1. **Sınıflar ve Nesneler:**

* + Sınıf tanımlama (class)
  + Nesne oluşturma
  + Özellikler (properties)
  + Metodlar

1. **İnşacılar (Constructors):**
   * Başlangıç metodu (init)
2. **Erişim Denetleyicileri (Access Control):**
   * public, private, fileprivate, internal
3. **Miras (Inheritance):**
   * Alt sınıf oluşturma
   * Override edilen metot ve özellikler
4. **Polimorfizm:**
   * Metotların override edilmesi
   * Sınıf ve protokol bazında polimorfizm
5. **Encapsulation:**
   * Özelliklerin ve metotların gizlenmesi
6. **Bileşim (Composition):**
   * Sınıfların birbiri içinde veya birbiriyle ilişkilendirilmesi
7. **Protokoller (Protocols):**
   * Protokol tanımlama
   * Sınıfların protokollerden türetilmesi
8. **Uzantılar (Extensions):**
   * Mevcut tiplere eklemeler yapma
9. **Tip Bağlama (Type Casting):**
   * as, is, as?, as! kullanımları
10. **Delegasyon (Delegation):**
    * Protokoller aracılığıyla delegasyon yapısı
11. **İç İçe Sınıflar (Nested Classes):**
    * Sınıfların başka sınıfların içinde tanımlanması

**1. Sınıflar ve Nesneler**

**Problem-1 :**

Bir kütüphanede, farklı kitap türlerini, yazarlarını ve yayın tarihlerini takip etmek istiyoruz. Kütüphanede bir kitabın adını, yazarını, türünü ve yayın tarihini saklamak ve bir kitap eklediğimizde bu bilgilere erişmek istiyoruz.

**Kod Çözümü Text**

class Kitap {  
    // Özellikler  
    var adi: String  
    var yazar: String  
    var tur: String  
    var yayinTarihi: String  
  
    // Başlangıç (initializer) metodu  
    init(adi: String, yazar: String, tur: String, yayinTarihi: String) {  
        self.adi = adi  
        self.yazar = yazar  
        self.tur = tur  
        self.yayinTarihi = yayinTarihi  
    }  
  
    // Kitap bilgilerini yazdıran metot  
    func bilgileriYazdir() {  
        print("Kitap Adı: \(adi)")  
        print("Yazar: \(yazar)")  
        print("Tür: \(tur)")  
        print("Yayın Tarihi: \(yayinTarihi)")  
    }  
}  
  
// Bir kitap örneği oluşturma  
let kitap1 = Kitap(adi: "1984", yazar: "George Orwell", tur: "Roman", yayinTarihi: "1949")  
  
// Kitap bilgilerini yazdırma  
kitap1.bilgileriYazdir()

**Problem - 2 :**

Bir kütüphanede kitapları takip etmenin yanı sıra kitapların hangi rafta olduğunu, kitapların kütüphanede olup olmadığını (ödünç alınıp alınmadığını) ve kütüphaneye yeni bir raf eklemek istiyoruz. Her rafın belirli bir kapasitesi vardır ve bu rafın kapasitesine göre kitap ekleyebiliriz.

Çözüm:

Bu sorunu çözmek için iki sınıf oluşturacağız: Kitap ve Raf.

* Kitap sınıfı: kitap adı, yazar, tür ve yayın tarihi bilgilerini içerir. Ayrıca kitabın ödünç alınıp alınmadığını kontrol etmek için bir özellik ekleyeceğiz.
* Raf sınıfı: raf numarası, kapasite ve mevcut kitaplar listesi olarak üç özelliğe sahip olacak. Bu sınıfa ayrıca kitap eklemek veya çıkarmak için metotlar ekleyeceğiz.

**Kod Çözümü Text**

class Kitap {  
    var adi: String  
    var yazar: String  
    var tur: String  
    var yayinTarihi: String  
    var oduncAlindiMi: Bool = false  
  
    init(adi: String, yazar: String, tur: String, yayinTarihi: String) {  
        self.adi = adi  
        self.yazar = yazar  
        self.tur = tur  
        self.yayinTarihi = yayinTarihi  
    }  
}  
  
class Raf {  
    var rafNumarasi: Int  
    var kapasite: Int  
    var kitaplar: [Kitap] = []  
  
    init(rafNumarasi: Int, kapasite: Int) {  
        self.rafNumarasi = rafNumarasi  
        self.kapasite = kapasite  
    }  
  
