**Javada erişim belirleyiciler:**



**Private:**

Nesne yönelimli programlamanın temel ilkelerinden olan sarmalama ilkesi gereği, sınıf içindeki değişkenler “private” olarak tanımlanır.

**Public:**

Public ögeler sadece ait olduğu sınıfta değil diğer sınıflardan da erişilebilir.

**Protected:**

Javada “protected” deyimi, public ve private arasında kalan erişim düzeyidir, Protected tanımlanan ögeler, kendisi ile aynı package(paket) içerisinde bulunan sınıflar tarafından doğrudan erişilebilir. Varsayılan erişim belirleyici protected’dır.

**STATIC:**

Java’da static deyimi sınıf değişkenleri ve tanımlanırken kullanılır. Eğer bir sınıfa ait değişkenlerin başına “static” yazılırsa, o değişkenler artık sınıf değişkeni olurlar. Sınıf değişkeni olarak tanımlanan değişkenler, her nesne oluşturduğumuzda ayrı ayrı oluşmazlar. Sınıfa it ne kadar nesne olursa olsun, sınıf değikenleri 1 tanedir. Sınıfa ait herhangi bir nesne üzerinden bu değişkene ulaşılabilir. Sınıf değişkenlerinin bir özelliği ise, ilgili sınıfa ait hiç nesne oluşturulmasa bile bellekte yer kaplar.

Static kod blokları:

Herhangi bir nesne üretilmeden hafızaya alınmasını istediğimiz kodları bu bloklar içerisine yazarız. Böylece bu kodlar bir çalıştırılır.

Final:

Javada sabit değerler tanımlamak için kullanılan deyimdir. Final değişkenler public olabilir değerleri değiştirilemeyeceği için public olması önemli değildir.

Encapsulation:

Nesne yönelimli programlamada sınıfa ait değişkenlere doğrudan erişimi engelleyip belirsiz durumların önüne geçilir.

Getter/Setter:

Getter metodlar class içerisinde private tanımlı değişkenlerin değerlerini elde edebilmek için kullanılan metodlardır.

Setter metodlar ise private değişkenlerin değerlerini değiştirebilmeyi sağlarlar.

SINIFLAR ARASI İLİŞKİLER:

Bağımlılık (Dependency):

Javada bir sınıfın başka bir sınıfı metod parametresi olarak kullanması “uses a”, kullanır ilişkisi vardır ve bağımlılık olarak tanımlanır. A sınıfı B sınıfını metod parametresi olarak kullanıyorsa, A sınıfı B sınıfına bağımlıdır. Bağımlılığın fazla olması programın bakım ve güncellenmesi işlemlerini zorlaştırır.

Birleştirme (Composition):

Javada bir sınıf başka bir sınıfı değişken olarak içeriyorsa bu durum “has a” olarak tanımlanır. Sahiplik ilişkisi vardır.

KALITIM (INHERITANCE):

Bir sınıf başka bir sınıfın değişken ve metodlarına sahipsa aralarında “is a” ilişkisi vardır ve buna kalıtım denir.

Tek yönlü kalıtım:

Bir sınıfın başka bir sınıfı genişlettiği alt ve ata sınıf ilişkisini ifade eder.

Çoklu Kalıtım (Multiple Inheritance):

Java çoklu kalıtımı desteklemez, çoklu kalıtım için interfaceler kullanılır.

Çok seviyeli Kalıtım (Multilevel Inheritance):

Bir sınıfın özelliklerini miras aldığı sınıfın, başka bir sınıfın özelliklerini miras almasıdır. C sınıfı B sınıfının özelliklerini, B sınıfı da A sınıfının özelliklerini miras alıyorsa, burada çok seviyeli bir kalıtım vardır.

Hiyerarşik kalıtım (Hierarchical Inheritance):

Birden fazla sınıfın aynı sınıftan kalıtım alarak genişletilmesidir.

Hibrit kalıtım (Hybrid Inheritance):

Birden fazla kalıtım türünün birleştirilerek kullanılmasıdır.

Kurucu zinciri (Constructer Chain):

Bir sınıf başka bir sınıfın özelliklerini miras alıyorsa, B sınıfı A sınıfının özelliklerini miras alıyorsa, B sınıfının kurucu metodu çalışmaya başladığında önce A sınıfının kurucu metodunu çalıştırı, A sınıfının kurucu metodunun çalışması sonlandıktan sonra B sınıfının kurucu metodunun çalışması sonlanır.Böylece kurucu zinciri tamamlanmış olur.

Super:

Süper komutu ata sınıfın hangi kurucu metodunun çalıştırılacağının belirlenmesini sağlar, eğer B sınıfı A sınıfını miras alıyorsa B sınıfının kurucu metodu içerisinde süper anahtar sözcüpü ile A sınıfının hangi kurucu metodunun çağırılacağı belirtilir, eğer bu belirtilmezse programcı derleme hatası alacaktır.

Metod Overriding (metod geçersiz kılma):

Kalıtımla alınan bir metodun adı ve parametreleriyle aynı özelliklerde bir metodun miras alan sınıf içerisinde tanımlanmasıdır, böylece atadan alınan metod değil miras alan içerisinde tanımlanan metod kullanılır ve atadan gelen metod geçersiz kılınmış olur.

