# UML

“+” değişken tür = public (Attributes)

“#” değişken tür = protected (Attributes)

“~” değişken tür = private (Attributes)

—----------------------------------------------

+yaz(): void

# Temel OOP

## Inheritance:

Kalıtım(Inheritance), bir sınıfın kendisine ait özellikleri ve işlevleri bir başka sınıfa aynen aktarması ya da bazı özellik ve işlevlerini diğer sınıfların kullanmasına izin vermesi şeklinde oluşmaktadır.

Üst sınıf (superclass) özelliklerini alt sınıflarda (subclass) kullanmak için extends deyimi ile kullanırız.

## Polymorphism:

Bir tek metot adıyla, birden çok farklı metotları koşturma özeliğine, java'da polymorphism denir. Şu önemli özeliği biliyoruz. Bir üst sınıfın referansı (işaretçi, pointer) bir alt sınıfa ait nesneyi işaret edebilir. Java, bu özelliği kullanarak, run-time aşamasında hangi overridden metodu seçmesi gerektiğini bulur.

-Birden çok veri türünü depolamak için tek değişken kullanılabilir.

-Farklı işlemler arasındaki bağlantıyı azaltır.

-Kodun tekrar kullanılması sağlanır.

-Yapılan programda hataları ayıklamak daha kolay olur.

## Interface:

Arayüz. Java'da arayüz soyut sınıf yerine kullanılır, ama soyut sınıftan farklı ve daha kullanışlıdır. Arayüz kullanarak, bir sınıfın neler yapacağını belirlerken, onları nasıl yapacağını gizleyebiliriz. Arayüzün yapısı sınıfın yapısına benzerse de aralarında önemli farklar vardır.

Bir sınıf birden fazla interface ile implement (çağırılabilir) edilebilir.

Interface, anlık değişken içermezler. Bu yüzden interface üzerinde tanımlanan değişkenler, interface ile türetilen sınıflar tarafından değiştirilemezler.

Tanımlamalarını gerçekleştirirken sadece public veya default erişim belirleyici kullanabiliriz.

Tanımladığımız Interface’i, birden fazla sınıf çağırarak kullanabilir.

Interface üzerinde tanımlanan metotlar gövdesizdir. Bu yapısı ile abstract metotlara benzerlik gösterir.

Interface sınıfını kullanarak nesne üretemeyiz.

## Abstract:

Java'da soyutlama (Abstraction), nesne yönelimli programlamanın yapı taşlarından biridir. Java'da soyutlama, gereksiz ayrıntıları göz ardı ederek bir nesnenin yalnızca gerekli özelliklerini belirleme işlevi olarak da nitelendirilebilir.

Nesne tabanlı programlamada sınıf hiyerarşisi oluşturulurken, bazen en tepede bulunan sınıf türünden nesneler programcılar için anlamlı olmayabilir. Hiyerarşinin en tepesinde bulunan sınıfın kendisinden türetilecek olan alt sınıflar için ortak bir arayüz (interface) görevi yapması istenebilir. Bunun için çözüm olarak oluşturulan metotlara ve sınıflara soyut metot (abstract method) ya da soyut sınıf (abstract class) denir. Soyut sınıflar büyük projelerde kullanılırlar ve kalıtım özelliğini kullanarak kod tekrarını azaltırlar. Soyut sınıflar diğer sınıflara taban olmak için kullanılırlar. Nesne türetemezler. Önlerine “abstract” sözcüğü yazılarak soyutlaştırılırlar. Önlerine "virtual" yazılmaz çünkü “abstract” sözcüğü uygulanan tüm sınıf ve metotlar zaten sanaldır.

Kullanırken dikkat edilmesi gerekenler:

• Soyut sınıflar “abstract” türünden nesneler tanımlamazlar.

• Soyut sınıflar içerilerinde soyut olmayan metotlar da barındırabilir ancak soyut metotlar sadece soyut sınıflar içerisinde bildirilebilir.

• Soyut metotlar türeyen sınıfta mutlaka bildirilmelidir.

• Soyut metotlar override edilmek zorundadırlar, aksi takdirde derleyici hatası alınır.

• “Static” metotlar soyut olarak tanımlanamazlar.

• Soyut sınıflar “private” olarak tanımlanamazlar ama “public” ve “protected” olarak tanımlanabilirler.

• Soyut sınıflar “sealed” anahtar sözcüğü ile ifade edilemezler.

## Override:

Overriding bir sınıfa ait bir metodun, o sınıftan türetilmiş bir sınıf içerisinde aynı isimli bir metod tanımlanarak, bu metodun temel sınıftaki metodun yerine geçirmeye denir. Bu işlem,bir metodun aynı sınıftan türetilmiş farklı sınıflarda farklı işlere yaramasını sağlar.

