Dart Notlarım **03.09.2024**

- Genel Class yapısı(Yapıcı metod, get,set, late anahtar kelimesi, private erişim belirteci, arrow function, pratik yapıcı fonksiyon yazımı)

```
void main() {
 User1 adem = User1("Adem", 100, age: 23,surname: "Kayabay");
 // set metodu çağrısı atama yapar gibi yazılır
 adem.setId="123456789";
 User2 yahya = User2("Yahya", 200,city: 'İstanbul');

 print("Kullanıcı ismi: ${adem.name}\nKullanıcı parası:
${adem.money}\nKullanıcı yaşı:  ${adem.age}\nKullanıcı şehir:
${adem.city}\nKullanıcı kodu: ${adem.userCode}");
 print("-" * 50);
 print("Kullanıcı ismi: ${yahya.name}\nKullanıcı parası:
${yahya.money}\nKullanıcı yaşı:  ${yahya.age}\nKullanıcı şehir:
${yahya.city}\nKullanıcı kodu: ${yahya.userCode}");
 print("-" * 50);

 //adem.name = "deneme";
 //adem.surname = "blabla";


 print(adem.getId());
 print(adem.get_Id_Arrow_Function_Version);
}

class User1{
 // late anahtar kelimesi bir değerin daha sonradan geleceğini
belirtir
 // bazı sorunları çözer ama dikkatli kullanmak gerekir
 // örnek vermek gerekirse halihazırda atanmış bir late değere
tekrardan bir değer atamaya kalkarsak editör uyarı vermez ama
derlenince hata verir
 // late kullanmazsak bu hatanın farkına derlemeden varabiliriz
 late final String name;
 final String surname;
 late final double money;
 late final int? age;
 late final String? city;
 late final String userCode;
```

```dart
// _ karakteri private erişim belirtecini temsil eder. Fakat sadece file private haline getirir.
 // Tam bir gizlilik için sınıfları ayrı bir model klasörüne almak gerekir.
 // bu tip değerlere ileride dışardan müdahale etmek için get set metodları yazılmalıdır
 late final String _id;

 // Constructor - Yapıcı Fonksiyon
 // {} içerisinde yazılanlar opsiyonel parametrelerdir
 User1(String name, double money, {int age = 22, String? city, required this.surname}){
   this.name = name;
   this.money = money;
   this.age = age;
   this.city = city;

   // city eğer null ise default olarak 'ist' değerini alır
   userCode = (city ?? 'ist') + name;
 }


 String getId(){
   return _id;
 }

 String get get_Id_Arrow_Function_Version => _id;

 set setId(String id) => this._id = id;
}

class User2{
 // late anahtar kelimesi bir değerin daha sonradan geleceğini belirtir
 final String name;
 final double money;
 final int age;
 final String? city;

 late final String userCode;

 // construactor tanımlamasına diğer bir alternatif
 User2(this.name,  this.money, {this.age = 22, this.city}){
   userCode = (city ?? 'ist') + name;
 }
}
```

- Extends(Türetme), abstract sınıflar

```
void main(){
 Customer customer1 = Customer(userName: "Adem");
 Admin admin1 = Admin(userName: "Yahya");
 print("Kullanıcı Adı: ${customer1.userName}\nKullanıcı Tipi:
${customer1.userType}");
 print("-"*25);
 print("Kullanıcı Adı: ${admin1.userName}\nKullanıcı Tipi:
${admin1.userType}");
}
/*
abstract sınıflar, Dart dilinde belirli bir sınıfın soyut bir
yapıya sahip olmasını sağlamak için kullanılır.
abstract bir sınıf, tek başına bir nesne oluşturmak için
kullanılamaz; bunun yerine, diğer sınıfların türetilmesi ve
bu sınıfların temelini oluşturması için kullanılır.

Temel özellikleri:
1)Soyutlama: abstract sınıflar, bir yapının genel tasarımını
tanımlar, ancak bazı metodlar ve özellikler tam olarak
tanımlanmamış olabilir.Bu metodlar, abstract olarak
işaretlenir ve alt sınıflar tarafından uygulanmak zorundadır.
2)Nesne Oluşturulamaz: abstract bir sınıfın kendisinden
doğrudan nesne oluşturulamaz.
3)Alt Sınıflar: abstract sınıfların asıl amacı, diğer
sınıfların bu sınıfı miras alarak (inheritance) kendilerine
özel işlevleri eklemesidir. Alt sınıflar, abstract sınıfta
tanımlanmış metodları ve özellikleri uygulamak zorundadır.
*/
abstract class IUser{
 late String userName;
 late final String userType;
 IUser(this.userName, this.userType){}

 //Ortak metod
 String get getUserType => userType;
}

class Customer extends IUser{
 Customer({required String userName}) :
super(userName,'customer');
}

class Admin extends IUser{
 Admin({required String userName}) : super(userName,'admin');
}
```