Metod overloading (Metodların aşırı yüklenmesi):

Aynı isimde fakat farklı parametreli ve farklı dönüş tipli metodların tanımlanması metod overloading denir.

Çok Biçimlilik (Polymorphism):

Çok biçimlilik aynı görevin veya işin farklı yollarla yapılabilmesini ifade eder. Nesne, aynı davranışı farklı fromlar ve görünüşler ile yerine getirebilir.

Overriding in JAVA (ezme)

Overloading JAVA (Aşırı yükleme)

Nested ve Inner classlar:

Non-Static

Nested classlar iç içe geçmiş sınıflardır. Bir sınıf kendi başına kullanılamaz durumda ise bunu başka bir sınıf içerisinde tanımlarız ve bu içeride tanımlanan sınıf inner class durumda olur.

class Out{

public int a = 5;

class Inner{

public int a = 10;

public void run(){

System.out.println(a);

System.out.println(this.a);

//Bu iki ekrana yazdırmada da aynı değer döner. Eğer run metodu içerisinde bir a değişkeni tanımlarsak ilk çıktı metod içerisindeli a değerini(local değişkeni) ikinci çıktı ise sınıfa ait olan a değişkenini ekrana yazar. Burada out sınıfı içerisindeki a değerini ekrana yazdırabilmek içn out.this.a komutunu kullanmamız gerekir, out.this ifadesi out sınıfı içerisinde tanımlı olan değişken ve metodlara erişebilmemizi sağlar. Dış sınıfın içteki sınıfın değişken ve metodlarına ulaşabilmesi için iç sınıftan bir nesne üretilmesi gerelir.

}

}

}

Main classdan içerideki sınıftan bir nesne üretebilmek için öncelikle dış sınıftan bir nesne üretmek gerekir, bu ürettiğimiz nesneyi kullanarak iç sınıftan da bir nesne üratebiliriz.

Out o = new Out();

o.Inner inner = o.new Inner();

tanımlama bu şekilde yapılmalıdır. Doğrudan new Inner(); şeklinde tanımlama yapamayız çünkü Inner classa out classından erişiyoruz bu nedenle Out clasından new lememiz gerekir.

Static class:

Eğer içeride tanımladığımız class static olursa bu sınıftan nesne üretmesek bile otomatik olarak ram’de yaratıldığı için içerideki sınıftan nesne üretmeden kullanabiliriz.

Local classlar:

Local classlar bir sınıfa ait metod içerisinde tanımlanmış classlardır, bu classlara sadece içerisinde tanımlandıkları metod’da kullanılabilir ve erişilebilirler, tıpkı meotd içerisinde tanımlanan değişkenler gibi.

Anonim classlar:

Javada bir class’dan nesne ürettikten sonra, ürettiğimiz bu nesnenin içerisine sanki ilgili class ın kod blokları arasında kod ekliyormuş gibi metodları @override edebiliriz. Ancak ana tanımlamada olmayan bir metod tanımlayıp bunu kullanamayız. Değişken tanımlaması da tapılabilir ana class’da var olan bir değişknele aynı isimde bir değişken tanımlanırsa nesne özelinde tanımladığımız değer baz alınarak çıktı sağlar.

class Anonim{

public int x = 20;

public void print(){

System.out.println(“Anonim class içerisinde tanımlı print”);

}

}

public class main{

public static void main(String [] args){

Anonim anonim = new Anonim(){

İnt x = 10; //bu şekilde değişken tanımlaması yapılabilir.

// buraya ekstradan metod override edebiliriz, değişken tanımlayabiliriz. Ancak ana sınıf içerisinde olmayan bir metodu tanımlarsak dışarıdan erişemeyiz.

@override

public void print(){

System.out.println(“Nesne içerisinde override edilen print metodu.”);

System.out.println(this.x);//bu kullanım bize nesne özelinde tanımladığımız x değerini döndürür.

}

Public void print2(){

System.ut.println(“Ana class içerisinde olmayan metod tanımlaması”)

}

}

anonim.print();//bu metod bize nesne özelinde override edilen halini döndürür.

anonim.print2();//bu şekilde kullanım mümkün değildir kullandığımız metodun ana class içerisinde de tanımlı olması gerekir.

}

}

İnner classlar eğer Şu şekilde tanımlanmış ise:  
İçerideki Private class’a ait powerof2 metoduna erişebilmek için, öncelikle bir adet Inner nesnesi üretilmesi gerekir.

Class Solution{

public static void main(String[] args){

int num = 7;

Inner inner=new Inner()// inner nesnesi üreterek Inner class içerisindeki Private class’a erişim sağlanabilir.

Object o =inner.new Private();// ürettiğimiz inner nesnesini kullanarak Private class’ına erişim sağlıyoruz.

