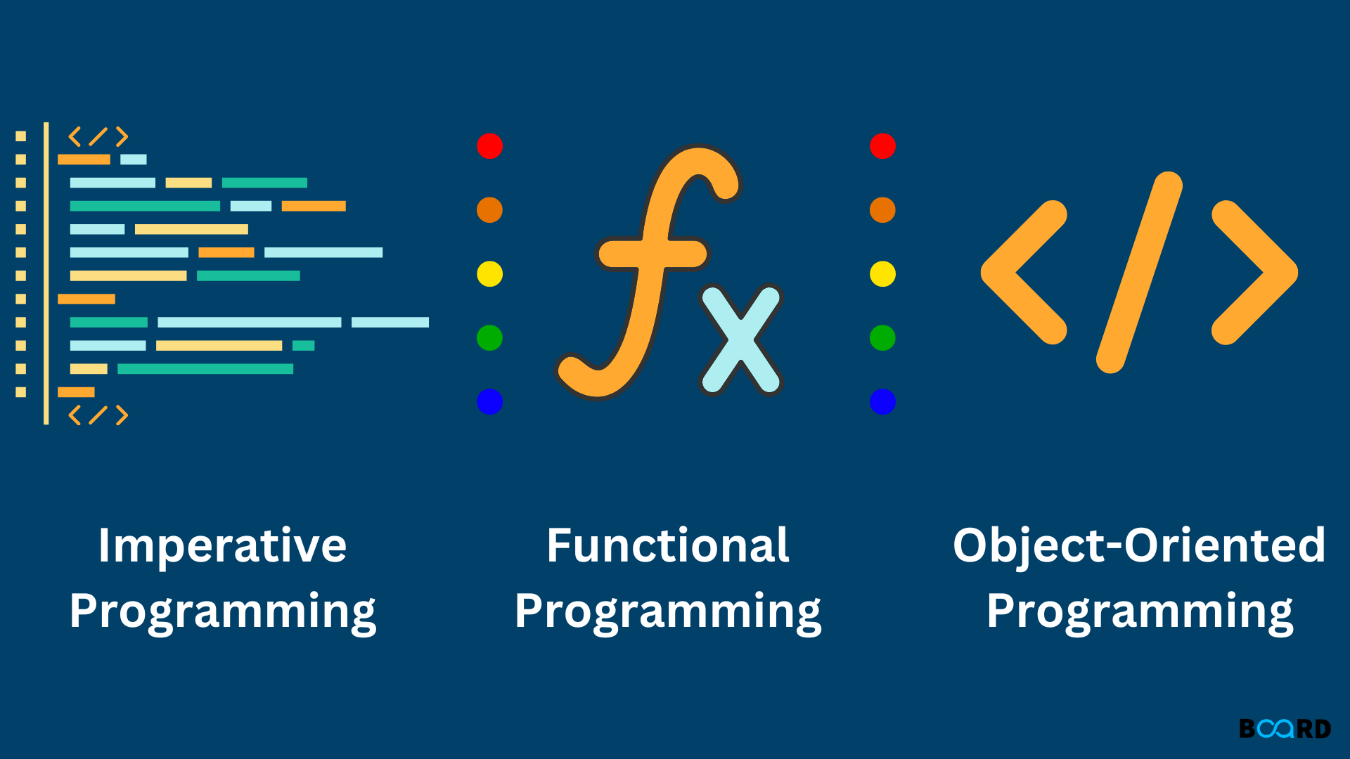
**NESNE YÖNELİMLİ PROGRAMLAMA:**

**TEMEL KAVRAMLAR:**

PROGRAMLAMA PARADİGMALARI NEDİR?



Programlama paradigmaları, bilgisayar programlarını yazmak, düzenlemek ve yönetmek için kullanılan farklı yaklaşımları ve yöntemleri ifade eder. Bu paradigmalar, programlama dillerinin temel felsefelerini ve nasıl çalıştıklarını şekillendirir. Programlama paradigmaları, yazılım geliştirme sürecinde kullanılan kavramları, mantığı ve yapıları belirler.

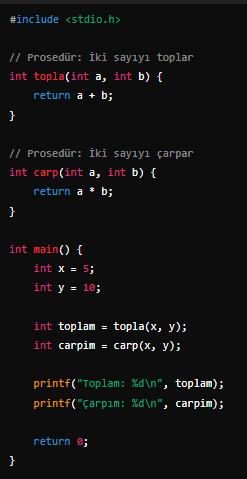
Bir alandaki paradigma, o alanın nasıl anlaşıldığını, neyin önemli olduğunu ve hangi yöntemlerin geçerli olduğunu belirler.

**Yapısal (Prosedürel) Programlama:**

Yapısal programlama, 1960'larda ortaya çıkmış ve programların karmaşıklığını azaltmak amacıyla geliştirilmiştir. Bu paradigmanın temel amacı, programı daha küçük, yönetilebilir alt parçalara bölmektir. Bu alt parçalar genellikle fonksiyonlar veya prosedürler şeklinde organize edilir.

* **Tanım:** Bu paradigmada program, bir dizi prosedür veya fonksiyon olarak yapılandırılır. Her prosedür bir işlevi yerine getirir ve program bu işlevler arasındaki etkileşimlerle çalışır.
* **Örnek Diller:** C, Pascal, Fortran.
* **Özellik:** Kodun modüler hale getirilmesi ve yeniden kullanılabilirliğinin artırılması amaçlanır. Yukarıdan aşşağı doğru kod çalışır
  + **Sıralı yürütme:** Kod, yukarıdan aşağıya doğru bir sırada çalışır.
  + **Dallanma:** if-else, switch gibi yapılarla karar verme işlemleri gerçekleştirilir.
  + **Döngüler:** for, while gibi yapılarla tekrarlayan işlemler gerçekleştirilir.
  + **Fonksiyonlar:** Kodun belirli bölümleri fonksiyonlar halinde düzenlenir.
  + Veri ve işlevler (fonksiyonlar) birbirinden ayrıdır. Veriler, global veya yerel değişkenler olarak saklanır ve fonksiyonlar, bu verilere erişerek işlem yapar.
  + Yapısal programlama, veri işleme ve durumu değiştirme üzerine yoğunlaşır.

Özellikle küçük ve orta ölçekli projelerde çok etkilidir. Modülerlik, kontrol yapıları ve fonksiyonların doğru kullanımı sayesinde programlar daha okunabilir.



OOP ile KIYASLAMA:

Fonksiyonlar veya prosedürler, belirli bir işlevi yerine getiren modüler bileşenler olarak kullanılır. Ancak bu fonksiyonlar, nesne yönelimli programlamadaki gibi bir sınıfın parçası değil, bağımsız yapılardır.

