

Java Effective

Java Effective	1
Github linki:	5
https://github.com/mrsonmez10/effective-java-deep-tutorial	5
Chapter 2: Creating and Destroying Objects	6
Item 1: Static methods vs constructors	6
Item 2: Builder Design Pattern	6
Item 3: Singleton Design Pattern	8
Item 4: Java Utility Classes	9
Item 5: Dependency Injection	10
Item 6: Autoboxing & Unboxing	12
Item 7: Garbage Collector	12
Item 8: Avoid finalizers and cleaners	13
Item 9: Prefer try with resources to try-finally	13
Chapter 3: Methods Common to All Objects	15
Item 10: Obey the general contract when overriding equal	15
Item 12: ToString	16
Item 13: Shallow Copy vs Deep copy	16
Item 14: Comparable	17
Chapter 4: Classes and Interfaces	17
Item 15: Minimize the accessibility of classes and members (Access Modifiers)	17
Item 16: In Public classes, use accessor methods, not public fields (Encapsulation)	18
Item 17: Minimize mutability (Mutable - Immutable objects)	19
Item 18: Favor composition over inheritance (composition vs inheritance(kalıtım))	19
Item 19: Design and document for inheritance or else prohibit it (inheritance kullanırken dikkat edilmesi gerekenler)	20
Item 20: Prefer Inheritance to abstract classes (abstract vs inheritance) (Bunun içinde biraz item 21 de var)	21
Item 22: Use interfaces only to define types (Doğru interface kullanımı nasıl olmalı?)	21
Item 23: Prefer class hierarchies to tagged classes (java class hierarchy)	22
Item 24: Favor static member classes over nonstatic (Inner classes)	22
Item 25: Limit source files to a single top level class (2 sınıf bir dosyada tanımlamayın)	23
Chapter 5: Generics	24
Item 26: Don't use raw types (Java Raw Types)	24
Item 27: Eliminate unchecked warnings - Supress Warnings	24
Item 29: Favor generic types (Java generics)	25
Item 30: Favor generic methods (Generic Methods in Java)	25
Item 31: Use bounded wildcards to increase API flexibility (Java Generics WildCards) – Zor Ders	26
Item 32: Combine Generics and varargs judiciously (Java Varargs) – Zor Ders	27
Item 33: Consider typesafe heterogeneous containers (Java Typesafe Heterogeneous Map)	27

Chapter 6: Enums and Annotation	28
Item 34: Use enums instead of int constants (Java enums and Strategy Design Pattern)	28
Item 35: Use instance fields instead of ordinals (Java Enum Ordinal)	30
Item 36: Use EnumSet instead of bit fields (Java EnumSet)	30
Item 37: Use EnumMap instead of ordinal indexing (Java EnumMap) – Biraz zor ders	31
Item 38: Emulate extensible enums with interfaces (Java Enum with Interfaces)	33
Item 39: Prefer annotations to naming patterns(Java Reflection / Yansıma) – Mülakat Soru Adayı	34
Item 40: Consistently use the Override annotation(Java notasyonları önemi - Nasıl olsa çalışıyor dersen patlarsın)	35
Item 41: Java Marker Interface - Marker Annotation (Mülakat soru Adayı)	36
Chapter 7: Lambdas and Streams	36
Item 42: Prefer Lambdas instead of anonymous classes (Java anonymous classes and lambda expressions)	36
Item 43: Prefer method references to lambdas (Method reference Java 8 and Lambda)	37
Item 44: Favor the use of standard functional interfaces (Supplier interface, BinaryOperation interface (Elde varsa yenisini yazma))	38
Item 45: Use streams judiciously (Java streams nedir? (Hangi koşullarda kullanmalıyım?))	38
Item 46: Prefer side-effect-free functions in streams (Stream kullanırken dikkat edilmesi gerekenler)	40
Item 47: Prefer Collections to Stream as a return type (Java stream vs Collection (return type)) (Senior mülakat sorusu)	40
Item 48: Use caution when making streams paralel (Java parallel streams (Kullanmadan önce 2 kere düşünün))	41
Chapter 8: Methods	42
Item 49: Check parameters for validity (Java tips for methods (1 dakikanız 1 saatiniz kurtarabilir.))	42
Item 50: Make defensive copies when needed (Java defensive copy (Mutable and Immutable in java))	42
Item 51: Design method signatures carefully (Java method signatures (Metodların altın kuralları))	43
Item 52: Use overloading judiciously (Java method overriding and overloading (Yeni mezun mülakat sorusu))	45
Item 53: Use varargs judiciously (Java varargs yanlış ve doğru kullanım örnekleri)	46
Item 54: Return empty collections or arrays, not nulls (Java do not return null (Return empty collections or arrays))	46
Item 55: Return optionals judiciously (Java optional kullanımı)	47
Chapter 9: General Programming	47
Item 58: Prefer foreach loops to traditional for loops (Java for loop vs foreach loop)	47
Item 59: Know and use the libraries (Java kütüphanelerini bilin ve kullanın (Tekerleği yeniden icat etme))	48
Item 60: Avoid float and double if exact answers are required (Java use BigDecimal for Monetary calculations (parasal işlemlerde double ve float kullanma))	48
Item 61: Prefer primitive types to boxed primitives (Primitive types vs boxed primitives)	48

Item 62: Avoid strings where other type are more appropriate (Java her zaman string kullanmak zorunda değilsiniz)	48
Item 63: Beware the performance of string concatenation	49
Item 64: Refer to objects by their interfaces (Java how to create set and list (Objeyaratmadan interface kontrol et))	49
Item 65-68: Prefer interfaces to reflection (Java Naming Convention - Native Methods - Optimization)	49
Chapter 10: Exceptions	50
Item 69: Use exceptions only for exceptional conditions (Java exceptions nedir - throw, throws, checked, unchecked)	50
Item 72: Favor the use of standard exceptions (Java exceptions detaylı anlatım – Nasıl yazarım? Bilinmesi gereken exceptionlar)	50
Chapter 11: Concurrency (eş zamanlı)	51
Item 78: Synchronized access to shared mutable data (Java concurrency – threads, synchronized, atomicboolean, volatile)	51
Item 79: Avoid excessive synchronization (Java deadlock – Synchronized ipuçları)	53
Item 80: Prefer executors, tasks, and streams to threads (JavaExecuterService - Executor Framework)	53
Item 81: Prefer concurrency utilities to wait and notify (Java wait and notify, ConcurrentHashMap, BlockingQueue, CountdownLatch, Semaphore)	54
Item 82: Document thread safety (Java StringBuffer vs StringBuilder - Document Thread - Thread tips)	54
Chapter 12: Serialization	55
Item 85 - 87: Prefer alternatives to java serialization (Java serialization and deserialization - json vs serialization)	55
Item 87: Considering using a custom serialized form (Java custom serialization nasıl yapabilirim? - Örnek transient kullanımı)	56
Item 88-89 : Write readObject methods defensively (Java serialization - readObject, Instance control, readResolve, Enum types)	57
Item 90 : Consider serialization proxies instead of serialized instances (Serialization Proxy nedir?)	58

Github linki:

<https://github.com/mrsonmez10/effective-java-deep-tutorial>

Chapter 2: Creating and Destroying Objects

Item 1: Static methods vs constructors

Java

```
public static Yemek sıcakYemek(String name){  
    return new Pide() //extends Yemek  
}  
  
public static Yemek soğukYemek(String name){  
    return new Dondurma() //extends yemek  
}  
  
//Kullanımı  
  
Yemek yemek = Yemek.sıcakYemek("kıymalı pide");
```

Neden constructor yerine static factory?

- 1) Constructor'a isim veremem
- 2) Constructor'da alt tipte bir obje üretemem (yukarıdaki pide ve dondurma örneği gibi)
- 3) Durmadan yeni instance ürettiğimde cacheleme şansım yok

Static örneği → Boolean.valueOf();

Gerektiğinde Kullanılmalı. Her zaman gerekmez.

Item 2: Builder Design Pattern

Sorun: Parametreler arttıkça ve isteğe bağlı oldukça her biri için ayrı bir constructor mi yazacağım?

Çözümler:

- 1) Telescope: Evet

```
Java
//zorunlu
private String kahveBoyu;

//isteğe bağlı
private String laktosuz;
private String yumuşak;

public Kahve(kahveBoyu){}
public Kahve(kahveBoyu, laktosuz){}
...

```

2) JavaBeans Pattern

Boş bir constructor oluşturulmalıdır. Diğer alanları setter'larla setle.

3) Builder Pattern

Eğer parametreler genelde lazım ise, çok kullanması önerilmez.

```
Java
public class KahveBuilder(){
    private String kahveBoyu;

    //isteğe bağlı
    private String laktosuz;
    private String yumuşak;

    public static class Builder(){
        private String kahveBoyu;

        //isteğe bağlı
        private String laktosuz;
        private String yumuşak;

        public Builder(kahveBoyu){}
        public Builder laktosuzlar(laktosuz){}
```

```

        //...
        return this;
    }
    public Builder yumuşak(yumuş){
        //...
        return this;
    }

    public Builder build(){
        return new KahveBuilder(this);
    }

}

public KahveBuilder(Builder b){
    this.kahveBoyutu = b.kahveBoyutu;
    //...
}

}

//kullanımı
KahveBuilder kb = new
KahveBuilder.Builder("kahveBoyutu").laktozsuz("laktoz").build();

```

!! Kontroller de eklenebilir Builder kısmındaki metodlara which is good.

Item 3: Singleton Design Pattern

Singleton farklı implementation'lara sahiptir. 3'üne bakacağınız:

- 1) Lazy: Sadece instance çağrıldığında oluşturulur. Multithread'de singleton prensibi çalışmamaktadır.

Java

```
public class DefineSingleton{
```

```

public static DefineSingleton ds;
private DefineSingleton(){}

public static getInstance DefineSingleton(){
    if(ds == null){
        ds = new DefineSingleton()
    }
    return ds;
}
}

```

- 2) Eager: En başta oluşturulmadan instance çoktan oluşturulmuş oluyor.
Çoktan heap'te yer tuttuğu için memory sıkıntısı olabilir.

Java

```

public class DefineSingleton(){
    public static final DefineSingleton ds = new
DefineSingleton();
    private DefineSingleton(){}

    public static getInstance DefineSingleton(){
        return ds;
    }
}

```

- 3) Enum: En kabul görmüş

Java

```

public enum DefineSingleton(){
    INSTANCE;
    //icerisinde normal metodlar devam eder
}

//Kullanım
DefineSingleton.INSTANCE.hashCode()

```

Item 4: Java Utility Classes

Önemli nokta herhangi bir obje oluşturmadan içerisindeki metodu kullanabilmektir.