System.out.println(num+“ is ”+(Solution.Inner.Private)o).powerof2(num));//Burada println metodu içerindeki (Solution.Inner.Private)o).powerof2(num) tanımlaması o değişkeninin tuttuğu nesnenin gerçek türünü Solution.Inner.Private olarak dönüştürmeye çalışır, ardından bu dönüştürülen nesne üzerinden powerof2 metodunu çağırır. Burada dikkat edilmesi gereken şey (Solution.Inner.Private)o) kodu o değişkenin tutuğu nesnenin Solution.Inner.Private olup olmadığını kontrol etmediğidir, eğer o değişkeni farklı türden bir nesneyi tutuyorsa “ClassCastException” hatasına sebep olur. Bu hatanın Önüne geçmek için “instence” operatörü ile o değişkeninin doğru türde nesne tutup tutmadığı kontrol edilmelidir.

}

Static class Inner{

private class Private{

private String powerof2(int num){

return((num&num-1)==0)?”power of 2”:”not a power of 2”;

}

}}}

Wrapper Classlar:

Primaitive değişkenlerden obje yaratmamıza imkan veren classlar wrapper classlardır.

Primitive(byte, short, int, long, double, float, char, boolean).

Wrapper(Byte, Short, Integer, Long, Double, Float, Character, Boolean).

Generic yapılarda ilkel veri tiplerinin kullanımına izin verilmediği için wrapper classlar kullanılır. Bir değişkeni Integer veya int diye kullanmak değişkeni kullanım açısından bir fark yaratmaz.

Boxing-Autoboxing-Unboxing:

Primitive tipdeki bir değişkenin Wrapper nesnesine dönüştürülmesi “boxing” wrapper nesnesinin primitive tipe dönüştürülmesi ise unboxing işlemidir.

İnt a = 10; // primitive değişken tanımlamasıdır.  
Integer b = 20; //AutoBoxing işlemidir, çünkü herhangi bir dönüşüm kodu yazmayız.

Integer c = Integer.valueOf(a); // Bu boxing işlemidir, çünkü dönüşüm manuel yazılır.

Generic Yapılar:

Classlar içerisinde normalde kullandığımız değişkenlerin bir veri tipi olur. Ancak Generic classlarda belirli bir veri tipi olmadan tanımlamalar yapılabilir. Bunun için tanımlama:

public class Nullable <T>{

private final T value;

public Nullable(T value){

this.value=value;

}

}

Bu tanımlamada class adı yanındaki <T> belirsiz olarak tanımladığımız değişken tipinin adıdır, T yerine istediğimiz herhangi bir isim verebiliriz, ama genel kullanım T dir. Bu T class içerisindeki değişkenlerin veri tipini oluşturur ve böylece herhangi bir veri tipini girdi olarak alabilir.

Generic classlarda değişken tipi illa belirsiz T gibi bir değer olmak zorunda değildir, wrapper classlardan veya kendi oluşturduğumuz herhangi başka bir class ı da veri tipi olarak yazabiliriz. Ancak generic classlar primitive tipleri kabul etmez bu nedenle wrapper classları kullanmak şarttır. <> işaretleri arasına istediğimiz kadar tip yerleştirebiliriz, public class generic <T1, T2, T3…> gibi tanımlamalar yapabiliriz. Elbette bu veri tipleri T1, T2 şeklinde olmak zorunda değildir, yine kendi oluşturduğumuz bir class veya wrapper classları da yazabiliriz.

Generic Metodlar:

Herhangi bir değişken tipiyle işlem yapan metodlar generic metodlardır. Generic metodların tanımlandıkları sınıfların generic yapıda olmasına gerek yoktur (Abstract classlarda metodların da abstract olması).

public static <T> void printArray(){

}

Tanımlama ve keywordlerin sıralaması bu şekilde olmalıdır.

Kullanım

Print.printArr(); şeklindedir ve içerisine yazdığımız değişkenin veri tipi otomatik olarak T’ye atanır. Generic classlarda yaptığımız <Integer> benzeri bir tanımlamaya gerek yoktur, metodlar doğrudan kullanılabilir.

Generic Interfaceler:

Generic interfacelerin tanımlanması da generic classların tanımlanması gibidir, ancak implements eden classların tanımlanması normal interfacelerdeki gibi olmaz. Tanımlama:

public class Student implements IDataBase<>{

} Şeklinde tanımlamak gerekir ve interfacenin yanındaki elmas işaretine de kullanmak istediğimiz veri tipini yazarız. Oraya yazdığımız veri tipine göre generic yapıdaki metodlar override edildiğinde generic yapıdan tanımladığımız değişken tiğine dönüşürler. Eğer implement ettiğimiz metodların interfacedeki gibi generic yapıda kalmasını istiyorsak, interface’i implements eden class’ın da generic tanımlanması gerekir. Tanımlama:

public class Student <T> implements <T>{

} Bu şekilde tanımladığımızda impelement edilen metodlar interface’de tanımlandıkları gibi generic yapıda kalırlar.

Generic Bounded Types(Sınırlandırılmış generic tipler):

Generic yapılarda, veri tiplerini sınırlandırabiliriz, örneğin generic bir sınıfı sadece sayısal olan wrapper class tiplerinde tanımlanacak şekilde tanımlayabiliriz. Sınırlandırılmış tanımlama:

public BoundedNullable <T extends Number>{

} Böylece, classın yalnızca Number classından extends edilmiş class tiplerini kabul etmesini sağlamış olduk.