    func kitapEkle(kitap: Kitap) -> Bool {  
        if kitaplar.count < kapasite {  
            kitaplar.append(kitap)  
            return true  
        } else {  
            print("Raf dolu!")  
            return false  
        }  
    }  
  
    func kitapCikar(kitap: Kitap) -> Bool {  
        if let index = kitaplar.firstIndex(where: { $0.adi == kitap.adi }) {  
            kitaplar.remove(at: index)  
            return true  
        } else {  
            print("Kitap rafda bulunamadı!")  
            return false  
        }  
    }  
}  
  
let kitap1 = Kitap(adi: "1984", yazar: "George Orwell", tur: "Roman", yayinTarihi: "1949")  
let raf1 = Raf(rafNumarasi: 1, kapasite: 10)  
  
raf1.kitapEkle(kitap: kitap1)

Bu kod, kitapların hangi rafta olduğunu takip etmemize ve raflardaki kitapları eklememize veya çıkarmamıza olanak tanır. Ayrıca, bir kitap ödünç alındığında oduncAlindiMi özelliğini true yaparak bu durumu takip edebilirsiniz.

**Problem - 3**

Bir otelde, farklı tipte odalar (tek kişilik, çift kişilik, suit) bulunmaktadır. Otelde, oda fiyatlarını, oda kapasitelerini ve oda durumunu (boş mu, dolu mu) yönetmek istiyoruz. Ayrıca, belirli bir tarihte boş olan odaları listelemek ve bir odayı rezerve etmek istiyoruz.

**Problem - 4**

Bir araba galerisinde farklı marka ve modellerde arabalar bulunmaktadır. Her araba modelinin, üretim yılı, fiyat, renk ve mevcut durumu (yeni, ikinci el) gibi bilgilere sahip olduğunu düşünelim. Galeri, bir arabayı satın aldığında veya sattığında, ilgili arabanın envanterden eklenmesi veya çıkarılması gerekiyor. Ayrıca, belirli bir fiyat aralığındaki mevcut arabaları listelemek istiyoruz.

**Problem - 5**

Bir gezegenler arası nakliye şirketiniz var. Bu şirkette, gezegenler arasında yük taşıyan uzay gemileri bulunuyor. Her gemi, belirli bir yük kapasitesine sahip ve sadece belirli tip yükleri taşıyabiliyor (örn. tehlikeli madde, canlı yaratıklar, donmuş gıdalar). Şirket, bir gezegenden diğerine hangi geminin hangi yükleri taşıdığını takip etmek istiyor.

**Problem - 6**

Bir şehirde birden çok hastane bulunmaktadır. Her hastanenin belirli bir kapasitesi, adı ve uzmanlık alanları vardır (örn. kardiyoloji, nöroloji, ortopedi). Aynı zamanda hastaların adı, soyadı, tedavi edildiği hastane ve uzmanlık alanına göre tedavi gördüğü bölüm bilgileri mevcuttur. Hastaların, hangi hastanede hangi bölümde tedavi gördüğünü takip eden bir sistem oluşturmak istiyoruz.

**Sınıflar ve Nesneler:**

* Sınıf tanımlama (class)
* Nesne oluşturma
* Özellikler (properties)
* Metodlar

### **Sınıf Tanımlama (Class)**

Sınıf, bir şablonun veya mavi bir planın oluşturulmasıdır. Swift'te bir sınıf tanımlamak için class anahtar kelimesi kullanılır. Sınıflar, bir nesnenin sahip olacağı özellikleri ve işlevleri tanımlar.

Örnek:

#### Swift

class Araba {  
    var renk: String = "Kırmızı"  
    var model: String = "Sedan"  
}

### **2. Nesne Oluşturma (Object Creation)**

Sınıf bir şablondur, gerçekte kullanılan şey nesnedir. Bir sınıfın örneği alınarak nesne oluşturulur.

Örnek:

#### Swift

let benimArabam = Araba()

Bu kod parçasıyla Araba sınıfından bir nesne oluşturduk.

### **3. Özellikler (Properties)**

Sınıfların özellikleri (ya da bazen fields olarak da adlandırılır), bir sınıfın veya nesnenin durumunu temsil eder. Özellikler, değişkenler veya sabitler olabilir. Yukarıdaki Araba sınıfındaki renk ve model özelliklere örnektir.

Bir nesnenin özelliklerine erişmek için . notasyonunu kullanırız:

#### Swift

print(benimArabam.renk)  // Kırmızı

### **4. Metodlar (Methods)**

Metodlar, bir sınıfa ait işlevlerdir. Bunlar, nesnenin ne yapabileceğini tanımlar. Bir sınıfın içinde tanımlanan işlevlere metod denir.