Örnek: Collections.sort() → Koca collection sınıfını oluşturmadan sadece sort etmek için metodu kullanabilirim.

!!Enum da kullanılabilir. → Örneğini singleton'da görmüştük.

Abstract class da olabilir. Önemli olan instance'ının new ile oluşturulamamasıdır.

Overview. A utility class is a class that is just a namespace for functions. No instances of it can exist, and all its members are static. For example, java.

Item 5: Dependency Injection

Problem: Sözlük ve yazı analizim var. Sözlük sınıfına verilen dilin yazı analizini nasıl yapmalıyım?

```
Java
public class Sözlük{
    String dil;
    //constructor
}
```

1) Utility

Yeterli esneklik yok. Utility'de hangi dil yazıyorsa ona göre analiz etmek zorunda kalıyoruz.

```
Java
public class YazıBilimiUtility{

    //private constructor
    //instance of Sözlük with new Sözlük(türkçe)

    public boolean isValid(){}
}

//Kullanımı
YazıBilimiUtility.isValid()
```

2) Singleton

Utility ile benzer kod ve aynı problem bulunmaktadır.

3) Dependency injection

Daha esnek bir yapı oluşturmuş olduk.

Java

```
public class Yazibilimi{  
  
    Sözlük sözlük;  
  
    public Yazibilimi(Sözlük sözlük){  
        this.sözlük = sözlük;  
    }  
  
    public boolean isValid(){  
  
    }  
    //Kullanımı  
    Yazibilimi yb = new Yazibilimi(new Sözlük("Türkçe"));  
    yb.isValid();
```

Lambda Functions

Örnek kullanım

1)

Java

```
Supplier<String> fs = () -> "Selin";  
fs.get(); //Selin döner
```

2)

Java

```
Supplier<String> fs = () -> Yazibilimi.cökUzunBirFonksiyonıŞsminiKısaltmak();  
fs.get(); //Daha kısa bir hale gelmiş oldu
```

3)

```
Java
double randomDouble = Math.random();

print(randomDouble);
print(randomDouble); //Both give the same result

Supplier<double> r = () -> Math.random();
print(r.get());
print(r.get()) //Both give different results
```

Item 6: Autoboxing & Unboxing

Avoid creating unnecessary objects. → If you have multiple users this may cause memory & performance issues.

Örnek:

String a = new String("selin"); → DON'T DO THIS

String a = "selin" → yes

Note: Rather than creating objects in every function call, you need to create an object and use that in those function calls.

```
Java
private static final Pattern ROMAN = new Pattern("regex expression");
//so you dont have to create this object every time you call isRoman();

public void isRoman(String s){
    ROMAN.matches(s);
}
```

Autoboxing → Javanın arkada primitive type'ı objeye çevirmesidir. ÖRN: long → Long
Burada aslında Long.valueOf() → long'u Long'a döndürür; metodu kullanılıyor.

Unboxing → yukarıdaki işlemin tam tersi yapılmaktadır. i.intValue() → Integer'i int'e döndürür.

Item 7: Garbage Collector

System.gc() —> manuel olarak garbage collectorı tetikler.
Hayat döngüsü biten objeler için garbage collector kullanılır.

ÖRN: `Sıla = new ...();`
`Mehmet = new ...();`

`Sıla = mehmet` —> garbage collector çalışır.

Stack örneği —> array'den pop ettiğimde arraydeki son elemna kalıyor ama referans ettiği bir şey yok ama referens ediliyor. Çözüm: pop ettiğim eleman'ı null'a eşitlemeyeşim ki garbage collector algılayabilirsin.

Önemli Tool: Visualvm — ne kadar heap kullanılıyor ona bakıyor.

Not: Çok yüklü bir objeyi static olarak oluşturmamak gereklidir genelde çünkü garbage collector bunu silemez. Projenin her yerinde kullanılmıyorsa static yapmaya çok.
Videoda verilen arraylist örneği var.

Örn:

- 1) buffer read kullanıp connection'ı kapatmamak
- 2) Cache'lerde iyi sistem kurmazsa

Memory leak olabilir. Garbage collector'a uygun kod yazılmamış demektir.

Item 8: Avoid finalizers and cleaners

Finalizer ve cleaners garbage collector ile birlikte çalışır.

Finalizers java 9 ile birlikte deprecated olur ve yerini cleaners bırakır çünkü tahmin edilemeyen, tehlikeli ve gereksizdirler genellikle.

Cleaners daha az teknik ama yine tahmin edilemezler.
Çok elleşmemek lazımdır.

Item 9: Prefer try with resources to try-finally

Normal try finally yapısı:

```
Java
try{
    //something
}finally{
    //something
}
```

Try with resource

```
Java
try(soemthing){
    //something
}catch{
    //something
}
```

- 1) Gerektiğinde iki finally yazmana gerek duydurmuyor. Tekte halledebilirsin.
- 2) Finally içerisindeki exception try içerisindeki exception kesebilir ve göremeyiz. Try with resources bunu çözer.

Chapter 3: Methods Common to All Objects

Item 10: Obey the general contract when overriding equal

1)

`10 == 20` → reference tip kıyaslama.
`'A' == 'B'` → reference tip kıyaslama.

`==` – Stackte bir kıyaslama yapar.

2)

```
Object a = new Object();
Object b = new Object();
```

`a == b`

Aynı değerlere sahip olsalar bile kıyaslama `false` döner. Çünkü stackde bu iki değer için de farklı referanslar vardır ve bu referanslar heap'de farklı objelere yönelirler.

Objeler `.equals()` metodunu kullanımlı.

- 3) Override edilmezse `equals` ve `hashcode` yanlış değerler verebilir. Override edilmeli.
- 4) Equals override edilirken dikkat edilmesi gerekenler:
 - a) Reflexive: Objek kendisine eşit olabilir. `True` dönmelidir.
 - b) Simetrik: $x = y$ ise $y = x$
 - c) Transitive: if $x = y$ ve $y = z$ then $x = z$.
 - d) Consistent: Equals sonucu sadece obje value'ları değişirse değişimeli.
 - e) Null control: `null` ile eşit mi diye bakıldığında `false` dönmelii.
- 5) Bir class başka bir classlı extend ettiği durumda bu sınıf için `equals` metodunu yazılamaz çünkü simetri kuralını bozar. Bunun yerine injection yapısını kullanarak çözüm üretebiliriz. Sınıfı extend etmek yerine variable olarak sınıf alınabilir.
- 6) Ne zaman `equals` override edilirse, `hashcode` da override edilmeli. Eğer override edilmezse sınıflar aynı değerlere sahip olmasına rağmen farklı hash kodlarına sahip olurlar.

Bunun dezavantajı:

Örnek: `Map<Obj, String>` gibi bir map oluşturup, işlem yapmaya kalktığımızda `put(new Obj)` ve `get(new Obj)` dediğimizde alınan değer `null` gelir aynı değerlere sahip olmasına rağmen.

- 7) Nasıl hashCode yazılır.
 - a) Elemanları al
 - b) Result = Integer.hashCode(eleman1);
 - c) Result = 31*result + Integer.hashCode(eleman2)
 - d) Tüm elemanlara aynı işlem uygulanır.

Bu sadece bir örnektir. Başka yöntemleri de vardır.

- 8) İkinci hashCode yazma yöntemi:
 - a) Objects.hash(elemanlar) – Çok yavaşlatır. Neden? Autoboxing Unboxing.

Item 12: ToString

Her zaman `toString` methodunu override et. Equals ve Hashcode kadar kritik değil fakat yine de önemli.

Item 13: Shallow Copy vs Deep copy

- 1) Shallow copy:

```
Stack st = new Stack();
Stack st1 = st;
```

`st == st1` – true (referansları aynıdır)

Note: `st`'de bir değişiklik `st1`'i direkt etkiler.

- 2) Deep Copy:

```
Stack st = new Stack();
st.name = "furkan"
```

```
Stack st1 = new Stack();
st1.name = st.name;
```

- 3) Clone methodu vardır. Nasıl kullanılabilir. Sınıfın `Cloneable`'ı implement etmesi gereklidir.
- 4) Direkt defaultta gelen clone methodu shallow copy gibidir. Objeler birbirinden etkilenir.
- 5) Override edilmeli.

Item 14: Comparable

- 1) Implements Comparable<T> – compareTo(T)
- 2) Bunun sayesinde bu class kendi içinde kıyaslanabilir, sıralanabilir hale gelir.
- 3) compareTo metodu override edilmeli class yapısına göre. Örnek: Kitap class'ı sayfa sayısına göre karşılaştırılabilir.
- 4) Bunun sayesinde Collections.sort() fonksiyonunu kullanılabılır hale geliriz çünkü sort edilmesi için comparable interface'ini implement etmiş oluruz.
- 5) Implements Comparator<T> – compare methodu var. İki obje alır. Örnek:
`m1.getName().comapreTo(m2.getName)`

Sonra:
`Collections.sort(kiatpListesi, new IsimKiyaslama());`
- 6) Ne zaman ne kullanılmalı?
 - a) Düz bir sıralama varsa değişmeyecekse, comparable kullanılabilir.
 - b) Farklı parametrelere göre farklı sıralamalar yapılacaksa comparator kullanılabilir.
- 7) İkisi birlikte de kullanılabilir:
 - a) Sınıf comparable'ı implement eder.
 - b) Sınıfın içerisinde:
 - i) Private static final Comparator<PhoneNumber> COMPARATOR =
`compareInt((PhoneNumber pm) -> pm.areaCode).thenCompareInt(.....)`
 - c) CompareTo Methodunun içinde:
 - i) COMPARATOR.compare(this, pn)
 - d) Note: YAVAŞ ama kullanılan bir yöntemmiş.