Exception Handling:

Exception Türkçesi istisna olan durumlardır. Programın normaş akışını beklenmeyen, hatalı bir şekilde kesen durumlara karşılık gelir. Ör: bir dosya okuması yaparken dosyanın harddiskte olmaması hatası.Hata yönetimi javada iki şekilde yapılabilir, ilk olarak hata alındıktan sonra program çalışmaya devam eder ve hata çözülmeye çalışılır. Veya program çalışmayı durduru.

Unchecked Exceptions: Programın derlenmesi sırasında bilinmeyen ancak program çalıştığı esnada ortaya çıkan hata tipleridir.

Checked Exceptions: Derleme aşamasında tespit edilen hatalardır. Bu hataların alınabileceği önceden bilinir.Ör: dosya açma işlemi java’da hata oluşturabilecek bir işlemdir. Bu fonksiyonun hata fırlatabileceği önceden belirtildiği için kodu yazarken java geliştirme platformu ona göre önlem almamızı ister.

Throwable: Exception hiyerarşisinin en üstündeki sınıftır Tüö exception sınıfları ondan kalıtım alır.

Error: Programdaki ciddi hataları temsil eder. JVM tarafından iletilen uygulama dışında oluşan hatalardır.Bu tip hatalar da “Unchecked Exceptions” tipindedir.Ör: Veri tabanı sınıcısına bağlanmaya çalışınca bağlantı hatası verşrse bunu ancak çalışma zamanında anlayabiliriz.

Exception: kullanıcı tanımlı exception sınıfları dahil olmak üzere tüm exception alt sınıflarının ATA sınıfıdır.”RuntimeException” dışındaki tüm exception hatları “Checked Exception” tipindedir.

RuntimeException:Geçersiz ya da hatalı bir işlem sonucunda uygulamada oluşan hatalardur. Bunlar da JVM tarafından fırlatılır. “Unchecked Exceptions” kategorisine girer.

Try-Catch-Finally:

Try catch blokları hata oluşabilecek yerlerde kullanılan bloklardır, try kısmı hata oluşabilecek kodları kapsar, catch ise hata yakalandığında yapılacak işlemin belirtildiği kısımdır.

Her yerde try catch kullanılması maliyeti artırır, bu nedenle sadece gerektiği yerde kullanılmalıdır.

Metodlar try catch içerisine alınamaz, yalnızca metodun içindekiler try catch içerisine alınabilir.

Catch parametresi olarak farklı türde hata tiplerini alabiliriz, ancak bir tipte hata nesnesi yakalayacak şekilde yazarsal diğer hatalarda bir işlem yapamaz. Spesifik olarak birden fazla hatayı yakalamak istiyorsak, her bir hata için bir catch bloku yazmamız gerekir, veya tek bir catch yazıp parametresini Exception türünden alırsak tüm hatalara tek bir karşılık vermiş oluruz.

Finally bloku:

Try catch blokları sonunda hata olsa da olmasa da çalışmasını istediğimiz kodların olduğu bloktur.

Throw:

Try-Catch blokları haricinde bir hata yakalama yöntemi de throwdur.

Bunun kullanımı için metodlara throws Exception ile metoda hata atılabilecek sınıfların dahil edilmesidir.Runtime olmayan hataların fırlatılabilrmesi için gerekir.

Ör:throw new Exception(“Yaşınız tutmuyor”);

Kendi Exception sınıfımızı yazmak:

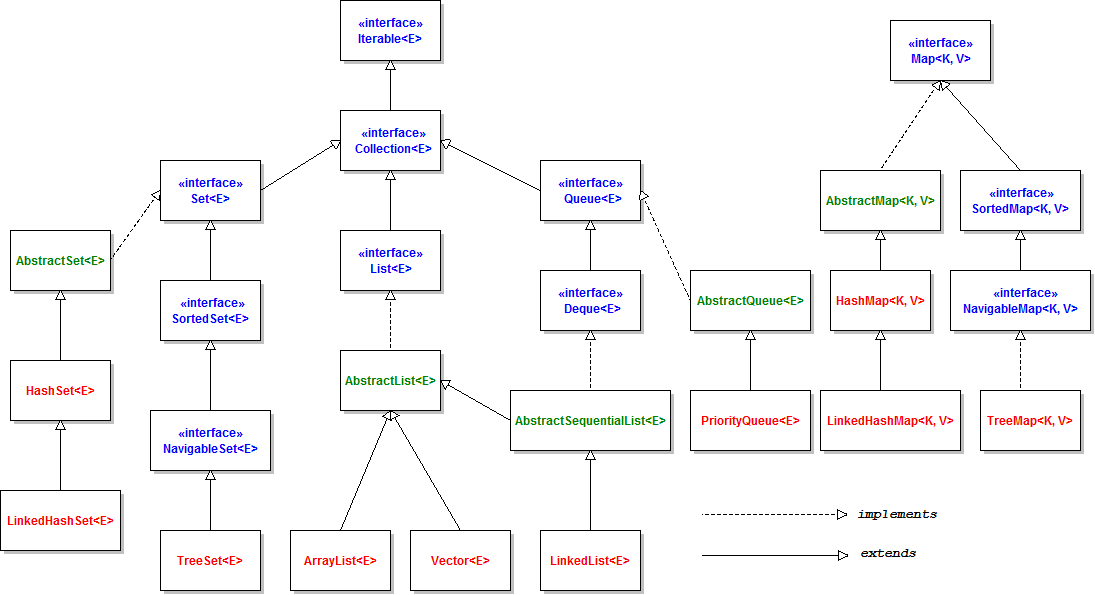
Oluşturduğumuz class ı Exception classından extends etmemiz gerekir. Temel bir sınıf i.in yalnızca constructerinde super metodunu kullanabiliriz, böylece hata olması durumunda vermek istediğimiz hata mesajını oluşturabiliriz. Exception sınıfından gelen metodları override edip kendimize göre ayarlayabiliriz.