Yapısal paradigma , sınıflar ve nesneler yerine, fonksiyonlar ve prosedürler etrafında döner. Veri ve fonksiyonlar birbirinden ayrı tutulur ve programlama daha çok prosedürel bir akış takip eder.

**Fonksiyonel Programlama (Functional Programming):**

* Programlar, matematiksel fonksiyonlar gibi çalışır ve değişmez (immutable) veriler kullanır.
* Fonksiyonlar birinci sınıf vatandaş olarak kabul edilir; yani fonksiyonlar değişkenlere atanabilir, argüman olarak geçilebilir ve fonksiyon olarak geri döndürülebilir.
* Fonksiyonel programlama, veri dönüşümü ve manipülasyonu üzerine yoğunlaşır. Yan etkisiz (side-effect free) fonksiyonlar önemli bir parçasıdır.
* Örnek: Haskell, Lisp.

Bir fonksiyon aynı değer almasına rağmen farklı sonuçlar üretiyorsa immutable değildir. Bir fonksiyonun değeri başka bir fonksiyona bağlı ise ve diğer fonksiyon rastgelelik veya sürekli farklı bir başlangıç değeri içeriyorsa immutable olmaz.

* Fonksiyonel programlama, yan etkisiz olmayı teşvik eder. Yan etkiler, bir fonksiyonun dış dünyayı değiştirmesi veya dış dünya tarafından değiştirilmesi anlamına gelir.
* Yan etkisiz fonksiyonlar, test edilebilirlik ve öngörülebilirlik açısından avantaj sağlar.
* Fonksiyonel programlama, döngüler yerine genellikle rekürsiyonu kullanır. Rekürsiyon, bir fonksiyonun kendini çağırdığı durumlardır.

Fonksiyonel programlama, veri ve işlevlerin ayrı tutulmasını teşvik eder. Bu, nesne yönelimli programlamadaki gibi veriyi saklayan ve işleyen yapıların (örneğin sınıfların) aksine, işlevlerin veriyi bağımsız olarak işlediği anlamına gelir.

**Mantıksal Programlama:**

* **Tanım:** Bu paradigma, mantıksal çıkarım ve kurallar üzerine kuruludur. Programlama, belirli bir problem için kurallar ve olgular tanımlamak ve bu kurallardan mantıksal sonuçlar çıkarma üzerine odaklanır.
* **Örnek Diller:** Prolog, Datalog.
* **Özellik:** Özellikle yapay zeka ve uzman sistemler gibi alanlarda kullanılır.

metin, ekran görüntüsü, yazı tipi, tasarım içeren bir resim

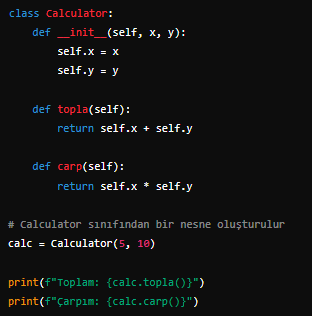
Açıklama otomatik olarak oluşturuldu

Şekil 1:Prolog

…

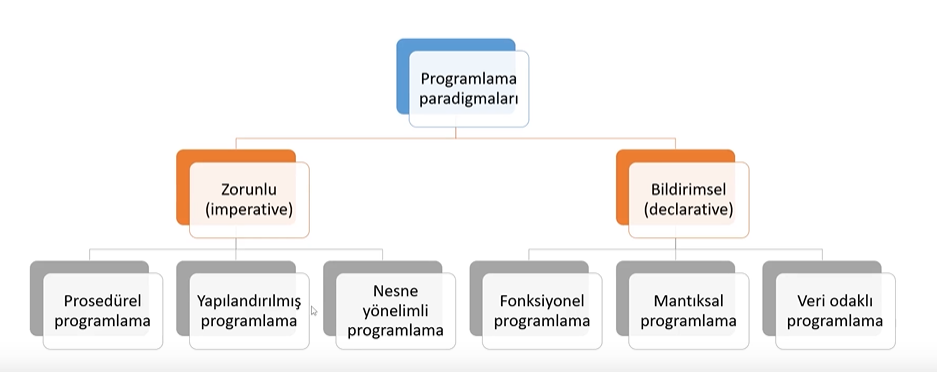
**Nesne Yönelimli Programlama (OOP):**

* **Tanım:** Bu paradigma, yazılımın nesneler üzerinden modellenmesi esasına dayanır. Nesneler, hem verileri (özellikler) hem de bu veriler üzerinde çalışan fonksiyonları (metotlar) içerir.
* **Örnek Diller:** Java, C++, Python, Kotlin .
* **Özellik:** Kapsülleme, kalıtım, çok biçimlilik gibi kavramları kullanarak karmaşık yazılımların daha yönetilebilir ve sürdürülebilir olmasını sağlar.



Nesne yönelimli paradigma; kapsülleme, kalıtım, polimorfizm gibi özellikleri kullanarak programcılara daha verimli kod yazma imkanı sağlar.

**Paralel Programlama:** Bir programın aynı anda birden fazla işlemci veya çekirdek üzerinde çalışacak şekilde tasarlanması anlamına gelir. Paralel programlama, performans artışı elde etmek amacıyla kullanılabilir ve genellikle yüksek hesaplama gücü gerektiren görevlerde tercih edilir.

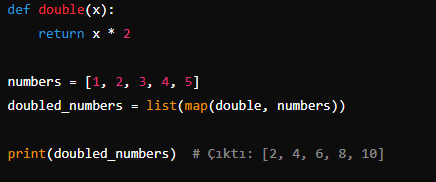


**1. Imperative (Emredici) Programlama**

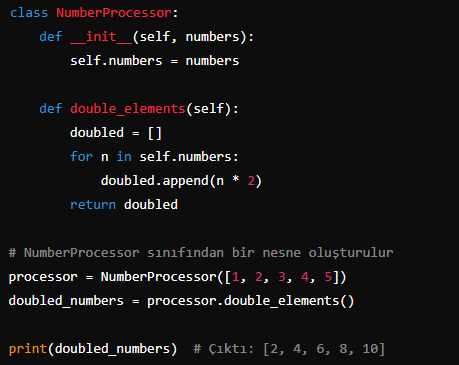
* Temel Özellik: Programcı, adım adım ne yapılması gerektiğini belirler. Kontrol akışı, döngüler, koşul ifadeleri ve değişken atamaları ile yönetilir.
* Örnek Diller: C, C++, Java, Python