Chapter 4: Classes and Interfaces

Item 15: Minimize the accessibility of classes and members (Access Modifiers)

- 1) Single Responsibility prensibine dayanır.
- 2) Modifiers
 - a) Private

- i) Private metodlara ve variable'lar sadece tanımlandığı sınıfın içerisinde erişilebilir. Dışında ulaşamayız.
 - b) Public
 - i) Her yerden erişilebilir. (Objenin instance'sı üzerinden) (variables, methods, ...)
 - c) Protected
 - i) Aynı paket içerisindeyse erişebilirim.
 - ii) Farklı paketten erişmek için sınıf extend edilir. Bu şekilde erişmeye başlayabilirim.
 - iii) Default olarak bir şeye belirteç (access modifier eklemezsem) protected olarak kabul edilir. Hem variable hem metodlar için.
 - d) Static
 - i) Public static – Sınıfla beraber erişilebilir olur. Örn: SınıfÖrnek.mesaj();
 - ii) Public static final – başka bir değer atanmaz. Method ise override edilemez. Bir sınıf final ise extend edilemez.
 - e) Abstract
 - i) Abstract sınıflar nesnesi oluşturulamaz.
 - ii) Abstract method yazmak için sınıfın abstract olması lazım.
 - iii) Abstract method'un gövdesi olmaz.
 - iv) Extend edilir. Extend eden sınıflar abstract metodları implement etmek zorundadır.
- 3) `Public static final String[] VALUES = new String[]{.....}` – Güvenlik açığı. Neden?
- a) Başka bir sınıfın bu variable'a erişebilirim. Normalde değiştirememem gereklidir ama primitive bir type array. Bu yüzden elemanları değiştirilebilir.
 - b) Çözüm: public yerine private yapılmalı ve unmodifiable liste şeklinde tanımlanmalıdır.

Item 16: In Public classes, use accessor methods, not public fields (Encapsulation)

- 1) Public field tanımladığında variable ismini kullanarak direk değerini değiştirebilir. Bu alanların doğrudan erişilebilir olmaması lazım. Çözüm:
 - a) Alanları private yap.
 - b) Getter, setter koyulur.
 - c) Advantages:
 - i) Setter'da veri kontrolleri yapılabilir. ÖRN: 0'dan küçük değer konulması engellenebilir.
 - ii) Dosya read only veya write only yapılabilir.
 - iii) Data hiding.
 - iv) Dataya direkt erişim keser.
- 2) Public fieldları final olarak belirtip constructor ile birlikte setlersem encapsulation'la yapmak istediğimiz işi bu şekilde de yapmış oluruz. Çok önerilmez.

Item 17: Minimize mutability (Mutable - Immutable objects)

- 1) Bir obje oluşturulduğunda bunun state'in değerini değiştirebiliyorsa mutable(değiştirilebilir)
- 2) Değiştirilemiyorsa immutable.
- 3) Örn: String a = new String("SELIN"); a.toLowerCase() – Bu operasyon ana objede bir değişiklik yapmaz. Bu bir immutable örneğidir.
- 4) Immutable objeler thread safe için önemli.
- 5) Mutable objeler farklı threadler çalışırken sıkıntı çıkarabilir.
- 6) Immutable sınıf örnekleri – private final String, Boolean ...
- 7) Bir sınıfı immutable yapmak için final koymamız gerekiyor. – public final class ... Neden? Final olmazsa extend edilebilir ve bu da yavru sınıfın parent sınıfının immutable özelliğini bozabilir.
- 8) Immutable dezavantaj – bir alan değiştirmek istersek memoryde sıkıntı oluyor. Örn: fact = fact.someMethod()

Item 18: Favor composition over inheritance (composition vs inheritance(kalıtım))

- 1) Inheritace – extends
- 2) Composition – sınıf diğer sınıfın içine alan olarak konulur.
- 3) İnheritance için güzel bir örnek – Kedi extends Hayvan
- 4) Composition için güzel örnek – Kitap sınıfı yaratılır. Kütüphane sınıfı kitapların bir listesini variable olarak tanımlar. Kütüphane extends Kitap gibi bir yapı yanlıştır.
- 5) Hangisini kullanacağımız daha sonra gelecek yeni implementationlar yapıyı nasıl etkiler sorusuya bulunabilir.

Item 19: Design and document for inheritance or else prohibit it (inheritance kullanırken dikkat edilmesi gerekenler)

- 1) Anasınıfı extend eden bir çocuk sınıf oluşturulduğunda ilk önce ana sınıfın constructor execute edilir.
- 2) Örnek:

Ana:

```
Java
public abstract class Ana {

    public Ana(){
        System.out.println("Ana constructor");
        override();
    }
    public abstract void override();
}
```

Çocuk:

```
Java
public class Cocuk extends Ana{

    int age ;
    public Cocuk(int age){
        System.out.println("Cocuk constructor");

        this.age = age;
    }
    @Override
    public void override(){
        System.out.println("Override in child: "+ age);
    }

    public static void main(String[] args) {
        Cocuk c = new Cocuk(32);

        System.out.println(c.age);
    }
}
```

```
}
```

Output:

```
Unset
Ana constructor
Override in child: 0
Cocuk constructor
32
```

- 3) Inheritance'i kullanmamak gerekiyorsa bunun da önünü kesmek gerekiyor. Nasıl?
 - a) Final class
 - b) Private constructor
 - c) Comment girilebilir. Ya da javadoc.

Item 20: Prefer Inheritance to abstract classes (abstract vs inheritance) (Bunun içinde biraz item 21 de var)

- 1) Interface farklı interface'i ve birden fazla interface'i extend edebilir.
- 2) Abstract class. Kesin olması gereken ama sınıfına göre yazılması gereken şeyler varsa. Örn: kredi hesaplama method abstract olursa krediler kendine göre kredi hesaplama için override etmek zorunda.
- 3) Java 8 den sonra gelen özellik: interface'in içinde default method yazabilir hale geldik. Implement eden sınıflar bu metodu tekrar override etmesine gerek yoktur.

Item 22: Use interfaces only to define types (Doğru interface kullanımı nasıl olmalı?)

- 1) App için belli sabitlerin varsa bunları tutmak için interface yapısı kullanma!
 - a) Düşünce nedir. Interface'e bu sabitleri yazarım. Başka bir sınıfın içinde implement edip kullanırmı. Bu yanlış!!! Antipattern bir yapı olur.

- b) Bir sınıfın diğer sınıfla bir işi yoksa implement de etmesin!
 - c) Güvenlik problemleri. Implement eden sınıf üzerinden sabitler dışarı açıldı. Başka bir yerde implemetn eden class instance'ı üzerinden bu değerlere ulaşabilirim. Olmamalı.
- 2) Sabit değerler için ne yapmalıyım?
- a) Utility sınıf!
 - i) constructor private olur.
 - ii) İçerisine static alanlar konulur.
 - b) Enum yapısı da kullanılabilir.
- 3) Interface kontrat olarak kullanılır.

Item 23: Prefer class hierarchies to tagged classes (java class hierarchy)

- 1) Klasik Shape örneği. Elimde dikdörtgen ve yuvarlak var. Alan hesaplaması yapmam lazımdır. Aynı sınıfın içinde olması daha sonra eklenecek şekillerde problem çıkaracaktır. Ne yapılmalıdır?
- a) Bir abstract class yaz. İçinde abstract area method olsun.
 - b) Daire sınıfı abstract 'ı extend eder. Kendisine özel constructor ve özel area implementation var.
 - c) Aynısını her şekil için yapabilirim.
 - d) Mainde – Square s = new Square(2) ve area().

Item 24: Favor static member classes over nonstatic (Inner classes)

- 1) Static nested class
- a) Sınıfın içinde static sınıfı oluşturulur:

```
Java
public class StaticNested{
    public static class staticNestedClass{

    }
}
```

- b) Çapırma – StaticNested.staticNestedClass nesne = new StaticNestes.staticNestedClass(); nesne.yazBunu();
 - c) Örnek: Anasınıf - Hesap makinesi, Static sınıf: Operasyon.
- 2) NonStatic
- a) Tek fark inner classta static kelimesinin olmaması
 - b) Oluşturma:
 - i) NonStaticNested nonStatic = new Non...();
 - ii) NonStaticsNested.NonStaticNestedClass nesne2 = nonStatic.new NonStaticNestedClass(); – Bellekte sıkıntısı olabilir. Garbage collector bunu toplayamayabiliyor bazen. (Yani çok önerilmiyor)
 - iii) nesne2.yazBunu()
- 3) LocalClasses:

Java

```
publiğc class LocalClass(){
    public void yazBirşeyler(){
        class localClassInner{
            //...
        }
    }
}
```

- 4) Anonim Classlar
- a) Nornal bir sınıf oluştur.
 - b) İçerisine yazdır methodu implement ediyorum.
 - c) Oluşturma kısmında:
 - i) SuperClass sc = new SuperClass(){
 Public void yazdır(){
 System.out.println("Yeniden methodu tanımladım");
 }
 }

Item 25: Limit source files to a single top level class (2 sınıf bir dosyada tanımlamayın)

- 1) Adı üzerinde tek java sınıfında iki tane class olmasın. İsimler farklı olacak dosya adıyla. Başka bir yerde tekrar tanımlamaya çalışabilirim. Ama derleyici IDE hata verecektir.

Chapter 5: Generics

Item 26: Don't use raw types (Java Raw Types)

- 1) Parameterized type – List<String> – Hepsinin String yapısında olduğunu biliyorum.
- 2) Actual type parameter – String
- 3) Generic type – List<E>
- 4) Formal type parameter – E
- 5) Unbounded wildcard type – List<?>
- 6) Raw type – List
- 7) Bounded type parameter – <E extends Number>
- 8) Recursive type bound – <T extends Comparable <T>>
- 9) Bounded wildcard type – List<? Extgends Number>

Örnek kod:

```
Java
List<String> strings = new ArrayList();
unsafeAdd(strings, Integer.valueOf(42));
String s = strings.get(0);

unsafeAdd(List list, Object o){
    list.add(o); // -- Hata verir
}
```

Bu kod runtime'da exception alacaktır. Büyük kodlarda sıkıntı çıkarır. Alanını belirtirsek IDE uyaracaktır.

Item 27: Eliminate unchecked warnings - Supress Warnings

```
Java
public <T> T[] toArray(T[] a){
    if(a.length < size)
        return (T[]) Arrays.copyOf(elements, size, a.getClass());
    System.arraycopy(elements, 0, a, 0, size)
```

```
}
```

Warning: required: T[], found Object[]

Aslında güvenlik açığı yok. Ne yapılmalı?

Java

```
public <T> T[] toArray(T[] a){  
    if(a.length < size)  
        @SupressWarnings("unchecked") T[] returnValue = (T[])  
        Arrays.copyOf(elements, size, a.getClass());  
    return returnValue;  
  
    System.arraycopy(elements, 0, a, 0, size)  
}
```

Bunu yaptığımızda kesinlikle methodun üstüne neden güvenlik açığı olmadığı hakkında açıklama yazılır.