#### Swift

class Araba {  
    var hiz: Int = 0  
  
    func hizlan() {  
        hiz += 10  
        print("Araba hızlandı. Şu anki hız: \(hiz) km/sa")  
    }  
  
    func yavasla() {  
        hiz -= 10  
        print("Araba yavaşladı. Şu anki hız: \(hiz) km/sa")  
    }  
}  
  
let sporAraba = Araba()  
sporAraba.hizlan()  // Araba hızlandı. Şu anki hız: 10 km/sa  
sporAraba.yavasla()  // Araba yavaşladı. Şu anki hız: 0 km/sa

**3. Solution**

#### Swift

enum OdaTipi {  
    case tekKisilik  
    case ciftKisilik  
    case suit  
}  
  
class Oda {  
    var tip: OdaTipi  
    var fiyat: Double  
    var kapasite: Int  
    var doluMu: Bool = false  
  
    init(tip: OdaTipi, fiyat: Double, kapasite: Int) {  
        self.tip = tip  
        self.fiyat = fiyat  
        self.kapasite = kapasite  
    }  
}  
  
class Otel {  
    var odalar: [Oda] = []  
  
    func bosOdalar() -> [Oda] {  
        return odalar.filter { !$0.doluMu }  
    }  
  
    func odaRezervEt(oda: Oda) -> Bool {  
        if !oda.doluMu {  
            oda.doluMu = true  
            return true  
        } else {  
            print("Oda zaten dolu!")  
            return false  
        }  
    }  
}  
  
let oda1 = Oda(tip: .tekKisilik, fiyat: 100.0, kapasite: 1)  
let oda2 = Oda(tip: .ciftKisilik, fiyat: 150.0, kapasite: 2)  
let oda3 = Oda(tip: .suit, fiyat: 300.0, kapasite: 3)  
  
let otel = Otel()  
otel.odalar = [oda1, oda2, oda3]  
  
otel.odaRezervEt(oda: oda1) // Odayı rezerve eder ve `true` döner.  
  
let mevcutBosOdalar = otel.bosOdalar() // Şu anki boş odaların listesini döndürür.

**6- Solution**

Bu problemi çözmek için üç ana sınıf oluşturacağız: Hastane, Bolum ve Hasta.

1. Hastane: Her hastanenin adı, kapasitesi ve uzmanlık bölümleri gibi bilgileri bulunacak.
2. Bolum: Uzmanlık alanını (örn. kardiyoloji, nöroloji) ve bu bölümde tedavi gören hastaların listesini bulunduracak.
3. Hasta: Hastanın adı, soyadı ve tedavi gördüğü bölümü bulunduracak.

**Swift**

enum UzmanlikAlani {  
    case kardiyoloji  
    case noroloji  
    case ortopedi  
}  
  
class Bolum {  
    var uzmanlik: UzmanlikAlani  
    var hastalar: [Hasta] = []  
  
    init(uzmanlik: UzmanlikAlani) {  
        self.uzmanlik = uzmanlik  
    }  
}  
  
class Hasta {  
    var adi: String  
    var soyadi: String  
    weak var bolum: Bolum?  
  
    init(adi: String, soyadi: String) {  
        self.adi = adi  
        self.soyadi = soyadi  
    }  
}  
  
class Hastane {  
    var adi: String  
    var kapasite: Int  
    var bolumler: [Bolum] = []  
  
    init(adi: String, kapasite: Int) {  
        self.adi = adi  
        self.kapasite = kapasite  
    }  
  
    func hastaEkle(hasta: Hasta, bolume: UzmanlikAlani) {  
        if let bolum = bolumler.first(where: { $0.uzmanlik == bolume }) {  
            bolum.hastalar.append(hasta)  
            hasta.bolum = bolum  
        }  
    }  
}  
  
let kardiyolojiBolumu = Bolum(uzmanlik: .kardiyoloji)  
let hastane = Hastane(adi: "Merkez Hastanesi", kapasite: 1000)  
hastane.bolumler.append(kardiyolojiBolumu)  
  
let hasta1 = Hasta(adi: "Ali", soyadi: "Veli")  
hastane.hastaEkle(hasta: hasta1, bolume: .kardiyoloji)

1. **Constructors**

**Initializers**

1. **Tanımlama**: Başlatıcılar, sınıf veya yapısal tipin bir örneği oluşturulduğunda çalışır ve özelliklerin başlangıç değerlerini ayarlar.
2. **Sözdizimi**: Başlatıcılar, init anahtar kelimesiyle tanımlanır.

