# Java Mülakat Soruları

# Compiler(Derleyici) nedir?

Yüksek seviyeli bir dili makinanın anlayabileceği şekle dönüştürmedir. Çıktı olarak üretilen makine kodu sonradan herhangi bir zamanda farklı girdilerle tekrar tekrar çalıştırılabilir.

Örnek olarak; Java derleyicisi javac verilebilir. Javac, .java uzantılı kaynak dosyasını Java Sanal Makinesi (Java Virtual Machine) olarak bilinen bir hayali makine için makine dili olan Java bytecode ile yazılmış .class dosyasına dönüştürür.

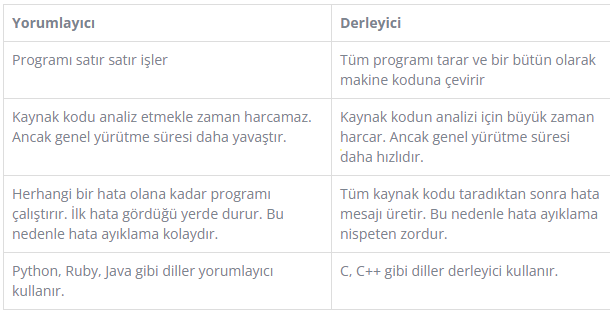
# Interpreter (Yorumlayıcı) nedir?

**Yorumlayıcı (Interpreter),** girdi olarak program için olan verilerle birlikte kaynak kodu alan, ve kaynak programı satır satır yürüten bir programdır.

Java’da derleyici ve yorumlayıcı beraber çalışır. Yani, önce oluşturulan kaynak koddan bir ara kod (bytecode) üretilmek için derlenir. Daha sonra bu derlenen bytecode Java Sanal Makinesi (JVM) üzerinde yorumlanarak yürütülür. Bu bazı avantajları da beraberinde getirir. En önemlisi platform bağımsızlığıdır. JVM çalışan her makinede bytecode’larımız sorunsuz çalışacaktır. İkinci avantajı ise Java bytecode’umuz bir sanal makine üzerinde çalıştığı için kötü amaçlı programlara karşı koruma sağlayan bir güvenlik katmanı ile korunmuş oluruz.

Java bytecode ve java yorumlayıcısı sadece Java dilinde kullanılmaz. Örnek olarak, Jython’u Python dilinden Java bytecode’una derlemek için kullanabiliriz ve daha sonra da yorumlamak için java‘yı kullanabiliriz. Ayrıca ML, Lisp ve Fortran derleyicileri de java bytecode’una derlerler.

# Compiler(Derleyici) ile Interpreter (Yorumlayıcı) farkları?



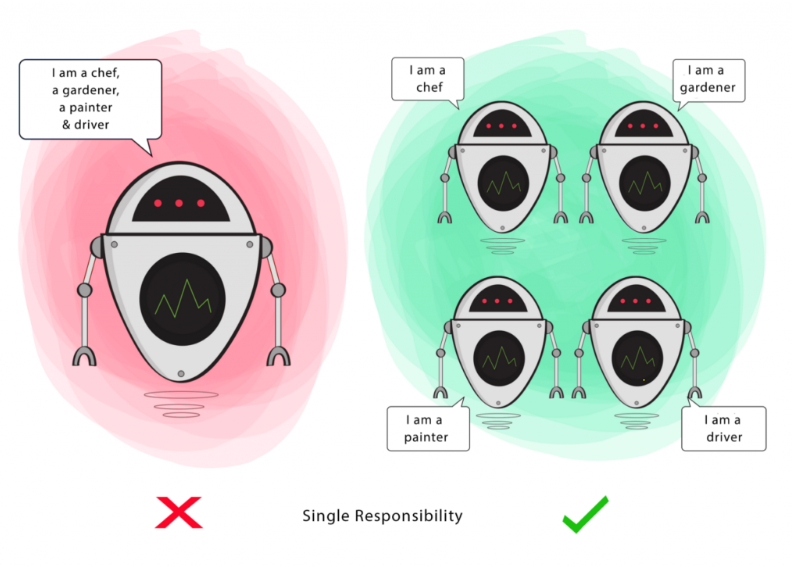
# SOLİD Prensibi

**SOLID prensipleri**, bir yazılım geliştiricinin Nesne Yönelimli Programlama (OOP) ile yazılım geliştirirken, geliştirdiği yazılımın esnek ve geliştirilmeye uygun olması için uyması gereken kurallar bütünüdür. **“Clean Code”** yazmamızı da yine bu prensipler sağlamaktadır.