**2. Declarative (Bildirimci) Programlama**

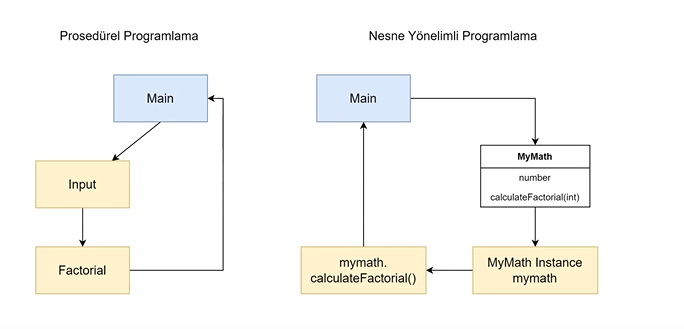
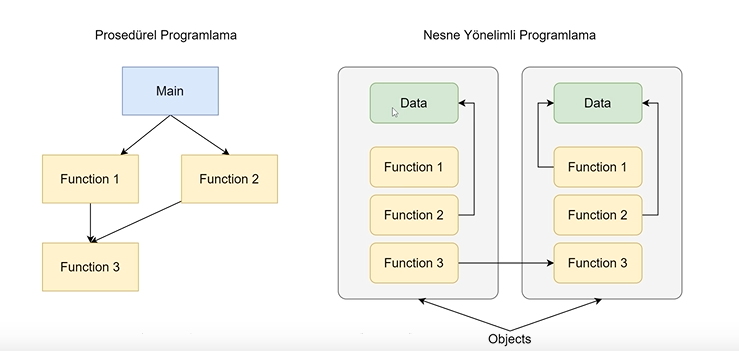
* Temel Özellik: Programcı, ne yapılması gerektiğini belirtir, nasıl yapılacağını belirtmez. Kontrol akışı dil veya çalışma ortamı tarafından yönetilir. Deklaratif yaklaşımlar genellikle bir amacın veya sonucun tanımlandığı, ancak bu sonuca nasıl ulaşılacağının detaylarının geliştirici tarafından değil, sistem tarafından yönetildiği durumlarda kullanılır
* Fonksiyonel Programlama: Yan etkisiz fonksiyonlar ve değişmez (immutable) veriler üzerine kurulu bir yaklaşım. (Örnek: Haskell, Lisp)
* Mantıksal Programlama: Mantık kuralları ve olgular (facts) üzerinden işlem yapılır. (Örnek: Prolog)
* Domain-Specific Languages (DSL): Belirli bir alan veya görev için özel olarak tasarlanmış diller. (Örnek: SQL)



Şekil 2: Fonksiyonel-Declerative



Şekil 3: OOP-Impretive



**UML DİYAGRAMLARI:**

PlantUML, UML diyagramları ve diğer çeşitli diyagram türlerini metin tabanlı bir betik dili kullanarak oluşturmayı sağlayan açık kaynaklı bir araçtır. Yazılımcıların diyagramları kolayca oluşturmasına ve güncellemesine yardımcı olmak amacıyla geliştirilmiştir. UML (Unified Modeling Language) diyagramlarını çizmek için grafiksel araçlar yerine, basit bir metin dosyasına betik yazarak diyagramları otomatik olarak oluşturur.

Sınıf diyagramları, etkinlik diyagramları, kullanım durumu diyagramları, dizin diyagramları ve daha birçok UML diyagramını destekler.

PlantUML, birçok IDE (Integrated Development Environment) ve araç ile entegre çalışabilir. Örneğin, Visual Studio Code, IntelliJ IDEA ve Eclipse gibi araçlar ile kullanılabilir.

@startuml

class User {

+String name

+String email

+login()

+logout()

}

class Admin {

+String privileges

+deleteUser()

}

User <|-- Admin

@enduml

**SOLID PRENSİPLERİ:**

SOLID prensipleri, yazılım geliştirmede daha esnek, sürdürülebilir ve bakımı kolay yazılımlar oluşturmak için kullanılan beş temel prensiptir. Bu prensipler, nesne yönelimli programlamada (OOP) iyi tasarım ilkelerini temsil eder ve yazılımcıların karmaşık sistemler geliştirmesini kolaylaştırır. SOLID, aşağıdaki beş prensibin baş harflerinden oluşan bir kısaltmadır

**1. Single Responsibility Principle (Tek Sorumluluk Prensibi)**

Bir sınıfın veya yapının **yalnızca tek bir sorumluluğa** sahip olması gerektiğini belirtir. Yani, bir sınıfın yalnızca bir amaca hizmet etmesi ve tek bir işlevi yerine getirmesi gereklidir. Bu sayede, sınıfın bir kısmı değişirse, sadece o amaca yönelik kısmı etkilenir ve diğer kısımlar bozulmaz.

* **Örnek**: Bir User sınıfı hem kullanıcı bilgilerini yönetiyor hem de veritabanı ile iletişim kuruyorsa, bu iki ayrı sorumluluk anlamına gelir. Bu durumda, veritabanı işlemlerini başka bir sınıfa taşımak gerekir.

**2. Open/Closed Principle (Açık/Kapalı Prensibi)**

Yazılım varlıklarının (sınıflar, fonksiyonlar, modüller) **genişletilmeye açık**, ancak **değiştirilmeye kapalı** olması gerektiğini savunur. Başka bir deyişle, mevcut kodu değiştirmeden yeni işlevsellik eklenebilmelidir. Bu, yazılımı daha modüler ve esnek hale getirir.

* **Örnek**: Yeni bir özellik eklemek için mevcut sınıfları değiştirmek yerine, bu sınıfların üzerine yeni sınıflar veya yöntemler ekleyebilirsiniz. Bu, özellikle miras ve arayüzler kullanılarak yapılabilir.

**3. Liskov Substitution Principle (Liskov’un Yerine Geçme Prensibi)**

Bir sınıfın türevi, temel sınıfın yerine kullanılabilir olmalıdır. Yani, bir sınıfın alt sınıfları, sınıfın işlevini bozmadan temel sınıfın yerine geçebilmelidir. Alt sınıflar, temel sınıfın işlevselliğini değiştirmemelidir.

* **Örnek**: Eğer bir Bird (Kuş) sınıfınız varsa ve bunun bir alt sınıfı olan Penguin sınıfını oluşturduysanız, Penguin de Bird olarak davranabilmelidir, fakat penguenlerin uçamaması bu durumu ihlal edebilir. Alt sınıflar temel sınıfın kontratını bozmayacak şekilde tasarlanmalıdır.

**4. Interface Segregation Principle (Arayüz Ayırma Prensibi)**

Sınıfların, **ihtiyaç duymadıkları metotlarla dolu büyük arayüzler yerine, küçük ve spesifik arayüzlerle** etkileşime girmesi gerektiğini savunur. Bu sayede, bir sınıf yalnızca ihtiyacı olan işlevleri yerine getirir ve gereksiz bağımlılıklardan kurtulur.