Java Overriding, programlarımızda kullandığımız kalıtım sayesinde üst sınıftan, alt sınıfa geçecek metotların o sınıf yapısına uygun bir şekilde tekrardan gövdesinin değiştirilmesine ve kod satırlarının yeniden şekil almasına olanak sağlayacaktır. Tabii bu imkanları sağlarken bazı uymamız ve dikkat etmemiz kuralların olduğunu unutmamalıyız.

### Java Overriding Kullanım Kuralları:

Üst sınıf içerisinde yer alan private metotlar override işlemine tabi tutulmaz. Aksi takdirde hata alınır.

Override ettiğimiz bir metodun erişim belirleyicisinin seviyesini düşüremeyiz fakat yükseltebiliriz.

Üst sınıftan miras aldığımız bir metodun dönüş tipini veya parametrelerini değiştiremeyiz.

Static ve Final olarak tanımlı metotları override edemeyiz.

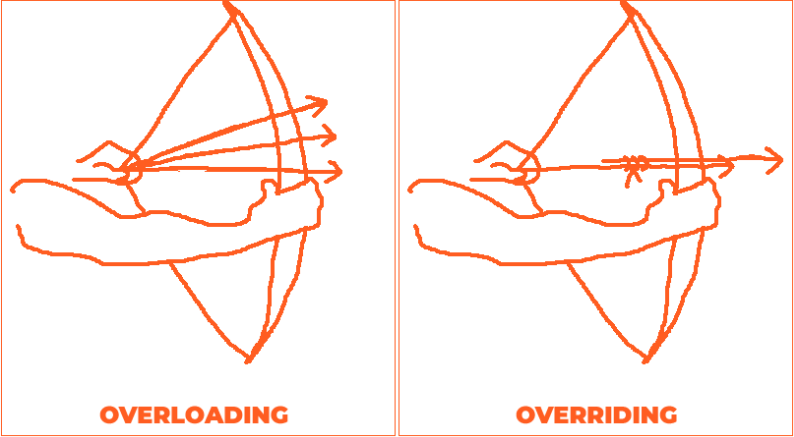
Üst sınıfımızdaki ve alt sınıfımızda bulunan override edeceğimiz metot isimleri aynı olmalıdır.

Constructor (Yapıcı / Kurucu) metotlar override edilemez.

## Overload:

Java, aynı isme sahip birden fazla metodlar tanımlamamıza izin vermektedir. Bu şekilde birden fazla aynı metodun yazılmasına overloading, aşırı yükleme adı verilir.

Java Overloading, projelerimizde oluşturmuş olduğumuz metotların yapısal olarak yeniden yazılmasına, esneklik sahibi olmasına ve daha fazla işlevsel bir hal almasına olanak sağlayarak aynı isimde birden fazla metodumuzun oluşmasına imkan verir.



### Overloading Yaparken Uymamız Gereken Kurallar:

Projelerimizde metot yapılarımızı aşırı yüklemek (overload) yapmak istediğimiz zaman dikkat edeceğimiz bazı hususlar vardır. Bu hususlar şu şekildedir:

-Overload yapacağımız metotların isimlerinin aynı olması gerekmektedir.

-Metotlarımızın sahip oldukları parametreler birbirinden farklı olmalıdır. Örneğin: metot(parametre 1) ve metot (parametre 1, parametre 2) şeklinde olmalıdır.

-Overload işlemine tabi tuttuğumuz metot yapılarının dönüş tipleri birbirinden farklı olabilir. Örneğin: void metot(prm1) ve int metot(prm1,prm2) şeklinde dönüş tiplerini değiştirebiliriz.

-Yukarıdaki madde üzerinde görüldüğü gibi devamı olarak ekstra bir tanımlama yapmak istiyorum, oluşturmuş olduğumuz metot yapılarının dönüş tiplerini sadece değiştirmemiz yetmez, ekstra olarak bir parametre tanımlaması veya çıkartması yapmamız gerekmektedir.

## Encapsulation:

Java Encapsulation ise, java sınıfı içinde yer alan değişkenlerimizi korumak ya da saklamak anlamına da gelir. Tanımladığımız bir sınıf içinde bulunan değişkenin direkt olarak değiştirilememesi, bunun yerine de bizim izin verdiğimiz kadar, metotlar aracılığı ile değiştirilmesi de gerekir.

Kapsülleme kavramını anlamlandırabilmek için önce **erişim belirleyicilerin** çalışma prensiplerini iyi biliyor olmamız gerekmekte.