Metodları ve Classları hata fırlatabilecek şekilde yazmak:

Metodlarımıza veya Classlarımıza throws anahtar kelimesi ile birlikte fırlatacığı spesifik hata türünü veya Exception ile herhangi bir hata fırlatabileceğini gelirtebiliriz, böylece trycatch kullanmadan exception fırlatabiliriz.

Collections:

Collectionlar arraylar gibi veri tutmamızı sağlar fakat arraylerden farklı olarak boyut sınırı olmayan yapılardır. Ekleme, silme, sıralama, arama yapma vs. İşlemleri kolayca yapmamızı sağlarlar.



Iterable interface’i diğer listelerde gezinebilmemizi sağlayan ana classtır. Colectionlar generic yapıdadırlar yani tüm wrapper classlar ve kendi oluşturduğumuz veri tipleriyle çalışabilirler.

Collectionlar, set(kümeler), list(listeler) ve queue(kuyruklar) türlerinden oluşurlar.

Set(kümeler):

İçerisinde tutulan değerler unique değerlerdir, tekrar eden değerler içermezler ayrıca genel olarak setler sıralama garantisi vermez, a b c sırasıyla eklediğimiz değerler aynı sırayla geri döndürülemeyebilir.

List(Listeler):

Listeler en çok kullanılan collection türleridir. Aynı veri türünde tekrar eden veriler de ekleyebiliriz, benzersizlik şartı yoktur. Veriler eklendikleri sırada geriye döndürülürler.

Queue(Kuyruklar):

FIFO mantığıyla çalışır, ilk giren ilk çıkar mantığıyla ilk giren değerden son giren veriye göre çalışır.

Map(Adresleme):Key => Value ilişkisine göre çalışır, değerler ile anahtarlar ilişkilidir, her değer için benzersiz bir anahtar değeri vardır ve bu anahtar değeriyle o veriye ulaşılır.

Set interface:

Collection interface’den kalıtım almıştır. Aynı elemanların veri kümesi i.inde tekrar bulunmasına izin vermez. Eleman tekrarının olmaması için nesnelerin equals ve hashCode metodlarının tanımlı olması gerekir. Haslerde Null değeri de tutulabilir ancak yalnızca bir tane Null değeri olabilir benzersizlik sağlanması için.

Tanımlama: HasSet<Integer> hasSet = new HasSet<>();

Boyut sınırı yoktur ve istenilen veri tipiyle tanımlanabilir.

Iterator:

İterator collection classların atasıdır ve collection classlar içerisinde gezinmemizi sağlar.

hasNext() metodu sonrasında bir değer olup olmadığını true veya false olarak bildirir.

İterator.next() metodu sonraki elemana geçer ve değeri geriye döndürür.

Linked Hasset:

LinkedHasSet HasSet classının bir alt classıdır.HasSet’in tüm özelliklerini sağlar, tek farkı eklenene verilerin sıralanmasını sağlamasıdır.

TreeSet:

TreeSet de LinkedHasSet gibi HasSet classının özelliklerine sahiptir, TreeSet’in LinkedHasSet’den farkı sıralama şartını kendi isteğimize göre belirleyebiliyor olmamızdır. LinkedHasSet’te yalnızca ilk giren ilk çıkar sıralaması vardı. Sıralamanın neye göre yapılacağını belirtmezsek program hata comparate hatası verir, bu nedenle Comparator Interface’inden implent ettiğimiz bir class kullanarak sıralmayı bu class üzerinden sağlayabiliriz. Ör:

TreeSet<Student> students = new TreeSet<>(new OrderNoteComparator());

Bu kullanımda OrderNoteComparator içerisinde bulunan comparator metoduna göre sıralama yapar, tersi için

TreeSet<Student> students = new TreeSet<>(new OrderNoteComparator().reversed());

Kullanılabilir, bu şekilde ters sıralama yapar.

Stringe göre sıralam yaparken compare metodunun return değerini:

O1.getName().compareTo(o2.getName); şeklinde olmalıdır. compareTo iki String ifadenin karşılaştırmasını yaparak tree’ye ekler.