* **Örnek**: Bir Shape arayüzü düşünün ve bu arayüzde hem Draw() hem de CalculateArea() metodları olsun. Ancak, Line gibi bir sınıf alan hesaplaması yapmayacağı için bu arayüzden türememelidir. Bunun yerine, Shape arayüzü ikiye bölünebilir: Drawable ve Measurable.

**5. Dependency Inversion Principle (Bağımlılıkların Tersine Çevrilmesi Prensibi)**

Sınıflar, **yüksek seviyeli modüllerin düşük seviyeli modüllere doğrudan bağımlı olmasını önlemek için soyutlamalara** dayanmalıdır. Yani, sınıflar birbirine sıkı sıkıya bağlı olmamalıdır; bunun yerine aralarındaki bağımlılıklar soyutlamalar aracılığıyla yönetilmelidir.

* **Örnek**: Bir UserService sınıfı doğrudan bir MySQLDatabase sınıfına bağlı olmamalıdır. Bunun yerine, bir DatabaseInterface arayüzüne bağlı olmalıdır, böylece farklı veritabanı sistemleri kolayca entegre edilebilir.

**Arayüz (Interface)**, nesne yönelimli programlamada (OOP) sınıflar arasında bir **iletişim protokolü** işlevi gören bir yapıdır. Arayüzler, bir sınıfın hangi işlevleri (metotları) sunması gerektiğini tanımlar, ancak bu işlevlerin nasıl uygulanacağına dair detayları içermez. Arayüzler, sadece bir sınıfın hangi metodları sağlayacağını belirleyen bir şablon gibidir.

**Arayüzlerin Temel Özellikleri:**

1. **Soyut Yapılar**: Arayüzler, bir sınıfın gerçekleştirmesi gereken metotları bildirir, ancak bu metotların içini doldurmaz. Uygulamayı gerçekleştiren sınıf, arayüzde belirtilen metodların içeriğini doldurmak zorundadır.
2. **Çoklu Kalıtım**: Bir sınıf birden fazla arayüzü implement (uygulama) edebilir. Bu, Java ve C# gibi dillerde çoklu kalıtımın getirdiği zorlukları ortadan kaldırarak bir sınıfın birden fazla farklı davranış kümesini sağlayabilmesini sağlar.
3. **Bağımlılık Azaltma**: Arayüzler, sınıfların belirli bir somut sınıfa bağımlı olmasını engeller. Sınıflar, diğer sınıfların iç işleyişi hakkında bilgi sahibi olmadan sadece arayüzler üzerinden iletişim kurabilir. Bu da bağımlılıkların soyutlamalar üzerinden yönetilmesine ve bağımlılıkların azaltılmasına olanak tanır (Dependency Inversion Principle).
4. **Kodun Yeniden Kullanılabilirliği**: Arayüzler, farklı sınıfların aynı arayüzü kullanarak ortak bir davranış sunmasını sağlar. Bu sayede kod tekrarından kaçınılır ve esneklik artırılır.

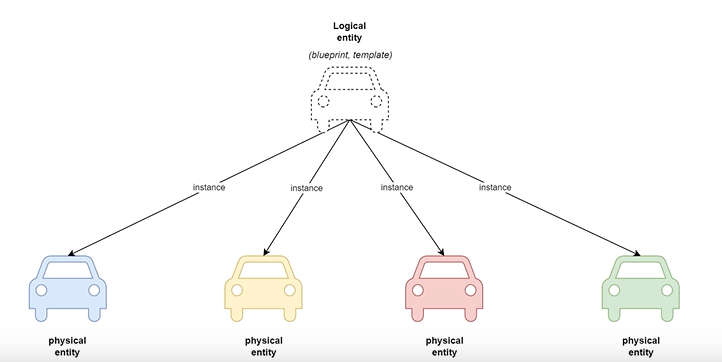
**NESNE YÖNELİMLİ PROGRAMLAMA:**

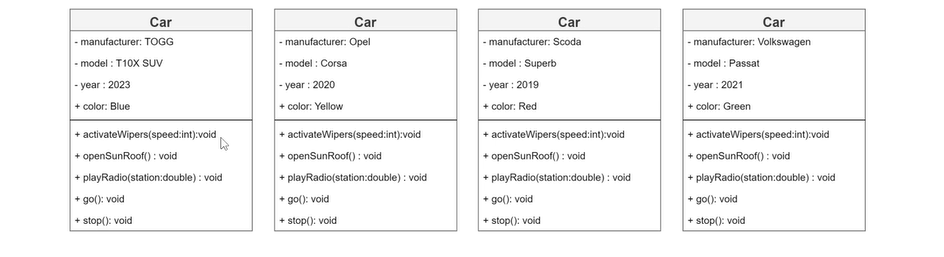
**SINIF-OBJE ve TANIMLANMASI:**

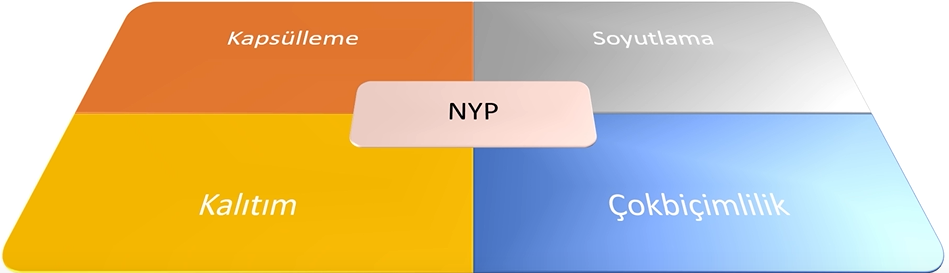
Obje (nesne), bir sınıfın somut bir örneğidir ve yazılımın temel yapı taşlarından biridir. Obje, veri (özellikler/alanlar) ve bu verilere etki eden davranışlar (metodlar/fonksiyonlar) içerir. Nesne yönelimli programlamanın (OOP) merkezinde yer alır ve sınıf tanımına dayanarak somutlaşan varlıklardır.

Sınıf, bir nesnenin (obje) şablonudur. Bir varlığın özelliklerini ve davranışlarını tanımlayan, onu modelleyen bir yapıdır. Daha basit bir ifadeyle, sınıf bir taslaktır; o sınıfa ait nesneler ise bu taslak üzerinden oluşturulan gerçek varlıklardır.

Mantıksal varlıklar(şablonlar) Sınıfın kendisi iken, fiziksel varlıklar(nesne) sınıflardan türetilen varlıklardır.