#### Swift

class Kisi {  
    var adi: String  
    var yasi: Int  
  
    init(adi: String, yasi: Int) {  
        self.adi = adi  
        self.yasi = yasi  
    }  
}

**Nesne Oluşturma**: Yukarıda tanımlanan Kisi sınıfından bir örneği oluştururken başlatıcıya gerekli değerleri sağlamamız gerekiyor:

**Swift**

let ali = Kisi(adi: "Ali", yasi: 25)

1. **Varsayılan Başlatıcılar**: Eğer bir sınıf ya da yapısal tip için başlatıcı tanımlamazsanız ve tüm özelliklere varsayılan bir değer verirseniz, Swift otomatik olarak bir varsayılan başlatıcı oluşturur.

**Swift**

class Urun {  
    var isim: String = ""  
    var fiyat: Double = 0.0  
}  
  
let urun = Urun()  // Otomatik olarak oluşturulan varsayılan başlatıcıyı kullanıyor

**Özellik Değerleri ve Başlatıcılar**: Başlatıcı içerisinde, özelliklere doğrudan değer atanabilir veya başka bir metot çağrılarak atanabilir.

#### Swift

class Otomobil {  
    var model: String  
    var yil: Int  
  
    init() {  
        self.model = "Sedan"  
        self.yil = yiliniAl()  
    }  
  
    func yiliniAl() -> Int {  
        let tarih = Date()  
        let takvim = Calendar.current  
        return takvim.component(.year, from: tarih)  
    }  
}  
  
let otomobil = Otomobil()  // Model "Sedan" olarak atanır ve yil mevcut yıl olacaktır

### 1. Başlatıcılar ve Miras:

Swift'te, bir sınıf başka bir sınıftan türetildiğinde (miras alındığında), türetilen sınıfın tüm özelliklerine erişim hakkı vardır. Ancak türetilen sınıfın, miras aldığı sınıfın başlatıcılarını otomatik olarak miras almadığını belirtmek önemlidir. Bunun nedeni, türetilen sınıfın yeni özelliklere sahip olması ve bu yeni özelliklerin başlatılması gerekliliğidir.

### 2. Başlatıcı Zincirleme (Initializer Chaining):

Bir sınıfın birden fazla başlatıcısı varsa, bir başlatıcı diğer başlatıcıyı çağırabilir. Bu, self.init ifadesi kullanılarak yapılır. Bu, başlatıcıların yeniden kullanılmasına ve kodun daha kuru (DRY - Don't Repeat Yourself) olmasına yardımcı olur.

### 3. Başlatıcıların Özellik Değerlerini Ayarlama:

Başlatıcılar, bir sınıfın veya yapının özelliklerinin ilk değerlerini ayarlamak için kullanılır. Swift'te, tüm özelliklerin bir değere sahip olmasını garantilemek için başlatıcının, tüm özelliklere değer atamadan önce bitmesi gerekmektedir.

### 4. Zorunlu Başlatıcılar (Required Initializers):

Eğer bir sınıfın başlatıcısının her zaman türetilen bir sınıf tarafından uygulanması gerekiyorsa, required anahtar kelimesi ile başlatıcı tanımlanabilir. Bu, türetilen sınıfların bu başlatıcıyı sağlamasını zorunlu kılar.

**Swift**

class Kitap {  
    var isim: String  
    required init(isim: String) {  
        self.isim = isim  
    }  
}  
  
class Roman: Kitap {  
    var yazar: String  
    init(isim: String, yazar: String) {  
        self.yazar = yazar  
        super.init(isim: isim)  
    }  
  
    required init(isim: String) {  
        self.yazar = "Bilinmiyor"  
        super.init(isim: isim)  
    }  
}

Daralt

Bu örnekte, Kitap sınıfında bir required başlatıcı bulunmaktadır, bu nedenle Roman sınıfı bu başlatıcıyı uygulamak zorundadır.

Bu konseptler, Swift'te başlatıcının rolünü ve nasıl çalıştığını anlamak için oldukça önemlidir. Başlatıcılar, nesnenin doğru bir durumda ve beklenen davranışı gösterebilmesi için özelliklerin doğru bir şekilde ayarlanmasını sağlar. Bu, kodun güvenli ve hata olasılığını azaltan bir şekilde çalışmasını garantiler.