Arraylist:

Normal arraylerin objeleşmiş şekilde tutulmasıyla oluşan listelere arraylist denir. Başlangıç değeri 10dur ve 2 katı şeklinde boyutu artar. Listedeki bir değeri elde etmeye çalıştığımızda o değerin index numarasına ihtiyacımız olur. Bu değeri elde etmek için 2 yol vardır, arraylistte tekrar eden değerlere izin verildiği için index numarasını elde ederken, listenin başından başkayıp ilk elde edilen değerin veya son elde edilen değerin index numarasını döndürebilir.

liste.indexOf(5): bu metod listenin başından başlayıp, ilk yakaladığı 5 değerinin index numarasını geri döndürür.

liste.lastIndexOf(5): bu metod listenin sonuna kadar gider ve en son yakaladığı 5 değerinin index değerini döndürür. Bu iki metod da listede olmayan bir değer için -1 değerini döndürür.

Arrayliste değer eklerken iki seçenek vardır birisi doğrudan sona ekler, diğeri belirtilen index numarasına değeri ekler ve diğer değerleri öteler.

Listede bir değerin var olup olmadığını kontrol etmek için contains metodu kullanılır, true veya false değeri döndürür.

Remove metodu ile istersek belirli bir index numarasıyla silme işlemi yapabiliriz, veya bir obje verip o objenin silinmesini sağlayabiliriz.

addAll metodu ile aynı veri tipindeki başka bir listenin tüm elemanları listeye eklenebilir, listede var olan derğerlere bir şey olmaz, addAll metodu sonrasında listeye add metodu ile değer eklmeye devam edebiliriz. Ama istersek addAll metoduna bir index numarası verebiliriz bu da listemizin hangi indexsinden başlayıp ekleme yapacağını belirtir, eğer 0 dersek listemizde varolan değelerin üzerine yazma işlemi yapar.

subList metodu, listemizin belirli bir başlangıç ve bitiş indexsine göre sublistini döndürür.

LinkedList:

Bellekte bir bağlı liste şeklinde verileri tutar, diziler gibi arda arda sıralı bellek adreslerinde tutulmazlar ve bir boyut belirlemek gerekmez, ram kapasitesi kadar artar.

Vektor:

Vektor classı arraylist ile aynı özelliklere sahip olmasıyla bazı farklılıkları vardır. En önemli farklı senkronize olmasıdır, senkronize olması demek birden fazla iş parçacığı ile çalışıldığı zaman aynı anda birden fazla iş parçacığının vektore erişmesi ve değişiklik yapması işlemi senkronizdir yani bir iş parçacığının yaptığı değişiklik diğer iş parçacığı için de yapılmış olur böylece güvenli bir şekilde işlem yapmılmasını sağlar. Ancak arraylistte böyle bir senkronizasyon olmadığı için çoklu iş parçacığında kullanılması güvenli değildir. Fakar arraylist senkronize olmadığı için Vektorden daha hızlıdır. Kısaca çoklu iş parçacıklarıyla çalışıldığında Vektor tek iş parçacığı ile çalışıldığında ArryList kullanılmalıdır.

QueueLinkedList:

First in First out mantığıyla çalışır. Add() metodu ile ekleme yapıldığında ekleme yapılamazsa hata verir, effor() metodu ile ekleme yapıldığında ekleme yapılmazsa false değeri döndürür.remove() metodu baştaki elemanı verir ve onu kuyruktan atar, eğer kuyruk boş ise hata verir,element() metodu kuyruğun başındaki elemanı verir ama onu kuyruktan atmaz, kuyruk boş ise hata verir.poll() baştaki değeri verir kuyruk boşsa false değeri döndürür ve baştaki değeri kuyruktan siler.peek() baştaki elemanı verir ama onu kuyruktan silmez, kuruk boş ise false değeri döndürür.

PrioryQueue:

Normal kuyruk yapısına ek olarak öncelik belirleyip sıralamayı bu önceliğe göre yapabiliriz, yani ilk giren ilk çıkar değil öncelikli olan ilk çıkar mantığıyla çalışıyor.

Maps:

MapInterface ve HasMap:

Generic olarak bir Key ve Bir Value değeri tutan yapılardır, Key değerine göre veya Value değerine göre arama yapılabilir. Ekleme işlemi sırasında hem Key hem de Value değeri eklenmelidir.Maplerde iterator bulunmaz iterator Collectionlara özeldir.LinkedHasMap ve TreeMap:

HasMaplerden farklı olarak bu yapılarda sıralama yapılabilir, LinkedHasMaplerde ilk eklenen ilk çıkar, TreeMaplerde kendi sıralma algoritmamızı oluşturabiliriz. Sıralama algoritması oluşturmak için iki yöntem vardır, Comparator veya Comparable Interfacelerinden faydalanırız,

Comparable Interface’i istediğimiz TreeMap türü olacak olan Class’a imlepents edebiliriz ve compareTo metodunu override ederek istediğimiz algoritmayı oluşturabiliriz.