**1. Soyutlama (Abstraction):**

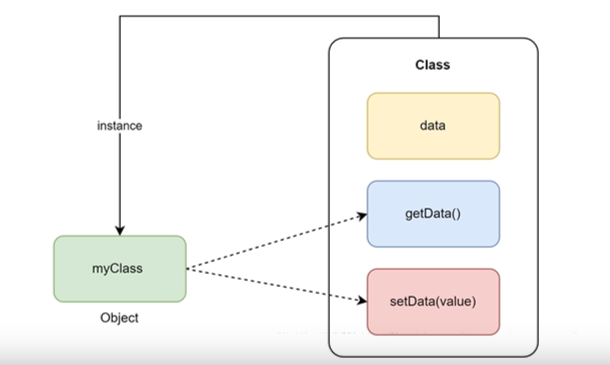
Soyutlama, karmaşık bir sistemin sadece önemli ve gerekli detaylarını kullanıcılara gösterirken, gereksiz ayrıntıları gizlemeye odaklanan bir prensiptir. Kullanıcının sistemin nasıl çalıştığıyla değil, ne yaptığıyla ilgilenmesini sağlar.

* Amaç: Karmaşıklığı azaltmak, sadece önemli işlevleri ortaya çıkarmak.
* Örnek: Bir araba düşünelim. Sürücü arabayı sürerken motorun nasıl çalıştığını, vites kutusunun iç yapısını bilmek zorunda değildir. Sadece direksiyonu çevirir, gaz pedalına basar ve arabayı kullanır. Soyutlama, bu iç mekanizmayı sürücüden gizler.
* Programlama Örneği: Bir programda sınıflar ve arayüzler, soyutlama işlemi ile gerçekleştirilir. Örneğin bir "Araba" sınıfı oluşturduğunuzda, dışarıdan bu sınıfın sadece sürüş fonksiyonlarına erişilirken, aracın iç mekanizması gizli kalır.

**2. Kapsülleme (Encapsulation):**

Kapsülleme, bir sınıfın özelliklerinin (verilerinin) ve işlevlerinin (metotlarının) bir arada tutulması ve bu özelliklere dışarıdan doğrudan erişimin engellenmesidir. Bu, verilerin güvenliğini sağlar ve sınıfın iç işleyişine dışarıdan müdahale edilmesini önler.

* **Amaç:** Veriyi korumak, sınıfın dışından direkt erişimi kısıtlamak ve sadece belirli yollarla erişim sağlamak.
* **Örnek:** Bir banka uygulamasında müşteri hesabının bakiyesi, dışarıdan erişilebilir olmamalıdır. Bunun yerine, bakiyeyi değiştirmek için belirli kurallara göre çalışan "para yatır" veya "para çek" gibi metotlar kullanılır.
* **Programlama Örneği:** Kapsülleme, genellikle değişkenlerin "private" veya "protected" olarak tanımlanması ve bu değişkenlere erişim için "getter" ve "setter" metotlarının kullanılmasıyla gerçekleştirilir. Ya da bazı dillerde ile dekoratör kullanımıyla gerçekleşir.



**3.Kalıtım(Inheritence):**

Herhangi bir kodu tekrar kullanabilmekten bahsettiğimizde genellikle kalıtımdan bahsederiz. Kalıtım, bir nesnenin başka bir nesnenin özelliklerini devralması işlemidir.

Kalıtım, bir sınıfın özelliklerinin, davranışlarının ve yöntemlerinin başka bir sınıf tarafından devralınabilmesini sağlar.

Böylece özelliklere ve davranışları miras alınan sınıf üzerinde değişiklikler yapılabilir.

**Kalıtım**, bir sınıfın özelliklerini ve metodlarını başka bir sınıfa devretmesini sağlayan bir OOP kavramıdır. Bu şekilde, yeni bir sınıf (alt sınıf veya türetilmiş sınıf) oluştururken mevcut bir sınıfın (üst sınıf veya temel sınıf) özelliklerinden ve davranışlarından yararlanabilirsiniz.

**4.Polimorfizm (Polymorphism)**

Polimorfizm, farklı sınıflardan nesnelerin aynı adı taşıyan metodları kullanabilmesi anlamına gelir. Yani, aynı metod adı farklı sınıflarda farklı işlevler gerçekleştirebilir. Polimorfizm, genellikle **method overriding** (metodun geçersiz kılınması) ve **method overloading** (metodun aşırı yüklenmesi) ile sağlanır. Python'da method overloading (metodun aşırı yüklenmesi) doğal olarak desteklenmez, ancak method overriding ve dinamik türler sayesinde polimorfizm elde edilebilir.

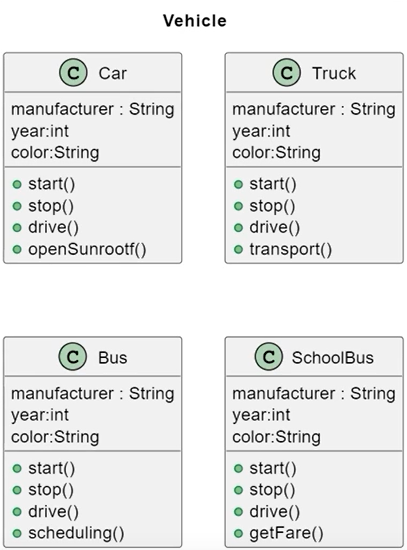
Polimorfizm (çok biçimlilik), nesne yönelimli programlama (OOP) konseptlerinden biridir ve farklı sınıfların ortak bir arayüzü paylaşarak, aynı fonksiyonları farklı şekillerde uygulamasını sağlar. Polimorfizm, bir nesnenin veya fonksiyonun, bağlı olduğu türe göre farklı davranışlar sergilemesi anlamına gelir. OOP'de çok biçimlilik (polimorfizm), bir temel sınıfın metotlarının alt sınıflar tarafından geçersiz kılınarak (method overriding) farklı şekillerde davranabilmesini sağlayan bir özelliktir. Bu sayede aynı isimdeki metotlar, alt sınıflarda farklı işlevsellik gösterebilir. Çok biçimlilik, nesne yönelimli programlamanın temel kavramlarından biridir ve esneklik ile yeniden kullanılabilirlik sağlar.

C++'da polimorfizmin kullanımına bir örnek, sanal (virtual) fonksiyonlarla gösterilebilir. Polimorfizm, özellikle kalıtım (inheritance) yoluyla türetilmiş sınıflarda, temel sınıfın işlevlerini yeniden tanımlayarak uygulanır.