Item 29: Favor generic types (Java generics)

- 1) Daha önceden yazılmış bir Stack class'ımız var. Nasıl Generic hale getirebilirim?
 - a) Teknik 1: Stack<E>
 - i) Object[] → E[] – Tgm kod da buna uyarlanır.
 - b) Teknik 2: Stack<E> – same
 - i) Object[] – aynı kalıyor.
 - ii) Pop metodu değişir. Dönüşü E tipinde olmalıdır. Cast etmemiz gerekiyor.
 - iii) Push methodu da E alır.
- 2) Generic tipler daha esnek ve daha tip güvenli hale getirir kodu. Autoboxing Unboxing olaylarının önüne geçmeni sağlar.

Item 30: Favor generic methods (Generic Methods in Java)

- 1) Generic method nasıl yazılır
 - a) Örn: public static <E> Set<E> union(...){...}
- 2) Java içinde örnekler: Collections.sort(), Collections.emptySet();

- 3) Public <T extends Number> int topla() – Boolean toplaması olmaz. Neleri toplayabilirim? Sayıları toplayabilirim. Bu şekilde sağlayabilirim.
- 4) Blueprint – access modifier <conventionları> dönüş Tipi method_name(parameters...)

Item 31: Use bounded wildcards to increase API flexibility (Java Generics WildCards) – Zor Ders

- 1) Problem: List<Object> list = new ArrayList<String>() – Hata alır
- 2) Extends, super – üst tür, alt tür ilişkisi kurar.
- 3) Note: Producer extend eder, Consumer super kullanır (PECS)
- 4) Extends Örnek:
 - a) Class A
 - b) B extends A
 - c) C extends B
 - d) D extends B
 - e) List<A> list = new ArrayList – Hata alır
 - f) List<? Extends A > list = new ArrayList(); – Artık hata almaz.
- 5) Super Örnek:
 - a) List<? Super B> list = new ArrayList<C>(); – Hata alır
 - b) List<? Super B> list = new ArrayList(); – Artık hata almaz



Item 32: Combine Generics and varargs judiciously (Java Varargs) – Zor Ders

Varargs Nedir:

```
Java
static void sumTest(int ...v){
    System.out.println(v[0]);
    System.out.println(v[1]); //...
}
```

Çağırma:

```
sumTest();
sumTest(1);
sumTest(1,2);
```

- 1) Heap pollution: Bir liste var içinde hem string var hem integer var gibi.

Note: Generic tipler arrayize edilemez.

- 2) ClassCastException: Arrayize etmeye çalıştığımız zaman verebilir.
- 3) Generic ve varargs ile oluşabilecek sorunlar:
 - a) Casting:
 - i) toArray(T... args)
 - ii) toArray("Selin","Kırmacı"); → Casting hatası olur. Neden?
 - (1) T... args'a çevirmek istediği her şeyi Object sınıfı gibi davranışır çünkü hangi tipte geleceğini bilemez.
 - (2) Generic tiplerde casting hatası oluşabilir: List<Object> list = new ArrayList<String>() – Hata alır. Cast edemez.
 - (3) Yukarıdaki örnekteki gibi casting hatası alır.
 - 4) @SafeVarargs – Varargs kullanırken güvenlik açığı olmadığından eminsek kullanabiliriz.

Item 33: Consider typesafe heterogeneous containers (Java Typesafe Heterogeneous Map)

- 1) Typesafe: String bir variable'dan integer bekleyemem. Type safe olmalı

- 2) Heterojen: HashMap gibi yapıtlarda key'i belirttiğinden sonra key yerine başka bir type'ta key beklemem. Ama heterojende bu key yerine integer da koyabilirim, string de.
- 3) Nasıl yapabilir?

Java

```
public class Favorites implements FavoritesContract{  
    private Map<Class<?>, Object> favorites = new HashMap();  
  
    public <T> void putFavorite(Class<T> type, T instance){  
        favorites.put(Objects.requireNonNull(type), instance);  
    }  
  
    public <T> T getFavorite(Class<T> type){  
        return type.cast(favorites.get(type));  
    }  
  
    FavoritesContract f = new Favorite();  
    f.putFavortie(String.class, "Java");  
    f.putFavortie(Integer.class, 1234);  
    f.putFavortie(Class.class, Favorites.class);  
}
```

Chapter 6: Enums and Annotation

Item 34: Use enums instead of int constants (Java enums and Strategy Design Pattern)

- 1) Tek tek her elemanı tanımlayıp her birine değerini verebilir.
- 2) Enum olarak nasıl yapabilirim? – Debug ederken bile daha kolaylık sağlar
 - a) Public enum Apple {FUJI, PIPPIN, GRANNY_SMITH}
 - b) Eğer toString methodunu Override ediyorsak fromEnumToString ve fromStringToEnum methodları implement etmekte yarar var.
- 3) Enum ne zaman kullanılmalı?
 - a) Değişkenlerin sabit olduğunu bildiğim caseler.
 - b) Örn: Gezegenler, bizim projede mesajlar.

Java

```
public enum Planet{
    MERCURY(1,2);
    Planet(int mass, int gravity){
        ...
    }

    // has the getter setters too

    // can also implement necessary methods too
}
```

4) Strategy Design Pattern

Java

```
public enum PayDay {
    MONDAY(PayType.WEEKDAY), SUNDAY(PayType.WEEKEND);
    private final PayType type;
    PayDay(PayType type){
        this.type = type;
    }

    public int pay(){
        return 10 + type.calculateOverTime();
    }

    enum PayType {
        WEEKDAY{
            int calculateOverTime(){
                return 1;
            }
        },
        WEEKEND{
            int calculateOverTime(){
                return 4;
            }
        };
    }

    abstract int calculateOverTime();
}
```

```
}

public static void main(String[] args) {
    for(PayDay day: values()){
        System.out.println(day + ":" + day.pay());
    }
}
```

Item 35: Use instance fields instead of ordinals (Java Enum Ordinal)

- 1) ENUM.ordinal() → Oluşturulduğu index'i.
 - a) BLUE, RED; → Sırasıyla ordinal fonksiyonu 0 ve 1 verir.
- 2) Bunun yerine oluştururken index vermek daha mantıklı olabilir.
 - a) RED(2), BLUE(3),....
 - b) Constructor int alır.

Item 36: Use EnumSet instead of bit fields (Java EnumSet)

- 1) Bitwise operasyonlar
 - a) Binary: 0 veya 1 → Örn: 0001 = 1, 0010 = 2, 0011 = 3
 - b) Javada bu operasyonlar <<, >> (signed operasyonlar), <<<, >>> (unsigned operasyonlar)
- 2) Signed Örnek
 - a) Y = 4; y>>1
 - b) 4 = 0100 → 0010 = 2
 - c) Yani y>>1 = 2
- 3) Unsigned örnek
 - a) X = -1 → 1111.....11
 - b) x>>>31 → 0000...001 = 1

- 4) Enum'larda nasıl kullanılır:
- Mesela elimizde final var: private static final int STYLE_BOLD = 1 << 0;
 - Bunu yerine enum oluştur. Fonksiyona birden fazla enum'ı göndermek istiyorsam EnumSet.of(Style.BOLD, ...) vererek set olarak göndermiş oluruz.

Item 37: Use EnumMap instead of ordinal indexing (Java EnumMap) – Biraz zor ders

Note: Diziler ve genericler birlikte uyumlu değillerdir. Generic'ler compile timeda tip güvenliğini sağlarlar ama diziler runtime da belli olur.

- Problem : Bitkilerim var. Kimsi 1 lifecycle kimi 2 gibi gibi. Bu yapıyı nasıl kurabilirim?
- Kod
 - Lifecycle enumı var
 - Plant sınıfı var constructorda ismi ve lifecycle'ı alır.
 - Have an EnumMap:
 - Map<Plant.Lifecycle, Set<Plant>> plantBulifeCycle = new EnumMap<>(Plant.Lifecycle.class);
 - Neden EnumMap? Çünkü burada benim key value'm enum. EnumMap enumlar için tasarlanmıştır.
 - Çağır: Arrays.stream(garden).collect(groupingBy(p -> p*.lifecycle, () -> new EnumMap<>(Lifecycle.class), toSet())) → Bu kısmı ileride detaylı işleyeceğiz.
- Örnek kod 1:

Java

```
import java.util.*;

public class Plant {

    enum LifeCycle{
        ONE, TWO, THREE, FOUR;
    }

    String name;
    LifeCycle lifeCycle;

    Plant(String name, LifeCycle lifeCycle){
        this.name = name;
    }
}
```

```

        this.lifeCycle = lifeCycle;
    }

@Override
public String toString(){
    return name;
}

public static void main(String[] args) {
    Plant[] garden = {
        new Plant("FirstPlant", LifeCycle.ONE),
        new Plant("SecondPlant", LifeCycle.TWO),
        new Plant("ThirdPlant", LifeCycle.THREE),
        new Plant("ForthPlant", LifeCycle.THREE),
        new Plant("NewPlant", LifeCycle.FOUR),
    };

    Map<LifeCycle, Set<Plant>> plantsByLifecycle = new
EnumMap<>(Plant.LifeCycle.class);

    for(LifeCycle lc : LifeCycle.values()){
        plantsByLifecycle.put(lc, new HashSet<>());
    }

    for(Plant p: garden){
        plantsByLifecycle.get(p.lifeCycle).add(p);
    }

    System.out.println(plantsByLifecycle);
}
}

```

4) Örnek kod 2:

Java

```
import java.util.EnumMap;
```

```
import java.util.Map;

public enum Matter {
    SOLID, LIQUID, GAS;

    enum Transition{
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID), BOIL(LIQUID,
GAS), CONDENSE(GAS, LIQUID);
        private final Matter from;
        private final Matter to;

        Transition(Matter from, Matter to){
            this.from = from;
            this.to = to;
        }
    }
}

public static void main(String[] args) {
    Map<Matter, Map<Matter, Transition>> transitionsMap = new
EnumMap<>(Matter.class);

    for(Matter m : Matter.values()){
        transitionsMap.put(m, new EnumMap<>(Matter.class));
    }

    for(Transition t : Transition.values()){
        for(Matter m : Matter.values()){
            if(m.equals(t.from)){
                transitionsMap.get(t.from).put(t.to, t);
            }
        }
    }
}

System.out.println(transitionsMap);
}
```

Item 38: Emulate extensible enums with interfaces (Java Enum with Interfaces)

- 1) Enumlar kalıtılamaz.
 - a) Oluşturulduğuda default olarak final gelir.
 - b) Kendisi Java.lang'i extend ediyor. Bu yüzden başka sınıfı extend edemez bu yüzden.
- 2) Interface implement edebilir.
 - a) Con'lar
 - i) İki enum aybni şeyi implement ederse ikisinde de aynı method implement etmeli. Kod tekrarı oluyor.
 - b) Pro'lar
 - i) İki tane operation implement eden enum classı varsa, .values() methodunu kullanarak kodu çok rahatlıkla değiştirerek istediğim enum classının valuelarına göre işlem yapabilirim.

