklasse implementieren
  void bremsen() {
    print('Das Auto bremst');
  }
}

class Bus extends FahrzeugGerichtetBeiUnterKlassen {
  // Die Unterklassen können auch spezifische Attribute haben
  int sitzplaetze;
  Bus._({required name, required geschwindigkeit, required this.sitzplaetze}) :
super._(name, geschwindigkeit);

  @override
  void fahren() {
    print('Der Bus fährt');
  }
}

class LKW extends FahrzeugGerichtetBeiUnterKlassen {
  int maxLast;
  LKW._({required name, required geschwindigkeit, required this.maxLast}) :
super._(name, geschwindigkeit);
  @override
  void fahren() {
    print('Der LKW fährt');
  }
}
```

Wichtiger Hinweis:

Bei der Erstellung eines Objekts nach dieser Methode wird das erstellte Objekt immer als `Fahrzeug` betrachtet. Dadurch sind die in der Tochterklasse definierten Attribute und Methoden, wie zum Beispiel `sitzplätze` in der `Bus`-Klasse, nicht direkt verfügbar. Um dieses Problem zu lösen, muss man beim Erstellen eines Objekts klarstellen, dass das erstellte Objekt entweder ein `Auto`, `Bus` oder `Lkw` ist. Dies kann durch das Schlüsselwort `as` erfolgen, zum Beispiel:

```
Fahrzeug fahrzeug = Factory.auto();
Auto auto = fahrzeug as Auto; // Hier wird das Objekt explizit als Auto behandelt
```

Aufgabe

Erstellen Sie eine `Mensch`-Klasse, die einen privaten Konstruktor verwendet. Erstellen Sie dann die Unterklassen `Arzt`, `Fußballspieler` und `Koch`. Die `Mensch`-Klasse soll allgemeine Attribute und Methoden enthalten, während die Unterklassen spezifische Attribute und Methoden haben.

Anforderungen:

1. **Die `Mensch`-Klasse**:
    - Soll allgemeine Attribute wie `name`, `alter`, `geschlecht` und eine Methode wie `vorstellen()` haben.
    - Der Konstruktor der `Mensch`-Klasse soll privat sein, damit keine Instanz dieser Klasse direkt erstellt werden kann.
2. **Die Unterklassen**:
    - `Arzt` soll spezifische Attribute wie `spezialisierung` und eine Methode wie `diagnoseStellen()` haben.
    - `Fußballspieler` soll spezifische Attribute wie `position` und eine Methode wie `spiele()` haben.
    - `Koch` soll spezifische Attribute wie `kochrichtung` und eine Methode wie `koche()` haben.
3. **Factory-Methoden in der `Mensch`-Klasse**:
    - Erstellen Sie drei Factory-Methoden in der `Mensch`-Klasse: `arztFactory()`, `fussballspielerFactory()` und `kochFactory()`.
    - Diese Factory-Methoden sollen jeweils die entsprechenden Unterklasseninstanzen (`Arzt`, `Fußballspieler`, `Koch`) erzeugen.
4. **Keine direkte Instanziierung**:
    - Es soll nicht möglich sein, ein `Mensch`, `Arzt`, `Fußballspieler` oder `Koch` direkt zu erstellen. Alle Instanzen müssen über die Factory-Methoden der `Mensch`-Klasse erstellt werden.