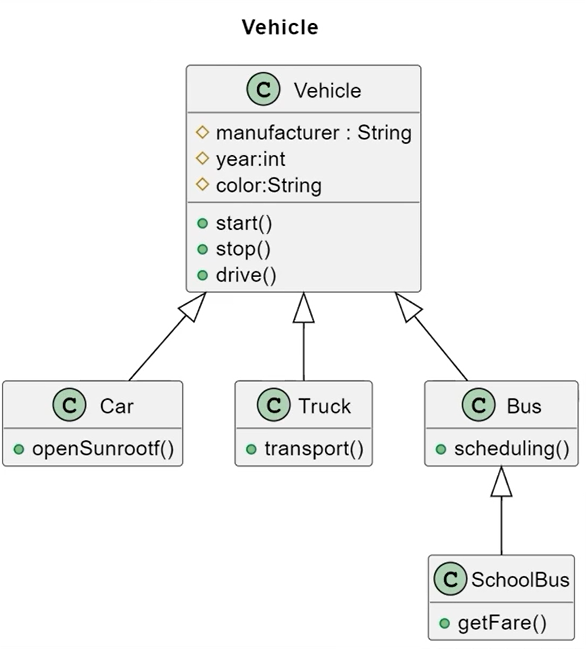
Not:

Bir türetilmiş sınıf, kendisinden türetilen başka bir sınıfın base (temel) sınıfı olabilir. Bu, **çok katmanlı kalıtım (multi-level inheritance)** olarak adlandırılır. C++ gibi nesne yönelimli programlama dillerinde, bir sınıfın başka bir sınıftan türetildiği ve ardından bu sınıftan da yeni bir sınıfın türetildiği bir yapı oluşturulabilir.

Bu aşamadan itibaren kalıtımı daha detaylı inceleyeceğiz.



Kalıtım ile;



**ÖNEMLİ!!:** Bir sınıfın yapıcısı çağrıldığında, öncelikle temel (base) sınıfın yapıcıları çalıştırılır, ardından alt (derived) sınıfın yapıcıları çalıştırılır. Bu işlem, sınıf hiyerarşisinin doğru bir şekilde başlatılmasını sağlamak için önemlidir.

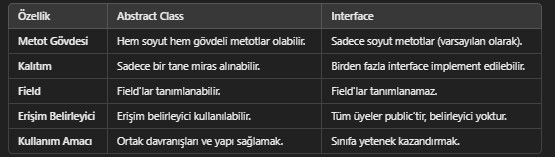
* **Varsayılan Yapıcı (Default Constructor)**: Eğer temel sınıfın varsayılan yapıcısı varsa, bu yapıcı, alt sınıfın yapıcısında : Base() ifadesi yazılmasa bile otomatik olarak çağrılır.
* **Parametreli Yapıcı**: Eğer temel sınıfın parametreli bir yapıcısı varsa, bu durumda alt sınıfın yapıcı metodunda temel sınıfın yapıcısını açıkça çağırmak gerekir (başlangıç listesi ile).
* Temel sınıfın parametresiz bir yapıcı varsa, : base() kullanmasan bile çalışır.
* Temel sınıfın yalnızca parametreli yapıcıları varsa, : base(parametreler) kullanmak zorundasın.

**ABSTRACT CLASS:**

Abstract class, nesne yönelimli programlamada (OOP) bir sınıf türüdür ve kendisinden doğrudan nesne oluşturulamayan, yalnızca türetilmiş sınıflar tarafından kalıtılabilen bir sınıftır. Abstract class'lar genellikle bir şablon görevi görür ve temel işlevleri, ortak özellikleri ve davranışları tanımlarken, alt sınıfların bu işlevleri özelleştirmesi veya uygulaması gerektiğini belirtir.

* **Abstract Class**: Hem soyut (gövdesiz) hem de gövdeli metotlar içerebilir. Yani bazı işlevleri tanımlayabilir, bazılarını alt sınıflara bırakabilir.
* **Interface**: Yalnızca soyut metotlar içerir (gövdesiz). C# 8.0 ve üstü versiyonlarda ise varsayılan olarak metot gövdesi olan "default implementation" eklenmiştir, ancak bu yine de bir istisna niteliğindedir.
* **Abstract Class**: Bir sınıf yalnızca bir abstract class'ı miras alabilir (çoklu kalıtıma izin verilmez).
* **Interface**: Bir sınıf birden fazla interface'i implement edebilir. Bu, çoklu kalıtım gerektiren durumlarda büyük avantaj sağlar.

Abstract Classlar genellikle interface'lere göre biraz daha hızlıdır çünkü abstract class'lar daha az esneklik sunar ve daha belirgin bir yapı sağlar.

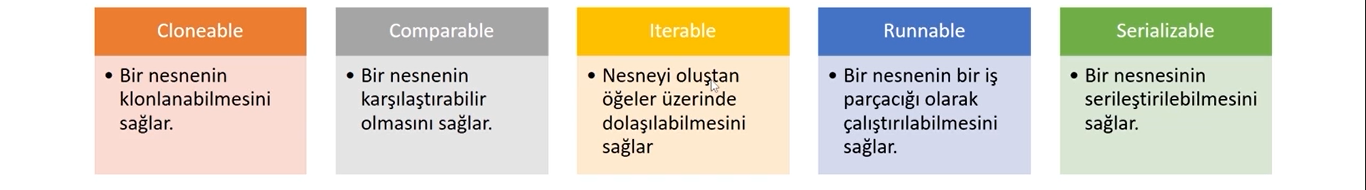


**INTERFACES:**

Arayüzler (interfaces), nesne yönelimli programlama (OOP) dillerinde, belirli bir sınıfın hangi işlevleri (metotları) içereceğini tanımlayan bir yapıdır. Ancak, arayüzler sadece ne yapılacağını belirtir, nasıl yapılacağını tanımlamazlar. Yani arayüzler, metotların gövdesini içermezler, sadece bu metotların başlıklarını (imzalarını) belirlerler

Java ve C# gibi dillerde: Comparable, Iterable, Serializable gibi varsayılan arayüzler(interfaces), belirli fonksiyonların uygulanmasını zorunlu kılar. Örneğin, Comparable arayüzünü implement eden bir sınıf, doğal bir sıralama mantığı sunmalıdır.

NOT: Bir Java dosyasında en fazla **bir adet** public sınıf bulunabilir. Eğer bir dosyada bir public sınıf varsa, dosya adı bu sınıfın adıyla aynı olmalıdır.



Comparable arayüzü, Java’da nesnelerin doğal sıralama (natural ordering) yeteneğini tanımlamak için kullanılan bir yerleşik arayüzdür. Bir sınıfın Comparable arayüzünü uygulaması, o sınıfın nesneleri arasında nasıl karşılaştırma yapılacağını belirlemesine olanak tanır. Arayüzlerin gövdesi olmadığı için implement edildikten sonra gövde implement edilen sınıfta tanımlanır.