**Çözümlü Problemler**

**Soru 1-** Bir gezegenler sistemi simülasyonunu oluşturmak istiyoruz. Bu simülasyonda gezegenlerin ismi, çapı, yüzey sıcaklığı ve kaçıncı sıraya sahip olduğunu (Güneşe olan uzaklığına göre) bilmek istiyoruz.

Çözüm :

#### Swift

class Gezegen {  
    var isim: String  
    var cap: Double  // km cinsinden  
    var yuzeySicakligi: Double  // Celsius cinsinden  
    var sirasi: Int  
  
    init(isim: String, cap: Double, yuzeySicakligi: Double, sirasi: Int) {  
        self.isim = isim  
        self.cap = cap  
        self.yuzeySicakligi = yuzeySicakligi  
        self.sirasi = sirasi  
    }  
  
    func bilgiGoster() {  
        print("\(isim), \(cap) km çapında, yüzey sıcaklığı \(yuzeySicakligi)°C olan ve Güneşe olan sırası \(sirasi) olan bir gezegendir.")  
    }  
}  
  
let mars = Gezegen(isim: "Mars", cap: 6779, yuzeySicakligi: -62, sirasi: 4)  
mars.bilgiGoster()

Bu soruya cevap vermek için başlatıcının (init) rolünü ve neden önemli olduğunu inceleyelim.

Başlatıcı (init), bir sınıfın nesnesinin oluşturulduğu an çalıştırılan özel bir metodtur. Temel görevi, sınıfın özelliklerini başlangıç değerleriyle başlatmaktır. Bazı durumlarda, özelliklerin doğru bir şekilde başlatılması olmazsa, nesnenin beklenen davranışı göstermeme riski vardır.

Yukarıdaki Gezegen sınıfına dönersek:

1. **Her Gezegen Farklıdır**: Gezegen sınıfındaki her özellik (isim, cap, yuzeySicakligi, sirasi) gezegenlere özgüdür ve her gezegen için farklı değerlere sahip olabilir. Bu, her bir gezegen nesnesi oluşturulduğunda bu özelliklere özel değerler sağlamamız gerektiği anlamına gelir.
2. **Garanti**: Başlatıcıyı kullanarak, bir Gezegen nesnesinin her zaman gerekli bilgilerle oluşturulduğundan emin olabiliriz. Başlatıcı olmadan, bazı değerleri atamayı unutabiliriz, bu da hatalı veya beklenmedik sonuçlara yol açabilir.
3. **Esneklik**: Gelecekte, Gezegen sınıfına yeni özellikler eklemek istediğimizde, başlatıcının bu özellikleri nasıl başlatacağını belirtmemiz gerekecektir. Bu, tüm nesnelerin doğru bir şekilde başlatıldığından emin olmamıza yardımcı olur.
4. Özetle, init metodunu kullanarak, sınıfın nesnelerinin daima geçerli ve beklenen durumda oluşturulmasını sağlamış oluruz. Bu, nesne yönelimli programlamada kritik bir konsepttir ve nesnelerin doğru bir şekilde başlatılmasını garanti altına alır.

**Soru - 2 :** Bir bankada müşteri hesapları için bir sistem tasarlamak istiyoruz. Her müşteri hesabı, hesap sahibinin adı, hesap numarası ve mevcut bakiye bilgilerini içermelidir. Önemli olan şu ki, her hesap benzersiz bir hesap numarasına sahip olmalı ve bu hesap numarası oluşturulduğunda otomatik olarak atanmalı. Ayrıca, bir müşteri hesabı oluşturulduğunda bakiye 0 olarak başlamalıdır.

**Çözüm**:

#### Swift

class BankaHesabi {  
    var sahibi: String  
    var hesapNumarasi: Int  
    var bakiye: Double = 0.0  
    static var sonrakiHesapNumarasi: Int = 1000  
  
    init(sahibi: String) {  
        self.sahibi = sahibi  
        self.hesapNumarasi = BankaHesabi.sonrakiHesapNumarasi  
        BankaHesabi.sonrakiHesapNumarasi += 1  
    }  
  
    func bilgiGoster() {  
        print("\(sahibi) adlı kişiye ait \(hesapNumarasi) numaralı hesapta \(bakiye) TL bulunmaktadır.")  
    }  
}  
  
let ahmetHesap = BankaHesabi(sahibi: "Ahmet")  
ahmetHesap.bilgiGoster()  // Ahmet adlı kişiye ait 1000 numaralı hesapta 0.0 TL bulunmaktadır.  
  
let ayseHesap = BankaHesabi(sahibi: "Ayşe")  
ayseHesap.bilgiGoster()  // Ayşe adlı kişiye ait 1001 numaralı hesapta 0.0 TL bulunmaktadır.