Comparator Interface’ini ise TreeMap tanımlarken anonim class türetmek için kullanırız. Anonim class içerisinde compare metodu override edilerek istediğimiz sıralama değişikliğini belirleyebiliriz.  
Comparable varsayılan listelemeyi belirlemek için kullanılırken, Comparator varsayılan sıralamayı değiştirmek için kullanılır.

I/O Veri Akışları:

Kullanıcıdan veri alınması input stream olarak adlandırılır, çıktı verilmesi output stream olarak adlandırılır. Veri akışında iki tip akış vardır, byte akış ve character akışı.

Byte Stream:

Akışı tek baytlık veriyi okumat ve yazmak için kullanılır, karakterden oluşmayan, resim, müzik gibi verilere Byte Stream ile input ve output işlemleri gerçekleştirebiliriz. Tüm byte akış sınıfları InputStrean ve OutputStream abstract classlarından türetilirler.

Character Stream:

Character stream karakterden oluşan veriler üzerinde input,output işlemleri gerçekleştirebilmemizi sağlar. Unicode kullandığı için farklı dillerin alfabelerine uyum sağlar.Tüm karakter akışı sınıfları, Reader ve Writer abstract sınıflarından türetilirler.

File Class’ı:

Java.io.file class import edilerek dosyalarla ilgili işlemleri yapabiliriz. Bir dosya oluşturmak için File classından gelen createNewFile() metodunu kullanabiliriz, bu metod eğer oluşturmak istediğimiz dosya yoksa oluşturur ve geriye true değeri döndürür, eğer oluşturmak istediğimiz dosya zaten var ise false değeri döndürür.

//Sadece dosya ismi vererek veya yol belirterek oluşturabiliriz.

File file = new File(“Oluşturmak istediğimiz dosyanın ismi ve uzantısı”);

//Dosya oluşturma metodu

file.createNewFile();

//Dosya silme metodu

file.delete();

File file = new File(“Deneme”)

//bu metod ile belirlediğimiz isimle bir dizin oluşturabiliriz.

file.mkdir() ;

//Bu metod ile içe içe dizinleri tek seferde oluşturabiliriz.file.mkdirs();

//Dizin’in alt klasörlerini listeleme

File file = new File(“Dizin adı”)

String[] liste = dizin.list();

For(String str: liste){

System.out.println();

}

FileInputStream:

InputStream’den miras alan bu sınıf byte byte veri almamızı sağlar. Contructer içerisine bir dosya objesi veya dosya adı yazılabilir.

try{

FileInputSteam input = new FileInputStream(“deneme.txt”)

}carch(FileNotFoundException e){

System.out.println(e.getMessage());

}

//dosyadan byte olarak beri okumak için kullanılır tek karakter okur

input.read();//sayısal bir değer döndürür.

// dosyada okunabilecek kaç adet karakter olduğunu sayısal olarak verir.

İnput.available()

//skip dosya okunurken atlanacak karakter sayısını girmemizi sağlar aşağıdaki kodla 5 karakter atlamayı sağlar.

İnput.skip(5)

FileOutputStream:

FileOutputStream ile dosyaya yazma işlemi gerçekleştirebiliriz, eğer içerisindeki verileri silmeyip dosyaya ekleme yapmak istiyorsak, metoda true parametresi eklememiz gerekir.

try {  
 FileOutputStream output=new FileOutputStream("deneme.txt",true);  
} catch (FileNotFoundException e) {  
 throw new RuntimeException(e);  
}

String değerleri FileOutputStream ile yazdırmak için önce byte a çevirmemiz gerekir.

String text="\n Deneme yazısı, bu yöntem ile dosyaya yazma işlemi gerçekleştirilir.";  
byte[] bytesOftext=text.getBytes();  
output.write(bytesOftext);  
output.close();

ByteArrayStream:

ByteArrayInputStream

Bir dizi girdi verisini bayt cinsinden okumaya yarar. Girdi akışını byte dizisi kullanarak oluşturur. Bu byte dizisinin verilerini depolamak için hahili dizi içerir.Genellikle şifrele işlemlerinde daha yaygın kullanılır.

byte[] array = {1, 2, 3, 54, 66};  
ByteArrayInputStream byteArray = new ByteArrayInputStream(array);

bu şekilde doğrudan kullanılabildiği gibi belirli bir başlangıç değeri ve aralıkla da kullanılabilir.

byte[] array = {1, 2, 3, 54, 66};  
ByteArrayInputStream byteArray = new ByteArrayInputStream(array,2,3);

bu kullanımda 2. İndexten başlayıp 3 elemanı okur.

ByteArrayOutputStream

Bir dizi çıktı verisini bayt olarak yazma işlemini gerçekleştirmeye yarar. Class içerisinde veri depolamak için dahili bir byte dizisi bulunur.

Serialization

Javada bazen kullandığımız nesneleri bir dosyaya ve farklı bir yere yazıp daha sonrasında tekrar okumamız gerekebilir, bu durumda okuduğumuz nesnenin tipinin ne olduğunu anlayamayız bu durumda serialization API ile bir nesneyi olduğu gibi bir dosyaya yazma ve sonrasında aynı şekilde okuma yapabiliriz, bu işleme object serialization denir.