**STATIC KULLANIMI:**

Java’da JVM ile birlikte program çalıştırıldığında sanal makine aşağıdaki sırayla birlikte yükleme yapar:



static anahtar kelimesi, bir sınıfın üyesini (değişken veya metot) sınıf düzeyinde tanımlamak için kullanılır. Bu, o üyeye ait verinin ya da işlemin nesneye değil, doğrudan sınıfa bağlı olduğu anlamına gelir. static üye, sınıfın herhangi bir örneği oluşturulmadan kullanılabilir ve sınıfın tüm örnekleri arasında ortak olarak paylaşılır. Yani statik sınıflar herhangi bir nesneye özgü olmayan kodları depolamak için de kullanılabilir.

Static metotlar, bir sınıfın bir nesnesine ihtiyaç duymadan çağrılabilen metotlardır.

Statik sınıflar, yalnızca statik değişkenler,statik yöntemler ve statik yapılar içerebilir.Yani sınıf statikse sınıf içi her şey statik olmalıdır.

public class MathUtils {

public static int square(int number) {

return number \* number;

}

}

int result = MathUtils.square(5); // Nesne oluşturmadan doğrudan sınıf ismi ile çağrılır

System.out.println(result); // Çıktı: 25

**1. Primitive Tür (İlkel Tür)**

Primitive türler, veriyi doğrudan bellekte saklarlar. Yani, bir değişkenin kendisi o veriyi tutar. Bellekte bu türler sabit bir yer kaplar ve her zaman bir değer içerirler. Örneğin, Java'da int, boolean, char, float gibi türler **primitive türlerdir**.

* **int**: Bir int değişkeni, bellekte doğrudan bir tamsayı değerini saklar. Örneğin:

int x = 5;

Bu durumda, x değişkeni bellekte doğrudan 5 değerini tutar. Eğer x değişkeninin değerini başka bir değişkene atarsanız, bu atama doğrudan bir kopya olur.

int y = x; // y de artık 5 değerini tutar, iki bağımsız değişken.

* **Bellek kullanımı**: Bellekte sadece veri saklanır.
* **Hafif ve hızlıdır**: Primitive türler doğrudan veri sakladıkları için daha az bellek kullanırlar ve işlemleri daha hızlıdır.

**2. Referans Tür (Nesne)**

Referans türler, nesneleri temsil eden ve bellekte bir referans (adres) ile tutulan veri türleridir. Bu türler, NYP'nin kalbini oluşturur çünkü nesnelerle çalışmamızı sağlarlar.

Referans türler ise, veriyi **dolaylı olarak** saklarlar. Yani, bir değişkenin kendisi veriyi doğrudan tutmaz, onun **bellekteki adresini** (referansını) tutar. Java'da her sınıf bir referans türüdür ve nesneler bu şekilde yönetilir. Örneğin, Integer, String, ArrayList gibi sınıflar referans türlerdir.

* **Integer**: Bir Integer değişkeni, bellekteki bir Integer nesnesine referans eder. Örneğin:

Integer a = 5;

Bu durumda, a değişkeni bellekte doğrudan 5 değerini tutmaz, onun yerine Integer nesnesinin bulunduğu bellekteki bir adrese referans eder. Bu adreste o nesne tutulur. Eğer a'yı başka bir değişkene atarsanız, sadece o bellekteki adrese işaret eden referans kopyalanır, nesnenin kendisi kopyalanmaz.

Integer b = a; // b ve a aynı nesneyi gösterir

Başka bir ifadeyle,

**Integer**: Bu, bir **referans tür** (nesne türü) olup, Java'da int primitive türünü sarmalayan (wrapper) bir sınıftır. Integer nesneleri, bir int değerini nesne olarak tutar ve ek işlevsellik sağlar.

* **Bellek kullanımı**: Bellekte nesnenin kendisi ve ona işaret eden referans saklanır.
* **Daha fazla bellek harcar ve yavaştır**: Nesne yönetimi (nesnenin oluşturulması, yıkılması) daha fazla işlem gerektirir. Ama bu küçük dezavantajı aslında birçok avantajın da oluşmasını sağlar nesne yönelimli paradigm sayesinde

Referas tür(nesne), OOP’nin bir gerkliliğidir. Faydalarından biri veri paylaşımı referans tür sayesinde kolaylaşır.

**Veri Paylaşımı**: Eğer aynı arabayı birden fazla yerde kullanmak istiyorsan, her yere yeni bir araba koymak yerine, hepsinin aynı adrese işaret etmesini sağlarsın. Yani, herkes aynı arabayı paylaşır.

Nesne yönelimli programlamada (OOP) primitive türler ile referans türler arasında dönüşümler (casting) vardır. Bu dönüşümün nedenleri ve nasıl çalıştığını açıklayalım:

1. Dönüşüm İhtiyacı

Primitive türler ve referans türler arasında dönüşüm gerekebilir çünkü her iki tür de farklı veri saklama yöntemlerine sahiptir. Örneğin:

* Primitive Türler: Doğrudan veri (örneğin, int, float, boolean) tutar.
* Referans Türler: Bellekte bir nesneye işaret eden adresleri tutar.

Bu nedenle, bir programda hem primitive hem de referans türler kullanıldığında, bu iki tür arasında veri aktarımı ve dönüşüm gerekebilir.

Bu dönüşüm, genellikle **boxing** ve **unboxing** işlemleri ile gerçekleştirilir:

**Boxing ve Unboxing Nedir?**

Boxing ve unboxing, **primitive (ilkel) veri türleri** ile **referans türleri** arasındaki otomatik dönüşüm işlemidir. Örneğin, bir int değeri gibi ilkel bir tür, Integer gibi bir referans türüne dönüştürülebilir. Bu dönüşüm otomatik olarak yapılır ve buna **boxing** denir. Tersi işlem de **unboxing** olarak adlandır

int primitive = 5;

Integer boxed = primitive; // Boxing işlemi

Integer boxed = 5;

int primitive = boxed; // Unboxing işlemi

Referans türler, nesnelerin bellekte nasıl yönetileceğini sağlar. OOP dillerinde, nesneler genellikle dinamik olarak oluşturulur ve bellekte belirli bir konumda saklanır. Referans türler, bu nesnelere işaret eden adresleri tutarak bellek yönetimini kolaylaştırır.

* **Nesne Oluşturma**: Nesneler dinamik olarak oluşturulduğunda, bu nesnelerin nerede bulunduğu referans türleri ile takip edilir. Örneğin:

Person person = new Person("Alice", 30);

Burada person, Person nesnesinin adresini tutan bir referans türdür.