Dünya standartlarında yazılım geliştirmemize olanak sağlayan bu prensipleri 5 ana başlıkta ele alabiliriz.  
1. **S**-Single Responsibility Principle  
2. **O**-Open/Closed Principle  
3. **L**-Liskov Substitution Principle  
4. **I**-Interface Segregation Principle  
5. **D**-Dependency Inversion Principle

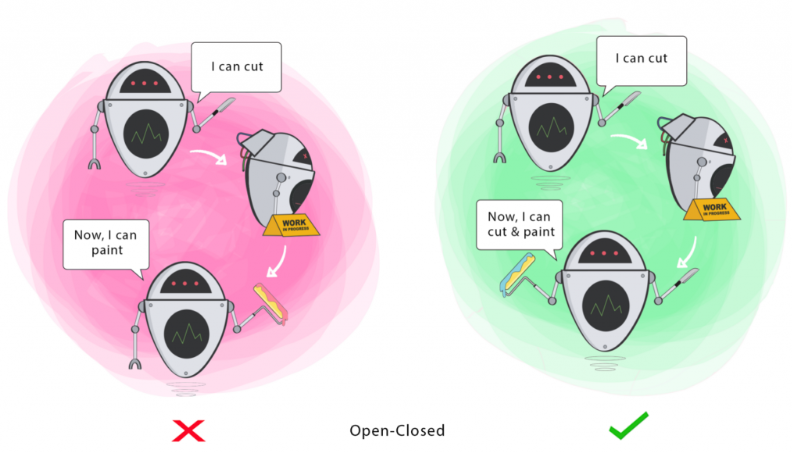
## Single Responsibility Principle

Türkçe karşılığı “Tek Sorumluluk” anlamına gelen bu prensipte amaç; geliştirilen projede bir güncelleme veya değişiklik yapılması istendiğinde kodların içinde kaybolmadan, yalnızca ilgili metoda giderek istenilen değişikliğin yapılmasının sağlanmasıdır. Biraz daha açacak olursak; bir fonksiyona birden fazla iş verip onu birçok işten sorumlu tutmak yerine, her bir iş için ayrı bir metot oluşturmalı ve ilerleyen zamanlarda bir değişiklik yapılacağında da kolaylıkla ilgili metoda giderek gerekli değişiklikleri yapabilmeliyiz.



## 2- Open/Closed Principle

Türkçe çevirisi “Açık/Kapalı” olan prensip, projede geliştirilen **nesnelerin geliştirilmeye açık ama değişime kapalı** olmaları gerektiğini ifade eder. Yani bir nesne davranışını değiştirmeden yeni özellikler kazabiliyor olmalıdır. Bu prensip, sürdürülebilir ve tekrar kullanılabilir yapıda kod yazmanın temelini oluşturur.



## Liskov Substitution Principle

“Yerine Geçme” olarak Türkçeye çevirdiğimiz prensibe göre; **miras alarak türemiş olan class’ların önce miras aldıkları nesnenin tüm özelliklerini kullanması, daha sonra da kendi özelliklerini barındırması** gerekir. Eğer oluşturduğumuz class, miras aldığı nesnenin ‘tüm’ özelliklerini kullanmayacaksa ortaya gereksiz kod blokları çıkar ve bu da bir geliştiricinin isteyeceği en son şeydir. Çünkü bir geliştirici her daim ‘Clean Code’ yazmaya çalışır.

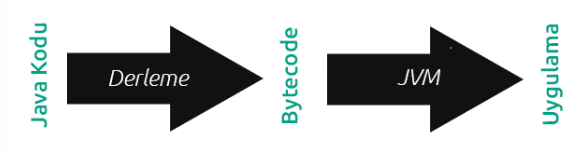
## Interface Segregation Principle

“Arayüz Ayırımı” prensibinde**; bir interface’e gerekenden fazla sorumluluk eklemek yerine, daha özelleştirilmiş birden fazla interface oluşturulmalıdır**. Nesneler, ihtiyacı olmayan özellik veya metotlar içeren interface’leri miras almaya zorlanmamalıdır. Sizinde farkettiğiniz üzere “Single Responsibility” ve “Interface Segregation” prensipleri birbirine oldukça benzemekte ve aynı amaca hizmet etmektedirler. Ancak burada gözden kaçırılmaması gereken en önemli husus şudur ki; ‘Interface Segregation’ prensibi interface’ler ile ilgilenirken, ‘Single Responsibility’ prensibi class’lar ile ilgilenmektedir.

# JVM Nedir?

Java’ nın **“bir kez yaz, her yerde çalıştır”** felsefesi de platform bağımsızlığını ifade eder. Platform bağımsızlığını sağlayan bileşen (Java Virtual Machine) Java Sanal Makinasıdır. **JVM**, tüm platformlarda Java kodlarını çalıştırmak üzere geliştirilmiş ve hemen her platforma uygun sürümü olan bir bileşendir. Linux ve Windows birbirinden çok farklı platformlar olmasına rağmen Java ile geliştirilmiş bir yazılım bu iki platformda da bulunan Java Sanal Makinası sayesinde kullanılabiliyor.