ObjectOuputStream

Javada serializable Interface’inden implements eden bir class’dan üretilen nesneleri bir dosyaya nesne olarak yazabilmemizi sağlar, constructor’ında bir FileInputStream tipinde değer alır ve belirtilen file’a nesne yazar.

FileOutputStream outputStream=new FileOutputStream("araba.txt");  
ObjectOutputStream output = new ObjectOutputStream(outputStream);  
output.writeObject(c1);  
output.close();

ObjectInputStream

Bu Sınıf da dosyadan nesne okuyabilmemizi sağlar,

FileInputStream inputStream = new FileInputStream("araba.txt");  
ObjectInputStream objectInputStream = new ObjectInputStream(inputStream);  
Car newCar = (Car) objectInputStream.readObject();

BufferredInputStream

Dosya okuma işlemlerin daha hızlı bir şekilde gerçekleştirilebilmesi için, bellek ve program arasında bir tampon görevi görür. Kullanım amacı Okuma işlemlerinini daha hızlı gerçekleştirilmesidir.

FileInputStream fileInputStream = new FileInputStream("deneme.txt");  
BufferedInputStream buffInput = new BufferedInputStream(fileInputStream);

BufferredOutputtStream

Dosya yazma işlemi için BufferedInputStream ile aynı etkiye sahiptir.

PrintSteam:

İçerisine atılan ilkel veritiplerini String türüne çevirir ve bir çıktı akışı oluşturur. Herhangi bir hata durumunda geriye hata fırlatmaz, ancak contructor tanımında örneğin bir file kullanılacaksa file’ın var olmaması durumu olduğu için nesne üretirken hata fırlatabilir.

PrintStream output = new PrintStream("print.txt");  
output.print(12345);

InputStreamReader:

Bir karakter akışını temsil eden class’dır, byte array class’ından farkı char set kullanılmasıdır. Varsayılan chatseti UTF8’dir, eğer istersek constuctor’ında kendi chatsetimizi de verebiliriz. Verdiğimiz chatset bulunamazsa, UnsoppurtedEncodingException hatası fırlatır.

InputStreamReader inputReader=new InputStreamReader(fileInput,"UTF8");  
System.*out*.print("Karakter seti:");  
System.*out*.println(inputReader.getEncoding());  
int i =inputReader.read();  
while (i!=-1){  
 System.*out*.print((char)+i);  
 i=inputReader.read();  
}

OutputStreamWrite:

StreamReader’dan farklı olarak bu class ile yazma işlemleri gerçekleştiririz. Bu class’ın constuctor’ında da chatset belirleyebilirz.

FileOutputStream fileOutput=new FileOutputStream("output.txt");  
OutputStreamWriter fileWriter= new OutputStreamWriter(fileOutput, Charset.*forName*("UTF8"));  
fileWriter.write(data);  
fileWriter.close();  
fileOutput.close();

FileReader:

Dosyadan veri okuyabilmemizi sağlayan bir sınıftır, constructor’ında chatset de belirleyebiliriz.

FileReader readFile = new FileReader("deneme.txt", Charset.*forName*("UTF8"));

FileWriter:

Dosyaya veri yazmamızı sağlayan bir class’dır, kendi içerisinde bir FileOutputSream nesnesi barındırır ve yazma işlemini bunun üzerinden gerçekleştirir.

FileWriter writerFİle=new FileWriter("fileWriterOut.txt");  
writerFİle.write(data);

BufferedReader:

Okuma işlemlerinde FileReader ile birlikte kullanılarak okuma işlemini hızlandıran bir class.

FileReader readFile = new FileReader("deneme.txt");  
BufferedReader readBuff = new BufferedReader(readFile);  
String line = readBuff.readLine();

BufferedWriter:

Yazma işlemini hızlandıran bir class, FileWriter ile birlikte kullanılır.

FileWriter writeFile=new FileWriter(file, Charset.*forName*("UTF8"));  
BufferedWriter writeBuff=new BufferedWriter(writeFile);  
writeBuff.write(data);

PrintWriter:

İçerisine attığımız her şeyi Stringe çevirip dosyaya yazma işlemi gerçekleştirir.

PrintWriter writer=new PrintWriter("output.txt");  
writer.write(data);  
writer.close();

Lambda Expressions:

Lambdadan önce Functional Interface’in ne olduğunu bilmek gerekir. Functional Interface, içerisinde yalnızca bir adet abstract metodu olan intefacelere denir. Bu da anonim classlar tanımlanmasını sağlar.

Ör:

Runnable r1 = () -> System.out.println(“Lambdalı”);  
r1.run;

Bu kullanım Şununla aynıdır:

Runnable r1= new Runnable() {

@override  
public void run(){

System.out.println(“Lambdasız”);

}

};  
r1.run();  
Bu kullanımda, zaten Runnable interface’i içerisinde tek bir metod bulunduğu için -> işaretinden sonra yazdığımız şeyler o metod için geçerli olur.