**Gerçek Değer (Nesne):** Nesne yönelimli programlamada, gerçek veriler (özellikler, metotlar) genellikle **heap** adı verilen bir bellek alanında tutulur. Bu, program çalışırken dinamik olarak ayrılan bir alandır. new anahtar kelimesi ile bir nesne oluşturduğumuzda, heap'te bu nesne için yer ayrılır ve bize bu nesnenin adresi verilir. new Person("Alice", 30); ile birlikte oluşturulan nesne burada tutulur.

**Referans (Adres):** Nesnenin bu heap üzerindeki konumunu gösteren adres ise genellikle **stack** adı verilen bir bellek alanında tutulur. Stack, programın yerel değişkenlerini ve fonksiyon çağrılarını yönetmek için kullanılan daha düzenli bir bellek bölgesidir. Person’a referans veren person bir referans türü olup stack adı verilen yerde saklanır.

**Neden Bu Ayrım Önemli?**

* **Bellek Yönetimi:** Heap ve stack'in farklı özellikleri, bellek yönetimini daha etkin hale getirir.
* **Performans:** Stack'e erişim genellikle heap'e erişimden daha hızlıdır.
* **Yaşam Süresi:** Stack'teki değişkenler, fonksiyon çağrısı bittiğinde otomatik olarak temizlenirken, heap'teki nesneler garbage collector tarafından yönetilir.

**Garbage Collection**

* Java'da, kullanılmayan nesnelerin bellekten otomatik olarak temizlenmesini sağlayan bir mekanizma olan **garbage collection** (çöp toplama) bulunur. Heap belleğinde oluşturulan nesneler, referansları kalmadığında otomatik olarak temizlenir.

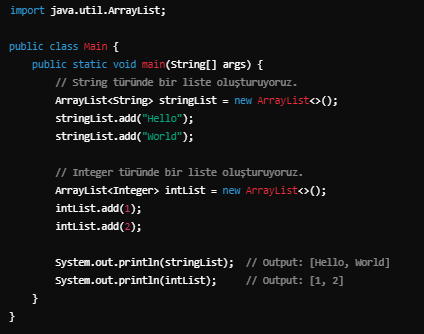
**Referans Sayma**: Bir nesneye olan referans sayısı sıfıra düştüğünde, nesne bellekten temizlenir.

**Özet**

* **Sınıf Tanımı**: Sınıfın kendisi, bellek üzerinde bir yapı olarak yer alır (sınıf meta verisi olarak).
* **Nesne Referansı**: Nesne referansları stack belleğinde saklanır.
* **Nesne Değerleri**: Gerçek nesne değerleri ise heap belleğinde tutulur.
* **Garbage Collection**: Kullanılmayan nesneler, garbage collection ile bellekten temizlenir.

**GENERIC VE DUCK TYPING:**

Generics, Java, C#, C++ gibi statik tipli programlama dillerinde kullanılır ve tür güvenliği sağlarken, kodu daha esnek ve yeniden kullanılabilir hale getiren bir tekniktir. Generic yapılar, hangi türlerle çalışacağını derleme zamanı (compile-time) sırasında belirler.



Burada ArrayList<T> generic bir sınıftır. T yerine herhangi bir tür kullanılabilir. Bu sayede, farklı veri türleriyle çalışırken kodu yeniden yazmanıza gerek kalmaz ve tip güvenliği sağlanır.

Generics, tür dönüşümü (casting) ihtiyacını ortadan kaldırır. Normalde, bir liste gibi veri yapıları **Object** türünde elemanları saklar, bu da kodu yazarken doğru türde elemanları çekebilmek için **cast** işlemi yapmayı gerektirir. Ancak cast işlemleri çalışma zamanında hataya yol açabilir. Generics, bunu ortadan kaldırarak hem **daha güvenli** hem de **daha okunabilir** bir kod yazmanızı sağlar.

( Object, Java'nın en üst düzey sınıfıdır ve tüm sınıfların (primitive türler hariç) ortak bir süper sınıfıdır. **Örnek**: Eğer kendi sınıfınızı oluşturuyorsanız, bu sınıf otomatik olarak Object sınıfından miras alır.)

* **Object Sınıfı**: Java'daki tüm nesne türlerinin temelini temsil eder ve temel metodlar içerir.
* **Kullanım**: Farklı türdeki nesneleri tek bir referansla saklamak için kullanılır.
* **Casting İhtiyacı**: Object türünde nesneleri kullanırken tür dönüşümü (casting) gereklidir ve bu, potansiyel hatalara yol açabilir.

Bu nedenle, **Generics**, Object türünü kullanmak yerine belirli bir türü belirterek, tür güvenliğini artırır ve casting ihtiyacını ortadan kaldırır.

**WRAPPER NESNELER:**

**Wrapper nesneleri**, Java'da temel veri türlerinin (primitive types) nesne tabanlı temsilidir. Java, temel veri türlerini nesne olarak kullanmanıza olanak tanıyan Wrapper sınıflarını sağlar. Bu sınıflar, temel veri türleriyle nesne tabanlı programlamada çalışmanıza yardımcı olur ve özellikle koleksiyonlar gibi nesne tabanlı yapılarla etkileşimde bulunurken önemlidir.

**Temel Veri Türleri ve Karşılık Gelen Wrapper Sınıfları**

Java'da bulunan temel veri türları ve bunların karşılık geldiği Wrapper sınıfları aşağıda listelenmiştir:

| **Temel Veri Türü** | **Wrapper Sınıf** |
| --- | --- |
| int | Integer |
| double | Double |
| char | Character |
| boolean | Boolean |
| byte | Byte |
| short | Short |
| long | Long |
| float | Float |

**Wrapper Sınıflarının Özellikleri**

1. **Nesne Temelli**: Wrapper sınıfları, temel veri türlerini nesne olarak kullanmanıza olanak tanır. Bu, nesne tabanlı koleksiyonlar (örneğin, List, Set, Map) kullanırken önemlidir.
2. **Otomatik Boxing ve Unboxing**: Java, temel veri türlerini Wrapper nesnelerine otomatik olarak dönüştürme (boxing) ve Wrapper nesnelerini temel veri türlerine otomatik olarak dönüştürme (unboxing) yeteneğine sahiptir. Bu, kod yazımını daha basit hale getirir.

**EK:**

Java'da her sınıf (eğer başka bir sınıftan miras almıyorsa), varsayılan olarak Java'nın temel sınıfı olan java.lang.Object sınıfından türetilir. Yani, senin Employee sınıfın aslında şununla eşdeğerdir:



toString() metodu, Java'daki tüm sınıfların doğrudan veya dolaylı olarak genişlettiği Object sınıfında tanımlıdır. Object sınıfındaki toString() metodu, nesne hakkında bazı temel bilgileri döner. Ancak, çoğu zaman bu metot override edilerek (ezilerek) daha anlamlı bir çıktı vermesi sağlanır.