Bu örnekte, init fonksiyonunun önemini şu şekillerde görebiliriz:

1. **Benzersiz Hesap Numarası**: BankaHesabi sınıfı içinde statik bir özellik olan sonrakiHesapNumarasi ile her yeni hesap için benzersiz bir hesap numarası oluşturduk. Başlatıcı, her yeni nesne oluşturulduğunda bu numarayı otomatik olarak artırır.
2. **Başlangıç Bakiyesi**: Başlatıcı, her yeni hesabın bakiyesini otomatik olarak 0 TL olarak başlatır. Bu, tüm müşteri hesaplarının doğru bir başlangıç bakiyesiyle oluşturulduğundan emin olmamıza yardımcı olur.

Bu tür senaryolarda, başlatıcıları kullanarak sınıf özelliklerini otomatik olarak ayarlamak ve belirli kurallara uymak esastır. Bu, kodun doğruluğunu ve tutarlılığını sağlar.

**Soru -3 :** Bir e-ticaret uygulamasında kullanıcı siparişini tamamladığında bir sipariş numarası oluşturulmalıdır. Bu sipariş numarası her kullanıcı için benzersiz olmalıdır. Ayrıca, siparişin tarihi de otomatik olarak oluşturulmalıdır.

#### Swift

import Foundation  
  
class Siparis {  
    var kullaniciAdi: String  
    static var sonSiparisNumarasi: Int = 100000  
    var siparisNumarasi: Int  
    var siparisTarihi: Date  
  
    init(kullaniciAdi: String) {  
        self.kullaniciAdi = kullaniciAdi  
  
        // Benzersiz sipariş numarası oluşturma  
        self.siparisNumarasi = Siparis.sonSiparisNumarasi  
        Siparis.sonSiparisNumarasi += 1  
  
        // Şu anki tarihi al  
        self.siparisTarihi = Date()  
    }  
  
    func siparisBilgisiGoster() {  
        let dateFormatter = DateFormatter()  
        dateFormatter.dateStyle = .short  
        dateFormatter.timeStyle = .short  
        print("\(kullaniciAdi) adlı kullanıcının \(siparisNumarasi) numaralı siparişi \(dateFormatter.string(from: siparisTarihi)) tarihinde oluşturuldu.")  
    }  
}  
  
let emreSiparis = Siparis(kullaniciAdi: "Emre")  
emreSiparis.siparisBilgisiGoster()    
  
let burcuSiparis = Siparis(kullaniciAdi: "Burcu")  
burcuSiparis.siparisBilgisiGoster()

Bu örnekte, init metodunun önemli rolleri var:

1. **Benzersiz Sipariş Numarası**: Başlatıcı, statik bir değişken olan sonSiparisNumarasi kullanarak her yeni sipariş için benzersiz bir sipariş numarası oluşturur. Bu, siparişler arasında ayrım yapabilmek için önemlidir.
2. **Otomatik Sipariş Tarihi**: Başlatıcı, siparişin oluşturulduğu anın tarihini ve saatini otomatik olarak kaydeder.

Başlatıcılar, bu tür durumlarda karmaşık iş kurallarını uygulamak ve nesnelerin tutarlı bir şekilde oluşturulmasını sağlamak için oldukça kullanışlıdır. Bu örnekte, her siparişin doğru bir şekilde oluşturulması için init metodunu kullanarak belirli kuralları zorunlu hale getiriyoruz.

**Soru - 3 :** Bir oyun geliştirelim. Bu oyunda, oyuncunun karakterleri olacak ve her karakterin belirli bir enerji seviyesi ve gücü olacak. Oyuncu, bir karakter yaratırken sadece karakterin adını belirtmelidir. Ancak, oyuncu aynı adla başka bir karakter yaratmaya çalışırsa, bu karakterin farklı bir varyasyonla oluşturulmasını istiyoruz (örneğin, "Mario" adında bir karakter varsa, aynı adla başka bir karakter yaratılırsa bu "Mario-1" olmalı).