Item 39: Prefer annotations to naming patterns(Java Reflection / Yansıma) – Mülakat Soru Adayı

- 1) Reflection nedir?
 - a) Java sanal makinesinde (JVM), çalışan uygulamaların çalışma zamanındaki (runtime) davranışlarını inceleme ve bu davranışlara yön verme imkanı sağlayan bir özelliktir. – Hata ayıklamada kullanılır.
 - b) Field, getDeclaredFields, getDeclaredConstructor, getMethod() gibi fonksiyonları alma olayı. Projede de yaptık.
- 2) Annotation yazmak istiyorsak interface gibi yazmalıyım
 - a) Notasyon: Bir veri hakkında bilgi barındırır ve veri sağlar. Konfigürasyon amaçlı kullanılır.
 - b) Public @interface Test{}
 - c) @Retention, @Target – genelde olmak zorunda en başında
 - d) Retention Enumları:

```
/*
 * SOURCE: Notasyon derleyici tarafından atılır.
 * CLASS: Notasyon derleyici tarafından oluşturulan sınıf dosyasına kaydedilir ve JVM tarafından saklanması gerekmek. Varsayılan davranış bicismid
 * RUNTIME: Notasyon sınıf dosyasına derleyici tarafından kaydedilir ve çalışma zamanında JVM tarafından saklanır, böylece reflection ile okunabilir.
 */

```
 - e)
 - f) Eskiden Annotation yokken JUnit @Test yokmuş. Methodlar “test” keywordu ile başlaması gerekiyormuş. Naming pattern yerine annotation kullanmamız gerekiyor.

Java

```
@Retention(RetentionPolicy.RUNTIME) //
@Target(ElementType.METHOD) //bu notasyon sadece methodlar için
kullanılabileceğini gösterir
public @interface Test{
//Parametre almıyor
}

//Başka
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test{
    Class<? extends Throwable> value();
}

//Kullanımı
@Test(ArithmeticException.class)
public void method(){...}

//Başka
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test{
    Class<? extends Exception>[] value();
}
//Kullanımı
@Test(ArithmeticException.class, NullPointerException.class, ...)
public void method(){...}

//Başka
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(ExceptionTestContainer.class)
public @interface ExceptionTest{
    Class<? extends Exception> value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTestContainer{
    ExceptionTest[] value();
}

//Kullanım
@ExceptionTest(ArithmaticException.class)
```

```
@ExceptionTest(NullPointerException.class)
public void test(){...}
```

Item 40: Consistently use the Override annotation (Java notasyonları önemi - Nasıl olsa çalışıyor dersen patlarsın)

- 1) Override – method üzerine yazma işlemi.
- 2) Override yazmayı unutursak methodun üzerine yazmadığımız için göremeyiz. Örnk: toString, hashCode gibi. Sonra işin yoksa hatayı bulmaya çalış. O yüzden dikkat edelim.

Item 41: Java Marker Interface - Marker Annotation (Mülakat soru Adayı)

- 1) Marker annotation: İçine hiçbir şey almayan notasyondur. (İşaretleyici)
- 2) Marker interface:
 - a) Örn: Serializable
 - b) Genelde kontrol amaçlı.

Chapter 7: Lambdas and Streams

Item 42: Prefer Lambdas instead of anonymous classes (Java anonymous classes and lambda expressions)

- 1) Lambda nedir?
 - a) Kendi başına tanımlanabilen, parametre kabul edebilen ve döndürebilen bir fonksiyon.
 - b) Lambdadan önce anonymous classes vardı.
 - c) Anonymous classes: (Sadece interface için değil. Class da olabilirdi.)

```
Java
public interface SuperClass(){
    yazBeni();
```

```
}
```

```
SuperClass superclass = new SuperClass(){
    @Override
    yazBeni(){//doldur için }
}
```

```
superclass.yazBeni();
```

2) Neden lambda geldi

- a) Anonymous çok uzun
- b) Lambda kısa ve öz
- c) Lambda olarak kullanacaksam bir tane method içermesi lazım!!!
(@FunctionalInterface – içerisinde sadece bir method olduğunu söyler (default methodlar olabilir çünkü gövdesi vardır.))
- d) Örn:

Java

```
public interface SuperInterface (){
    yazBeni();
}

SuperInterface superinterface = () => {return "Hello"};
superinterface.yazBeni();
```

Java

```
public class SuperClass{
    String example(){...}
}
SuperClass sp = new SuperClass();

SuperInterface superInterface = sp::example; //Yukarıdakiyle aynı mantık.
Sadece başka bir classtan alıyor
superclass.yazBeni();
```

Item 43: Prefer method references to lambdas (Method reference Java 8 and Lambda)

- 1) Lambda: ()->{...}
- 2) Method reference: Integer.sum
- 3) Kısacası hangisi daha kısa ve pratik olacaksız onu seçmeliyiz.
- 4) Arz talep meselesine göre araştırılmalı.
- 5) Örn:

Method Ref Type	Example	Lambda Equivalent
Static	Integer.parseInt	str -> Integer.parseInt(str)
Bound	Instant.now()::isAfter	Instant then = Instant.now(); t -> then.isAfter(t)
Unbound	String.toLowerCase	str -> str.toLowerCase()
Class Constructor	TreeMap<K,V>::new	() -> new TreeMap<K,V>
Array Constructor	int[]::new	len -> new int[len]

Item 44: Favor the use of standard functional interfaces (Supplier interface, BinaryOperation interface (Elde varsa yenisini yazma))

- 1) Supplier:

```
Supplier<String> dahaKısa = () -> YazıAnaliziDependencyInjection.upuzunSayağıUzunBirMetodYazıyorumŞuAnda();
dahaKısa.get();
```

```
Supplier<LocalDate> s1 = LocalDate::now;
LocalDate s2 = LocalDate.now();
```

- 2) Herkes kafasına göre iş yapmasın. Belli işlemler belli bir standarta bağlı olsun. Bu yüzden çoktan yazılmış ve java tarafından verilmiş interfaceler varsa onları kullanalım.
- 3) Önemli interfaceler:

Interface	Function Signature	Example
UnaryOperator<T>	T apply(T t)	String::toLowerCase
BinaryOperator<T>	T apply(T t1, T t2)	BigInteger::add
Predicate<T>	boolean test(T t)	Collection::isEmpty
Function<T,R>	R apply(T t)	Arrays::asList
Supplier<T>	T get()	Instant::now
Consumer<T>	void accept(T t)	System.out::println

Item 45: Use streams judiciously (Java streams nedir? (Hangi koşullarda kullanmalıyım?))

1) Stream nedir?

- a) Java 8 ile gelmiştir.
- b) Sıralı, paralel toplu işlemleri kolaylaştırmak içindir.
- c) Örn:

```
List<String> myList = Arrays.asList("jose saramago", "stefan zweig", "tess gerritsen","selami");
myList
    .stream()
    .filter(s -> s.startsWith("s"))
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

- d) Intermediate operation: Ara işlem. Zincirlemeye yarar. Dönüşler birbiriyle uyumlu.
- e) Terminal: Void döner. Bitirmek içindir.

Operation	Return Type	Type Of Operation	What It Does?
filter()	Stream<T>	Intermediate	Returns a stream of elements which satisfy the given predicate.
map()	Stream<R>	Intermediate	Returns a stream consisting of results after applying given function to elements of the stream.
distinct()	Stream<T>	Intermediate	Returns a stream of unique elements.
sorted()	Stream<T>	Intermediate	Returns a stream consisting of elements sorted according to natural order.
limit()	Stream<T>	Intermediate	Returns a stream containing first n elements.
skip()	Stream<T>	Intermediate	Returns a stream after skipping first n elements.
forEach()	void	Terminal	Performs an action on all elements of a stream.
toArray()	Object[]	Terminal	Returns an array containing elements of a stream.
reduce()	type T	Terminal	Performs reduction operation on elements of a stream using initial value and binary operation.
collect()	Container of type R	Terminal	Returns mutable result container such as List or Set.
min()	Optional<T>	Terminal	Returns minimum element in a stream wrapped in an Optional object.
max()	Optional<T>	Terminal	Returns maximum element in a stream wrapped in an Optional object.
count()	long	Terminal	Returns the number of elements in a stream.
anyMatch()	boolean	Terminal	Returns true if any one element of a stream matches with given predicate.
allMatch()	boolean	Terminal	Returns true if all the elements of a stream matches with given predicate.
noneMatch()	boolean	Terminal	Returns true only if all the elements of a stream doesn't match with given predicate.
findFirst()	Optional<T>	Terminal	Returns first element of a stream wrapped in an Optional object.
findAny()	Optional<T>	Terminal	Randomly returns any one element in a stream.

f)

g) Kullanım caseler:

```
/*
  Bir dizi elemanın homojen olarak dönüşüme tabi tutulması
  Elemanların filtrelenmesi
  Tek bir işlemle bir dizi elemanın toplanması, minimum değer hesaplanması, ard arda eklenmesi vs.
  Bir dizi elemanın gruplanarak bir koleksiyona yazılması
  Bir dizi eleman içerisinde belki kriterlere göre arama yapılması|  [
*/
```

h) Streamler çok uzun olmaması okunaklılığı daha faydalı.

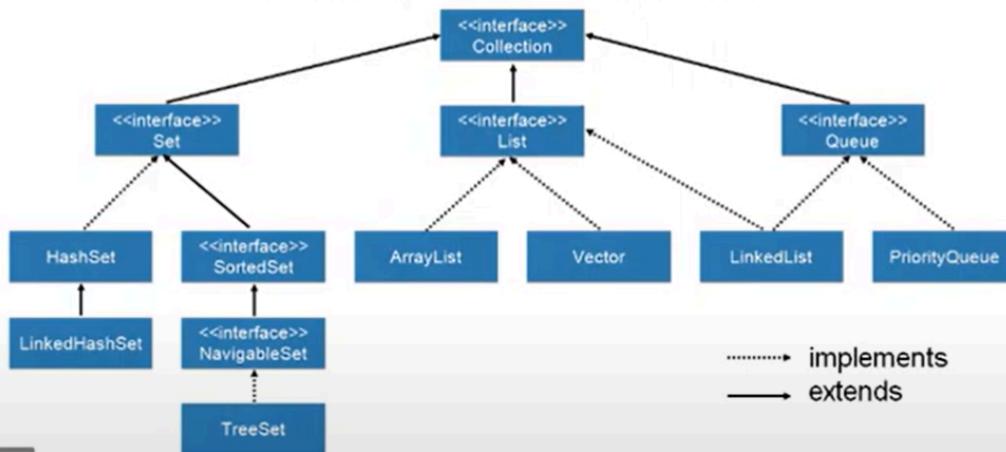
Item 46: Prefer side-effect-free functions in streams (Stream kullanırken dikkat edilmesi gerekenler)

- 1) stream 'de foreach önceki hesaplamları raporlamak için kullanmalıyım. Hesaplama yapmak için kullanılmaz. (for loop'undan çok bir farkı yok)
- 2) Case study: Kelime içeren bir listem var. Ben hangi kelimenin kaç kere kullanıldığını bulmak istiyorum.
 - a) Yanlış stream kullanımı: list.forEach(...) → içinde saymak.
 - b) Doğru stream kullanımı: list.collect(...) → içinde groupingBy kullanımı ile yapılır.
- 3) Önemli stream kullanımları:
 - a) toList: returns a list that cannot be added to or removed from.
 - b) toSet:
 - c) toMap: creates a collector that accumulates elements from a stream into a Map.
 - d) groupingBy: takes a function as an argument and applies it to each element in the stream.
 - e) Joining: concatenate the elements of the stream into a single String

Item 47: Prefer Collections to Stream as a return type (Java stream vs Collection (return type)) (Senior mülakat sorusu)

Collections:

Collection Interface



- 1) Eager vs Lazy
 - a) Lazy: `getInstance` method çağrılmadan objeyi oluşturmuyodu.
 - b) Eager: Çapırılsa da çağrılmasa da obje olarak diskte yer tutuyor.
- 2) Case: Eğer binlerce satır okuma varsa, yüklü bir dönüş yapılacaksa
 - a) Stream tercih edilemeli. (Lazy). Diski yormaz
 - b) Fakat az satır ise Collection kullanılır. (Eager olduğu için)
- 3) Case: Esneklik ve fonksiyonel yapı lazımsa
 - a) Stream kullanılmalı.
 - b) Ama streamler iki kere üst üste okunamaz.
- 4) Case: Modifiye durumlar varsa
 - a) Collection kullanılmalı.
 - b) Stream liste döndürünce add remove yapamam.

Item 48: Use caution when making streams parallel (Java parallel streams (Kullanmadan önce 2 kere düşünün))

- 1) `list.parallelStream()` veya `stream.parallel`
- 2) Good case: Bankada müşteri numaralarım var. Ben bir müşteri için bir işlem yapmak istiyorsam numaraların bulunduğu listede paralel ilerleyebilir.
 - a) `list.parallelStream().forEach(...)`

- 3) Bad case: MersennePrimes örneğindeki gibi çok büyük işlemlerin olduğu durumlar varsa işlemciyi çok yorar ve kod çalışma ortamını çok kötü etkilemiş olur.
- 4) Genel olarak not. Nadir kullanılan bir özellikle. İlla kullanılabilsa unit testlerin çok iyi yazılması lazım. Productiona çıktığında problem çıkma ihtimali yüksek.

Chapter 8: Methods

Item 49: Check parameters for validity (Java tips for methods (1 dakikanız 1 saatiniz kurtarabilir.))

- 1) Methodun aldığı parametrelerin kontrol edilmesi gerekiyorsa, methodun en başında kontrol edip gerekli exceptionlar atılmalı.
- 2) Null check: Object.requireNonNull(object)

Item 50: Make defensive copies when needed (Java defensive copy (Mutable and Immutable in java))

- 1) Immutable: Üzerinde işlemler yapılmasına rağmen değişmemeye durumu. (final sınıflar mesela String)
 - a) isim.toUpperCase() – büyümüş sonuç verir ama direk isim objesini değiştirmez
- 2) Mutable: Üzerinde işlemler yapınca değişme durumu.
 - a) Date objesinde setYear gibi işlem yapınca objenin kendisinde değiştirir.
- 3) Normal kod:

Java

```
public final class Period{  
    public Period(Date start, Date end){  
        this.start = start;  
    }  
}
```

//Bu durumda start date'i alındığında ve setYear denildiğinde obje değişmiş olur ama ben Period class'ımı immutable olarak yazdım.

4)

```
Java
public final class Period{
    public Period(Date start, Date end){
        this.start = new Date(start.getTime());
    }
}

//Bu durumda start date'i alındığında ve setYear denildiğinde obje
değişmez.
```

Item 51: Design method signatures carefully (Java method signatures (Metodların altın kuralları))

- 1) Adını okuduğumuzda ne yaptığını anlamamız gerekiyor.

get	Gets something from an object, such as <code>bean.getBar()</code> . This is also used with a key to lookup a list by index or a map by key, such as <code>list.get(index)</code> or <code>map.get(key)</code> .
is	Checks if something is true or false about an object or a property of an object. Example <code>foo.isValid()</code> .
check	Checks if something is true, throwing an exception if it is not true. Example <code>foo.checkValid()</code> .
contains	Checks if a collection contains the queried object, such as <code>coll.contains(bar)</code> . This can be used on classes that wrap, or otherwise act as, a collection.
remove	Removes an element from a collection, such as <code>coll.remove(bar)</code> . This can be used on classes that wrap, or otherwise act as, a collection.
clear	Clears the object, typically but not necessarily a collection, so that it is "empty".
put	Mutable putter. This mutates the target object replacing or adding some form of key-value pair. Examples are <code>map.put(key,bar)</code> and <code>bean.putFoo(key,bar)</code>
set	Mutable setter. This mutates the target object setting a property, such as <code>bean.getBar(bar)</code> .
with	Immutable "setter". Returns a copy of the original with one or more properties changed, such as <code>result = original.withBar(bar)</code> .
to	Converts this object to an independent object, generally of another type. This generally takes no arguments, but might if it is appropriate.
as	Converts this object to another object where changes to the original are exposed in the result, such as <code>Arrays.asList()</code> .
build	Builds another object based on either the specified arguments, the state of the target object, or both.
add/subtract	Adds/subtracts a value to the quantity. This mutates the target quantity (a number, date, time, distance...) adding/subtracting the "foo" property. This name is also separately used for adding elements to a collection.
plus/minus	Immutable version of add/subtract for a quantity. Returns a copy of the original with the value added/subtracted. This name does not seem to work as well for adding elements to an immutable collection.
append	Sometimes used to by methods that add to the end of a list, such as in <code>StringBuilder</code> .
reset	Resets the object back to a suitable initial state ready to be re-used.
past tense	Used on immutable classes to helpfully suggest to the caller that the method doesn't mutate the target, but must instead be assigned to another variable. Returns a copy of the target object with the method name applied. Examples are <code>immutable.normalized()</code> , <code>immutable.multipliedBy(bar)</code> , <code>immutable.dividedBy(bar)</code> and <code>immutable.negated()</code>

And here are some static method names:

of	Static factory method. Typically used with immutable classes where constructors are private (permitting caching). This is used by <code>EnumSet</code> and <code>ThreeTen/JSR-310</code> .
valueOf	Longer form of <code>of</code> used by the JDK.
from	Longer form of <code>of</code> . JSR-310 uses 'from' when performing a "loose" conversion between types, ie. one that has a reasonable chance of failure. By contrast ' <code>of</code> ' is used when the conversion is almost certain to succeed.
parse	Static factory method that creates an instance of the class by parsing a string. This could just be

- 2) Solid prensibinde "S"ye uy. → single responsibility. Sadece tek bir iş yapısın.
- 3) En fazla 4 parametre. Daha fazlaysa methodlara böl.
- 4) Boolean yerine 2 elemanlı enumlar kullanılabilir.

Item 52: Use overloading judiciously (Java method overriding and overloading (Yeni mezun mülakat sorusu))

- 1) Overloading: Aynı isimli metodların farklı parametre alması.

```
public class CollectionClassifier {

    public static String classify(Set<?> s) {
        return "Set";
    }

    public static String classify(List<?> lst) {
        return "List";
    }

    public static String classify(Collection<?> c) {
        return "Unknown Collection";
    }

    public static void main(String[] args) {
        Collection<?>[] collections =
        {
            new HashSet<String>(),
            new ArrayList<BigInteger>(),
            new HashMap<String, String>().values()
        };

        for (Collection<?> c : collections)
            System.out.println(classify(c));
    }
}
```

- a) 3'ü de unknown collection yazdırır.
b) Hangi metodun seçildiği runtime'da değil compile time'da karar verilir.
2) Farklı isimlerle çözüm bulunabilir.
a) Constructor yazarken böyle bir opsiyonumuz yok.

Item 53: Use varargs judiciously (Java varargs yanlış ve doğru kullanım örnekleri)

1)

```
// The WRONG way to use varargs to pass one or more arguments! (Page 245)
// static int min(int... args) {
//     if (args.length == 0)
//         throw new IllegalArgumentException("Too few arguments");
//     int min = args[0];
//     for (int i = 1; i < args.length; i++)
//         if (args[i] < min)
//             min = args[i];
//     return min;
// }