### Java’da Kapsülleme Avantajları:

-Kapsülleme ile diğer sınıflardan erişilmesini engelleyebiliriz ve bu sayede verileri gizleyebiliriz.

-Kapsülleme, programın sürdürülebilirliğini arttırır.

-Kapsülleme ile kodları daha esnek, düzenli ve daha okunaklı bir duruma getirebiliriz.

-Kapsülleme, yeniden kullanılabilirliği arttırır ve yeni gereksinimlerle göre kolayca değiştirilebilir.

### Getter ve Setter Metotlar:

Java Encapsulation kavramı ile ayrılmaz bir ikili konumunda olan getter ve setter metotlar oldukça önemli bir konudur. Erişim belirleyiciler başlığı altında oluşturmuş olduğum tabloda ve örnek üzerinde private olarak tanımladığımız elemanlara, başka bir sınıf üzerinde erişim sağlayamayacağımızı anlatmıştım. Java’da bu gizli olarak tuttuğumuz bu bilgilere ulaşabilmemiz için C# dilinde get-set olarak bilinen metotları kullanacağız.

Bir nevi anahtar görevi gören bu metotlar sayesinde private olarak tanımlı elemanlar okunabilir veya içerisindeki değerleri değiştirebiliriz. Bu verileri okuyabilmek için getter, içerisindeki verileri değiştirmek için setter metotlarını kullanırız.

# Try/Catch

Bu yapıda, try gövdesinin içerisine istisna, hata oluşturabilecek kod bloğu yerleştirilir. Her try kavramından sonra catch komutu gelmektedir. Bir istisna veya hata meydana geldiğinde catch kavramı tarafında yakalanır.

try{

//Hata oluşması beklenen kod bloğu

}

catch(hata tipi){

// Bu hatanın yönetilmesi için gerekli kod bloğu

}

finally{

//Hata oluşsada oluşmasada son olarak çalıştırılacak kod bloğu

}

public class MatematikselIslemler {

publicstaticvoid main(String[] args) {

try {

Scanner sc=new Scanner(System.in);

System.out.println("1.sayıyı girin:");

doublebolunen=sc.nextDouble();

System.out.println("2.sayıyı girin:");

doublebolen=sc.nextDouble();

doublesonuc=bolunen/bolen;

System.out.println("Sonuç="+sonuc);

} catch (InputMismatchException e) {

System.out.println("Lütfen sadece sayı giriniz.");

}

finally{

System.out.println("İşlem sonlandı.");

}

}

}

# Throw/Exception

Java Exception Handling konusunun son kullanıcıdan çok yazılımcıları etkileyen bir yönü de Throw Exception (Hata Fırlatma) konusudur. Hata fırlatma özelliği yazılımcının işini kolaylaştıran aynı zamanda hataları kendi koduna göre özelleştirmesini sağlayan bir özelliktir.

# Timer

Timer sınıfı; bir kod bloğunun ne zaman veya ne sıklıkta çalışacağını belirlemek için kullanılır. Timer sınıfını kullanabilmek için aşağıdaki paketi import etmemiz gerekir.

java.util.Timer

Timer sınıfından bir nesneyi aşağıdaki yapılandırıcılar ile yaratabiliriz.

Timer()

Timer nesnesi yaratır.

Timer(String name)

Timer nesnesini bizim belirleyeceğimiz özel bir adıyla yaratır.

Timer sınıfından yaratılmış bir nesne aşağıdaki metodlar ile kullanılabilir.

void cancel()

Timer’ı sona erdirir.

void schedule(TimerTask task, Date time)

Belirtilen görevin hangi zamanda çalışacağını belirler.

void schedule(TimerTask task, Date firstTime, long period)

Belirtilen görevi hangi zamanda başlatacağını ve ne sıklıkla çalışacağını belirler.

void schedule(TimerTask task, long delay)

Belirtilen görevi ne kadarlık bir gecikmeyle başlayacağını zamanlar.

void schedule(TimerTask task, long delay, long period)

Belirtilen görevi ne kadarlık bir gecikmeyle başlayacağını ve ne sıklıkla çalışacağını belirler.

void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)

Belirtilen görevi hangi zamanda başlatacağını ve hangi sabit oranlı periyodlar ile çalışacağını belirler.

void scheduleAtFixedRate(TimerTask task, long delay, long period)

Belirtilen görevi ne kadarlık bir gecikme ile başlayacağını ve hangi sabit oranlı periyodlar ile çalışacağını zamanlar.

# Thread

# Enum

# Lambda