- Implements, override

```
// Arayüz (Interface)
// implement ile bir sınıf türetildiğinde türetilen
sınıfın temel iskelet olarak bire bir aynısı olur
// fakat soyut sınıfın içerisindeki metodları tekrar
override edip kendisine implement etmelidir
// extend ile temel farkı budur extend edilen
sınıflardaki metodlar direkt olarak kullanılabilir burda
override edilmelidir
abstract class Animal {
 void makeSound(); // Soyut metot
}

// Sınıfın arayüzü implement etmesi
class Dog implements Animal {
 @override
 void makeSound() {
   print('hav!');
 }
}

class Bee implements Animal {
 @override
 void makeSound() {
   print('Wwzzzzzz!');
 }
}

void main() {
 var dog = Dog();
 var bee = Bee();
 dog.makeSound(); // Output: hav!
 bee.makeSound(); // Output: Wwzzzzzz!
}
```

Not:

-Interface(Implements), polymorphism'in bir türüdür ve polymorphism'in genel kavramını kapsayan üst bir kavram değildir. Polymorphism'in farklı uygulanış biçimidir.

-Override, bir alt sınıfta üst sınıfta tanımlanmış bir metodun veya property'sinin yeniden tanımlanmasıdır. Bu işlem, alt sınıfın, üst sınıftaki metodun davranışını kendi ihtiyaçlarına göre değiştirmesine olanak tanır.

- Enum, Extension

```dart
void main() {
 final customerMouse = Mouses.a4;
 print(customerMouse.index);
 print(customerMouse.name);

 if (customerMouse == Mouses.a4) {}

 if (customerMouse.name == 'a4') {}

 if (customerMouse.isCheckName('a4')) {
   print('aa');
 }
}

// Enum, belirli bir kümedeki sabitleri tanımlamak için kullanılan bir veri türüdür.
// Enum'lar, genellikle belirli bir grup sabit değer için anlamlı isimler kullanarak bu değerleri daha anlamlı ve yönetilebilir hale getirir.
enum Mouses {
 magic,
 apple,
 logitech,
 a4,
}

// Extension'lar, mevcut sınıflara ek işlevsellik kazandırmak için esnek ve genişletilebilir bir yaklaşımdır.
extension MousesSelectedExtension on Mouses {
 bool isCheckName(String name) {
   return this.name == name;
 }
}
```

- Operator Overloading

```
void main(){
Bank moneyBank1 = Bank(50);
Bank moneyBank2 = Bank(50);

//operatör overload edildiğinde, operatöre sağ taraftan gelen nesne,
//yani + operatöründen sonraki nesne, metot içinde tanımlanan parametreye atanır.
//moneyBank1 this anahtar kelimesiyle erişilen nesnedir (sol taraftaki nesne).
//moneyBank2 ise newBank olarak operatöre geçirilen nesnedir (sağ taraftaki nesne).
print(moneyBank1 + moneyBank2);
}

class Bank{
 final int money;

 Bank(this.money);

   //Operator overloading, sınıflar için belirli operatörlerin anlamını değiştirme işlemidir.
  //Bu özellik, sınıf özelinde operatörlerin nasıl çalışacağını özelleştirmemizi sağlar.
 int operator +(Bank newBank){
   return this.money + newBank.money;
 }
}
```