#### Swift

class Karakter {  
    var ad: String  
    var enerji: Int = 100  
    var guc: Int = 10  
    static var olusturulanKarakterler: [String: Int] = [:]  
  
    required init(ad: String) {  
        if let varolanSayi = Karakter.olusturulanKarakterler[ad] {  
            Karakter.olusturulanKarakterler[ad] = varolanSayi + 1  
            self.ad = "\(ad)-\(varolanSayi)"  
        } else {  
            Karakter.olusturulanKarakterler[ad] = 1  
            self.ad = ad  
        }  
    }  
  
    convenience init(ad: String, enerji: Int, guc: Int) {  
        self.init(ad: ad)  
        self.enerji = enerji  
        self.guc = guc  
    }  
}  
  
let mario = Karakter(ad: "Mario")  
print(mario.ad)  // Mario  
  
let mario2 = Karakter(ad: "Mario")  
print(mario2.ad)  // Mario-1  
  
let mario3 = Karakter(ad: "Mario", enerji: 120, guc: 15)  
print(mario3.ad)  // Mario-2  
print(mario3.enerji)  // 120  
print(mario3.guc)  // 15

Bu örnekte:

* olusturulanKarakterler statik bir sözlüğü kullanarak hangi karakter adlarının kaç kez oluşturulduğunu takip ediyoruz.
* required init(ad: String) başlatıcısı, verilen adın daha önce oluşturulup oluşturulmadığını kontrol eder ve buna göre karakterin adını belirler.
* convenience init(ad: String, enerji: Int, guc: Int) başlatıcısı, karakterin enerji ve gücünü de ayarlamak için kullanılır ve bu başlatıcı, ana başlatıcı olan required init(ad: String)'ı kullanarak adla ilgili işlemi gerçekleştirir.

Bu tasarım, oyuncunun aynı adla birden fazla karakter yaratmasına izin verirken, her karakterin benzersiz bir ada sahip olmasını garantiler. Aynı zamanda, karakterin enerji ve gücü gibi diğer özelliklerinin de ayarlanabilmesine izin verir.

**İleri Seviye Problemler**

**1-**Bir bankada hesap açmak için bir sistem tasarlayacağız. Her müşteri için TC kimlik numarası, isim, soyisim ve başlangıç bakiyesi olmalıdır. Ancak, başlangıç bakiyesi belirtilmediğinde otomatik olarak 0 olmalıdır. Müşteri TC kimlik numarası her zaman 11 haneli olmalıdır. Eğer 11 haneli değilse, bir hata mesajı gösterilmelidir.

**Çözüm**:

#### Swift

class Musteri {  
    var tcKimlik: String {  
        didSet {  
            guard tcKimlik.count == 11 else {  
                print("Hatalı TC kimlik numarası. 11 haneli olmalıdır.")  
                tcKimlik = oldValue  
                return  
            }  
        }  
    }  
    var isim: String  
    var soyisim: String  
    var bakiye: Double  
  
    init(tcKimlik: String, isim: String, soyisim: String) {  
        self.tcKimlik = tcKimlik  
        self.isim = isim  
        self.soyisim = soyisim  
        self.bakiye = 0.0  
  
        guard tcKimlik.count == 11 else {  
            fatalError("Hatalı TC kimlik numarası. 11 haneli olmalıdır.")  
        }  
    }  
  
    convenience init(tcKimlik: String, isim: String, soyisim: String, bakiye: Double) {  
        self.init(tcKimlik: tcKimlik, isim: isim, soyisim: soyisim)  
        self.bakiye = bakiye  
    }  
}  
  
let ahmet = Musteri(tcKimlik: "12345678901", isim: "Ahmet", soyisim: "Yılmaz")  
print(ahmet.bakiye)  // 0.0  
  
let burak = Musteri(tcKimlik: "123456789012", isim: "Burak", soyisim: "Kara", bakiye: 500.0)    
// "Hatalı TC kimlik numarası. 11 haneli olmalıdır." mesajını verecek ve program sonlanacak

**2-**Bir müzik uygulaması oluşturuyorsunuz. Kullanıcılar bu uygulamada şarkı oluşturabilirler. Her şarkı için isim, süre ve sanatçı bilgisi vardır. Şarkılar varsayılan olarak 3 dakika süre ile oluşturulmalıdır, ama bu süreyi kullanıcılar özelleştirebilmelidir. Aynı sanatçının birden fazla şarkısı olabileceği için, aynı sanatçı ismi için şarkı oluşturulduğunda bu bilgiyi kullanarak şarkının sanatçı bilgisini otomatik doldurmak istiyoruz.

**3-**Bir e-ticaret platformu oluşturuyorsunuz. Müşteriler, platform üzerinden ürün satın alabilirler. Her ürün için isim, kategori ve fiyat bilgisi bulunmaktadır. Aynı kategorideki ürünler için varsayılan bir indirim oranı uygulanacaktır. Örneğin, "Elektronik" kategorisindeki ürünler için %10, "Giyim" için %5 indirim uygulanabilir. Fakat bazı ürünler için özel indirimler de tanımlanabilmelidir.