// The right way to use varargs to pass one or more arguments (Page 246)
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}

public static void main(String[] args) {
    System.out.println(sum(1, 2, 3, 4, 5, 6, 7));
    System.out.println(min());
}
```

- 2) Sebep: Runtime'da exception atmak yerine compile time'da istenilen parametrelerin kesin alındığından emin olmak için.

Item 54: Return empty collections or arrays, not nulls (Java do not return null (Return empty collections or arrays))

- 1) Null döndüğümüz noktalarda kullandığımız yerde null check yapmak gerekiyor. Null dönmek mantıklı ise yine dönülebilir ama başka bir çözüm varsa onu dönelim.
- 2) Null dönmek iyi bir API tasarımlı olmaz.
- 3) Collections.emptyList()
- 4) New Cheese[0]
- 5) Null döndüğümüzde performansa da katkısı yoktur.

Item 55: Return optionals judiciously (Java optional kullanımı)

- 1) Optional nedir?
 - a) Amacı null pointer exceptionları en aza indirmek.
 - b) Null kontrolü yapılmasına gerek kalmaz.
 - c) Yazılım ve okunabilirlik artar.
- 2) Exception fırlatmak pahalı bir şeydir. Traceability de gider.
- 3) Optional.isEmpty(), Optional.of, Optional.ofNullable(), ... gibi metodları var.

Item 56: Write doc comments for all the exposed API elements (Java doc - Java comments)

- 1) Dokümantasyon kodun ne yaptığı, parametrelerin ne olduğu, ne exceptionları fırlatır onları anlatır.

Chapter 9: General Programming

Item 57: Minimize the scope of local variables (Java local variables (Scope'u küçük tutmanın avantajları))

- 1) Yerel değişken: İki süslü parametrenin içindeki değişken.
- 2) Örn: for loop
- 3) Yerel değişkeni olabildiğince dar bir alanda kullanmalıyız.

Item 58: Prefer foreach loops to traditional for loops (Java for loop vs foreach loop)

- 1) Index takibi yapmıyorsak veya listede modifiye yapmıyorsak foreach kullanalım.

Item 59: Know and use the libraries (Java kütüphanelerini bilin ve kullanın (Tekerleği yeniden icat etme))

- 1) Java.lang, java.util ve java.io paketlerini bilmeliyiz.

Item 60: Avoid float and double if exact answers are required (Java use BigDecimal for Monetary calculations (parasal işlemlerde double ve float kullanma))

- 1) Örnk: double iki sayı 1.03- 0.42 sonucunu 0.61 beklerken bana 0.6100...01 sonucunu veriyor. → Floating point
- 2) Big Decimal bu sorunu çözer. Ama integer'a göre daha yavaştır.

Item 61: Prefer primitive types to boxed primitives (Primitive types vs boxed primitives)

- 1) Primitive types: int, double, boolean...
- 2) Boxed primitives: Integer, Double, Boolean...
- 3) Karşılaştırmalarda direkt Integerlarda == kullanamam çünkü o artık bir nesnedir. İçindeki değere bakmak yerine nesne olarak bakar. Unboxing yapıp o şekilde karşılaştırma yapmak gereklidir.
- 4) Boxed primitives null alabilir. Bu yüzden bir şeye eşitlenmeden işlem yapılmasına compile time'da hata vermez. Bu da riskli olabilir çünkü null değerdir. Primitive tipler null almaz. Bu yüzden initialize edilmeden önce kullanılmaya çalışılırsa compile timeda uyarı alırız.
- 5) Primitive tipler işlemlerde daha hızlıdır.

Item 62: Avoid strings where other type are more appropriate (Java her zaman string kullanmak zorunda değilsiniz)

- 1) Örn: Json gelen mesajı String alırsak parse etmesi çok kod kalabalıklığı oluşturacak. Onun yerine aldığımız yerde model oluşturup maplememiz bizim işimizi çok kolaylaştıracaktır.
- 2) Örn2: String ile if case'lerinde eğer camel case problemi varsa tekrardan kod kalabalık olacaktır. O noktada enum kullanmak daha mantıklı olur.

Item 63: Beware the performance of string concatenation

- 1) Cocatenate işlemi iki string arasında olunca ikisini birleştirmek yerine ikisinin de kopyasını oluşturur çünkü string immutable'dır. Üzerinde değişiklik yapamam.
 - a) String result = ""
 - b) For loop içinde result += "a"; → her seferinde kopyalam ve yer ayırma process'i var.
- 2) Yerine StringBuilder kullanmalıyız.
 - a) StringBuilder b = new...
 - b) b.append("a") → b bir nesne. Onun üzerinde değişiklik yapabilirim. Tekrar tekrar kopyalama işlemine gerek kalmaz.

Item 64: Refer to objects by their interfaces (Java how to create set and list (Obje yaratmadan interface kontrol et))

- 1) LinkedHasSet lhs = new LinkedHashSet();
 - a) Bu tanımla direkt objenin kendisini kullanıyor.
 - b) İleride bunu hashSet'e çevirmek istesem hata verecek.
 - c) Örn: lhs = new HashSet() → hata verir.
- 2) Ne yapmalıyız?
 - a) Set set = new LinkedHashSet();
 - b) Bu tanım interface kullanarak oluşturmuş. Bu sayede ileride değişiklik yapabilirim.
 - c) Set = new HashSet(); → hata vermez.

Item 65-68: Prefer interfaces to reflection (Java Naming Convention - Native Methods - Optimization)

- 1) Item65: Reflection: Objenin instancen direkt oluşturmadan alanlarına methodlarına ulaşabiliyoruz.
 - a) Okuması zor.
 - b) Compile time type checking yapamayız.
 - c) Performans suffers.
- 2) Item66: Native metodlar:
 - a) Java platform bağımsız. Her yerde çalışır.
 - b) Bu yüzden platform bağlı işlem yapmak zorlaşabilir.
 - c) Java native interface (JNI) – c,c++ kodunu java entegre edebiliriz.
- 3) Item67: Hızlı program yerine iyi program yazın. Okunabilir kod, güzel patternli kod daha tercih edilmeli.
- 4) Item68: Naming Conventions:

- a) Oracle sitesinde var.

Chapter 10: Exceptions

Item 69: Use exceptions only for exceptional conditions (Java exceptions nedir - throw, throws, checked, unchecked)

- 1) Exception = istisna
- 2) Exception aldığımda sistemimin down olmasını istemediğim için handle edilmesi gerek.
- 3) Throwable tüm exceptionların atasıdır.
- 4) Try catch'lerle exception handling yapılabılır.
- 5) Catchler yazılrken özelden genele doğru gitmeliyiz.
- 6) Hata türleri
 - a) Uncheck – runtimeda alınabilecek hatalar
 - i) Method throws StackOverflowException
 - ii) Handle edemeyeceğimiz durumlarda atılabilir.
 - iii) Örn: Kullanıcı dosya adı yerine null bir şey gönderdi. Bunu beklemiyorum. Unchecked atılabilir.
 - b) Checked – compiler tarafından check edilen hatalar (örn: IOException)
 - i) method() throws IOException gibi
 - ii) Handle edebileceğim durumlarda kullanılır.
 - iii) Örn: Kullanıcı yanlış dosya ismi girdi. Bunu handle edebilirim. Checked exception kullanılır.

```
public static void checkAge(int age){  
    if(age<18)  
        throw new ArithmeticException("Not Eligible for voting");  
    else  
        System.out.println("Eligible for voting");  
}
```

- 7)
- 8) Doğru bir kullanım değildir. Fonksiyon yaş kontroll etmek için yazılmıştır ve yaşı 18den küçük gelmesi de akışın bir parçasıdır. Bu durumda exception fırlatmak mantıklı değil. Exceptional bir durum değildir.

Item 72: Favor the use of standard exceptions (Java exceptions detaylı anlatım – Nasıl yazarım? Bilinmesi gereken exceptionlar)

```
// Bilinmesi gereken exceptionlar  
/*  
 *  IllegalArgumentException  
 *  IllegalStateException  
 *  NullPointerException  
 *  ArrayIndexOutOfBoundsException  
 *  ConcurrentModificationException  
 *  UnsupportedOperationException  
 */
```

- 1)
- 2) `IllegalArgumentException` – Beklenenden farklı bir argüman girilmesi sonucu
- 3) `IllegalStateException` – dosyayı açmadan okumaya çalışmak gibi
- 4) `NullPointerException` – null ile işlem yapılmaya çalışıldı
 - a) Önemli nokta – ali = null
 - b) ali.equals("g4wegw") – exception fırlatır.
 - c) "ewgw".equals(ali) – exception fırlatmaz.
- 5) `ArrayIndexOutOfBoundsException` – array var. Array length dışında ulaşmaya çalışırsak hata alır.
- 6) `ConcurrentModificationException` – Örn:
 - a) `ArrayList` – add 1,2,3
 - b) For each integer – remove 2
 - c) Concurrent exception fırlatır.
 - d) Nedeni: Tüm listenin elemanlarını dönmeye çalışırken (foreach ile) içinden bir eleman eklendiğin için listenin yapısı bozulmuş oluyor.
- 7) `UnsupportedOperationException` – desteklenmeyen işlem yapılmaya çalışıldığında
 - a) Örn: `list = ArrayList(array)`
 - b) `list.add("")` – patlatır.

Chapter 11: Concurrency (es zamanlı)

Item 78: Synchronized access to shared mutable data (Java concurrency – threads, synchronized, atomicboolean, volatile)

- 1) `Synchronized` – ensures that only one single thread can execute method or code block at one time
- 2) `Volatile` – uçucu demek

```

public class StopThread3 {

    private static volatile boolean stopRequested;

    public static void main(String[] args) throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
                i++;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}

```

3)

```

public class StopThread4 {

    private static AtomicBoolean stopRequested = new AtomicBoolean(false);

    public static void main(String[] args) throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested.get())
                i++;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested.set(true);
    }
}

```

4)

```

//Properly synchronized cooperative thread termination
public class StopThread2 {

    private static boolean stopRequested;

    private static synchronized void requestStop() {
        stopRequested = true;
    }

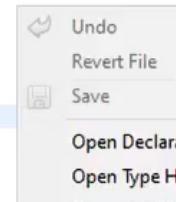
    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested())
                i++;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}

```

5)



Item 79: Avoid excessive synchronization (Java deadlock – Synchronized ipuçları)

- 1) Deadlock nedir? – İki ya da daha fazla işlemin devam etmesi için birbirinin bitmesini beklemesi ve sonuçta ikisinin de devam edememesidir.

Item 80: Prefer executors, tasks, and streams to threads (JavaExecuterService - Executor Framework)

- 1) Executor Framework:
 - a) Single thread executor
 - i) Executor.newSingleThreadExecutor()
 - ii) Tek thread'le iş hallediyor. Araya yeni bir thread sokmuyor.
 - b) Fixed thread executor
 - i) Executor.newFixedThreadPool(3) – fix sayıda thread alır.
 - ii) Note: Executor.newFixedThreadPool(1) eşit değildir
newSingleThreadExecutor
 - c) Cached Thread Pool
 - i) Executor.newCachedThreadPool()
 - ii) Fixed sayıda thread üretmez. Gelen iş kadar üretmeye başlar. Eğer threadlerden biri boştaysa gelen işi ona verir. Değilse yeni thread üretir. Threadler ön bellekte(cache) tutulur. 1 dakikadan boyunca kullanılmayan threadler silinir.
 - d) Scheduled Executor
 - i) executor = Executor.newScheduledThreadPool(3)
 - ii) executor.scheduleWithFixedRate(...)
 - iii) executor.scheduleAtFixedDelay(...)

Try adding a `Thread.sleep(1000);` call within your `run()` method... Basically it's the difference between scheduling something based on when the previous execution *ends* and when it (logically) *starts*.

For example, suppose I schedule an alarm to go off with a fixed *rate* of once an hour, and every time it goes off, I have a cup of coffee, which takes 10 minutes. Suppose that starts at midnight, I'd have:

```
00:00: Start making coffee
00:10: Finish making coffee
01:00: Start making coffee
01:10: Finish making coffee
02:00: Start making coffee
02:10: Finish making coffee
```

If I schedule with a fixed delay of one hour, I'd have:

```
00:00: Start making coffee
00:10: Finish making coffee
01:10: Start making coffee
01:20: Finish making coffee
02:20: Start making coffee
02:30: Finish making coffee
```

Which one you want depends on your task.

Item 81: Prefer concurrency utilities to wait and notify (Java wait and notify, ConcurrentHashMap, BlockingQueue, CountdownLatch, Semaphore)

- 1) Wait and notify
 - a) Wait threadi kitler notify veya notifyAll gelene kadar
 - i) Notify: Aynı obje içerisindeki wait çağrılmış tek bir threadi notify eder.
 - ii) NotifyAll: Aynı obje içerisindeki tüm wait etmiş thread'leri notify eder.
- 2) Concurrency utilities
 - a) Concurrent collections:
 - i) ConcurrentHashMap:
 - (1) Hashmapin aksine birden fazla thread'in aynı anda map'e erişip en az bir tanesinin update işlemi yapabildiği ek bir senkronizasyon yapısı gerektirmeyen veri yapısıdır.
 - (2) Tüm metodları thread safetir.
 - (3) Hashmaple kıyasla daha yavaş olabilir. Synchronized yapısından dolayı.
 - ii) BlockingQueue
 - (1) Consumer producer örneği
 - b) Synchronizers
 - i) CountdownLatch
 - (1) latch.countDown() – bir azaltır.
 - (2) Latch.await – latchcountdown 0 olana kadar bekler. 0 olunca devam eder.
 - ii) Semaphore
 - (1) We can use semaphores to limit the number of concurrent threads accessing a specific resource.

Item 82: Document thread safety (Java StringBuffer vs StringBuilder - Document Thread - Thread tips)

- 1) StringBuffer - synchronized - veri tutarlılığı için
- 2) StringBuilder - not synchronized - hız için

`StringBuilder` is faster than `StringBuffer` because it's not `synchronized`.

Here's a simple benchmark test:

```
public class Main {  
    public static void main(String[] args) {  
        int N = 77777777;  
        long t;  
  
        {  
            StringBuffer sb = new StringBuffer();  
            t = System.currentTimeMillis();  
            for (int i = N; i-- > 0;) {  
                sb.append("");  
            }  
            System.out.println(System.currentTimeMillis() - t);  
        }  
  
        {  
            StringBuilder sb = new StringBuilder();  
            t = System.currentTimeMillis();  
            for (int i = N; i > 0; i--) {  
                sb.append("");  
            }  
            System.out.println(System.currentTimeMillis() - t);  
        }  
    }  
}
```

A [test run](#) gives the numbers of 2241 ms for `StringBuffer` VS 753 ms for `StringBuilder`.

3)

```
// 84  
/*  
 * The best way to write a robust and responsive program is to ensure that the average number of runnable threads is not  
 * significantly greater than the number of processors.  
 */
```

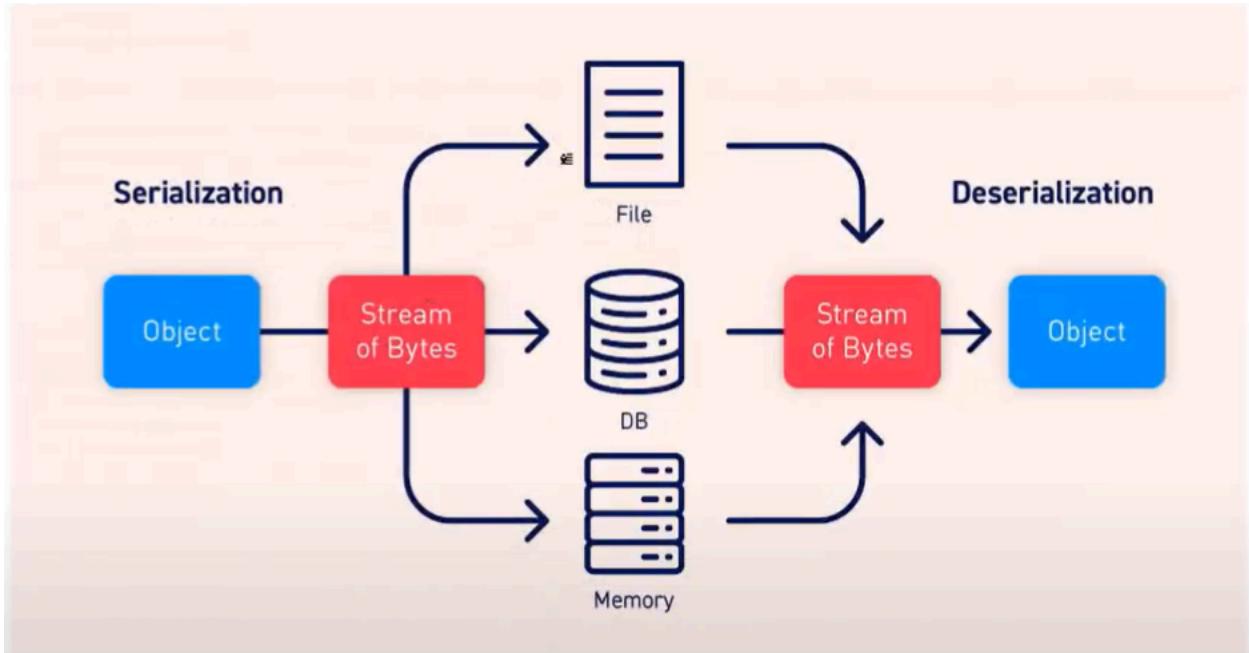
4)

Chapter 12: Serialization

Item 85 - 87: Prefer alternatives to java serialization (Java serialization and deserialization - json vs serialization)

- 1) Yeni bir sistem kurduğumuzda aslında serialization kullanmamız için bir neden yok.
- 2) **Serialization:** Objeleri `byteStream`'e çevirdiğimiz işlemidir. Bu şekilde kayıt yapabiliyorum.
- 3) **Deserialization:** Tam tersi işlemidir.
- 4) Bu işlemleri yapabilmek için `Serializable` implement etmeli.

- a) serialVersionUID
- i) Eklenirse (örn 1) ile yazılıp, okunurken değişmişse (örn: 2) hata verir.
 - ii) Yazmazsa kriptografik karma işlemiyle kendisi otomatik üretiyor. – Maliyetli



- 5)
- 6) Bunları kullanmayalım deniyor
 - a) Maliyetli
 - b) Güvenlik açığına neden olabilir.i
- 7) Ne kullanmalıyım?
 - a) Json kullanabiliriz.
 - i) Popüler
 - ii) Başka kütüphaneler kullanmak gerekiyor. (jackson)

Item 87: Considering using a custom serialized form (Java custom serialization nasıl yapabilirim? - Örnek transient kullanımı)

- 1) Transient: geçici
- 2) Transient serialize edildiğin dosyaya yazım ve okudum. Bu değer null gelir. – Uçucu
 - a) Static transient String a = "wrvwr"; veya static final transient... – Uçucu özelliğini kaybeder.

```

private void writeObject(ObjectOutputStream oos) throws IOException {
    String userName = name;
    String encryptedPassword = password + " salt";
    oos.writeObject(userName);
    oos.writeObject(encryptedPassword);
}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    // Decrypt ederek aldığıni düşün..
    name = (String) in.readObject();
    password = (String) in.readObject();
    // password = password.replace(" salt", "");
}

```

3)

Item 88-89 : Write readObject methods defensively (Java serialization - readObject, Instance control, readResolve, Enum types)

- 1) Read object methodunda (serialization deseriazliation için) defensive copy yapmak daha sağlıklı bir çözüm olur. (Item 50) Obje olarak verilen şeyleri yeni bir obje oluşturarak kullan.

To summarize, use enum types to enforce instance control invariants wherever possible. If this is not possible and you need a class to be both serializable and instance-controlled, you must provide a `readResolve` method and ensure that all of the class's instance fields are either primitive or transient.

- 2)

Item 90 : Consider serialization proxies instead of serialized instances (Serialization Proxy nedir?)

```
+ 5 public class SerializationProxy implements Serializable {
6
7     private String name;
8     private int age;
9
10    public SerializationProxy(TruePerson tp) {
11        this.name = tp.getName();
12        this.age = tp.getAge();
13    }
14
15    public String getName() {
16        return name;
17    }
18    public void setName(String name) {
19        this.name = name;
20    }
21    public int getAge() {
22        return age;
23    }
24    public void setAge(int age) {
25        this.age = age;
26    }
27
28    public Object readResolve(){
29        return new TruePerson(name, age);
30    }
31
32 }
```

1)

```
public class TruePerson extends WrongPerson implements Serializable {
    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public TruePerson(String name, int age) {
        // TODO Auto-generated constructor stub
        super(name, age);
    }

    private Object writeReplace() {
        return new SerializationProxy(this);
    }

    private void readObject(ObjectInputStream stream) throws InvalidObjectException {
        throw new InvalidObjectException("Use Serialization Proxy instead.");
    }

    @Override
    public String toString() {
        return "TruePerson [writeReplace()=" + writeReplace() + ", getName()=" + getName() + ", getAge()=" + getAge()
               + ", getClass()=" + getClass() + ", hashCode()=" + hashCode() + ", toString()=" + super.toString()
               + "]";
    }
}
```

2)