**Çözüm:**

1. **static** anahtar kelimesi Swift'te sınıf, yapı ve enum tanımlamalarında kullanılır. Bir özellik veya metoda **static** anahtar kelimesi eklenirse, bu özellik veya metod, sınıfın veya yapılanmanın bir örneği oluşturulmadan doğrudan sınıf veya yapı üzerinden erişilebilir hale gelir. Bu tür özelliklere veya metotlara "sınıf özellikleri" veya "sınıf metotları" denir.
2. **Veri Yapısı Oluşturma**

Ürünleri temsil eden bir Urun sınıfı oluşturduk. Bu sınıfın özellikleri arasında isim, kategori ve baslangicFiyati bulunmaktadır.

1. **Varsayılan İndirim Oranları**

Varsayılan indirim oranlarını saklamak için Urun sınıfı içinde statik bir sözlük oluşturduk. Bu sözlük, kategori isimlerini anahtar ve indirim oranlarını değer olarak tutar.

#### Swift

static let varsayilanIndirimOranlari: [String: Double] = ["Elektronik": 0.10, "Giyim": 0.05]

**İndirim Oranını Hesaplayan Özellik**

Her ürünün kendi kategorisine göre indirim oranını elde etmek için indirimOrani adında bir hesaplanan özellik (computed property) kullandık. Bu özellik, ürünün kategorisini varsayilanIndirimOranlari sözlüğünde arar ve eğer bir değer bulursa onu, bulamazsa 0.0 değerini döndürür.

#### Swift

var indirimOrani: Double {    return Urun.varsayilanIndirimOranlari[kategori] ?? 0.0}

**Son Fiyatı Hesaplayan Özellik**

Ürünün son fiyatını, başlangıç fiyatından indirim oranını çıkararak hesaplayan bir diğer hesaplanan özellik oluşturduk.

#### Swift

var sonFiyat: Double {    return baslangicFiyati \* (1 - indirimOrani)}

**Özel İndirim Oranları**

Ürün oluşturulurken, ozelIndirimOrani adlı bir parametre ile özel bir indirim oranı belirtilebilir. Eğer bu parametre verilirse, bu oran varsayılan indirim oranını geçersiz kılar. Bu, başlatıcı (init) içerisinde yapılır.

#### Swift

init(isim: String, kategori: String, fiyat: Double, ozelIndirimOrani: Double? = nil) {    self.isim = isim    self.kategori = kategori    self.baslangicFiyati = fiyat            if let ozelIndirim = ozelIndirimOrani {        Urun.varsayilanIndirimOranlari[kategori] = ozelIndirim    }}

**Kodun Son Hali**

**Swift**

class Urun {  
    static let varsayilanIndirimOranlari: [String: Double] = [  
        "Elektronik": 0.10,  
        "Giyim": 0.05  
    ]  
  
    var isim: String  
    var kategori: String  
    var baslangicFiyati: Double  
    var indirimOrani: Double {  
        return Urun.varsayilanIndirimOranlari[kategori] ?? 0.0  
    }  
    var sonFiyat: Double {  
        return baslangicFiyati \* (1 - indirimOrani)  
    }  
  
    init(isim: String, kategori: String, fiyat: Double, ozelIndirimOrani: Double? = nil) {  
        self.isim = isim  
        self.kategori = kategori  
        self.baslangicFiyati = fiyat  
          
        if let ozelIndirim = ozelIndirimOrani {  
            Urun.varsayilanIndirimOranlari[kategori] = ozelIndirim  
        }  
    }  
}  
  
let urun1 = Urun(isim: "Laptop", kategori: "Elektronik", fiyat: 5000.0)  
print(urun1.sonFiyat)  // 4500.0  
  
let urun2 = Urun(isim: "Gömlek", kategori: "Giyim", fiyat: 200.0, ozelIndirimOrani: 0.2)  
print(urun2.sonFiyat)  // 160.0

**4-**Bir hava durumu uygulamanız var. Bu uygulamada, kullanıcılar farklı şehirlerdeki hava durumunu sorgulayabilirler. Her şehir için sıcaklık, nem ve rüzgar bilgisi bulunmaktadır. Kullanıcılar, istedikleri şehir için sıcaklık bilgisi ekleyebilir veya mevcut sıcaklık bilgisini güncelleyebilirler. Ancak, sıcaklık bilgisi belirli bir aralığın (örneğin, -50 ile 50) dışına çıkmamalıdır.