## JVM Nasıl çalışır?



Programcının yazdığı Java kodları geliştirme ortamı tarafından yazım (syntax) hatalarına karşı kontrol edilir. Hatalar giderildiğinde, JDK paketindeki derleyici (**compiler**) aracılığı ile Java kodları **bytecode** denilen bir ara dilin kodlarına dönüştürülür. Üzerinde çalışılan sistemdeki **JVM** **bu bytecode’ u yorumlar ve çalıştırır.**

# Java %100 OOP midir?

Hayır değildir. Çünkü içerisinde primitive tipler bulunmaktadır. Bu sebeple değildir. Ayrıca bir programlama dilinin %100 nesne yönelimli olması için sağlaması gereken bir kurallar kümesi vardır. Bunları şu şekilde listeleyebiliriz:

* Encapsulation / Data hiding -> Kapsülleme, sarmalama, paketleme / Veri gizleme
* Inheritance -> Kalıtım, miras
* Polymorphism -> Çok çeşitlilik
* Abstraction -> Soyutlama
* Tüm öntanımlı tiplerin nesne olması
* Tüm kullanıcı tanımlı tiplerin nesne olması
* Tüm işlemlerin nesnelere mesaj gönderilerek gerçekleştirilmesi

%100 nesne yönelimli olan dillere **Eiffel**, **Smalltalk** ve **Ruby** örnek olarak verilebilir.

NOT= Java String türü primitive tip **DEĞİLDİR**.

# Software Architecture nedir?

Bir bina yapılmaya başlamadan önce mimarlar tarafından projenin ön çizimi, tasarımı çizilir. Tıpkı bunun gibi bir yazılım projesinin de yapılmaya başlamadan önce planlanması gerekir. Bu planlamaya “Yazılım Mimarisi” bu planı tasarlayan kişilere de “Yazılım Mimarı” denir. Mimari, yazılım uygulamasının bir donanımın, ağların ve bir işletmenin diğer bileşenleriyle nasıl etkileşime gireceğini ana hatlarıyla anlatan eksiksiz bir tasarım belgeleri seti içerir. Böylelikle yazılım geliştiricilerin izleyeceği yol genel hatları ile belirlenmiş olur.

# Java 8 ile gelen yenilikler nelerdir?

1. Java 7 de gelen array lerin multi threading çalıştırılabilmeleri, java nın çok çekirdekli işlemcilerde çalışmasını anlamlı kılıyordu. Ancak, java 8 de bu yetenek Collection lar üzerinde de uygulanabilir hale gelmiştir.(Bknz. Stream API implementation)
2. Garbage collector ile ilgili olan “PermGen” Stack’ ten Heap’ e taşınmıştır.
3. Java 8 öncesinde herhangi bir Interface sınıfında gövdeli(implement) metod yazmak mümkün değildi. Java 8 ile gelen default keyword ü ile artık Interface sınıflarda gövdeli metot yazmak mümkün. Spring Data API sinde, bu yenilik kullanılarak ORM yeteneklerinin daha da güzelleştiğini gözlemlemekteyiz.
4. Lambda ifadelerini uygulayabilmek için, functional interface ler geldi. İçerisinde, tek bir abstract methodu olan ve birden fazla static ve default metodlar barındırabilen interface lere functional interface denir.
5. Lamda expression lar geldi. Lamda expressionlarda daha az satır kod ile imperative programlamada yazdığımız kodları yazabilmekteyiz. Stream API içerisinde lambda expressionlar sıklıkla kullanılmaktadır.
6. Imperative programming den , functional programming e geçildiğini gözlemliyoruz. Functional programming örneği olarak; “bana filtrelenmiş listeyi getir”, imperative ise ; “filtrelenmiş yeni liste nesnesi oluştur, liste içerisinde dön, her liste elemanının içeriği filtreye uygunsa yeni liste elemanına ekle, son eklenmiş listeyi getir, …” şeklinde örnek verilebilir.
7. Methodlar “::” söz dizimi ile referans verilebilmektedir. Static methodlar class name ile, static olmayan methodlar ise instance objeleri ile referans verilebilmektedir.

Örnek;

testList.forEach(TestClass::staticMetod);

TestClass testClass=new TestClass();

testList.forEach(testClass::nonStaticMetod);

1. Javascript kodlarının çok hızlı çalışmasını sağlayan, Nashorn javascript engine geldi.
2. AtomicLong dan daha yüksek başarımlı sayaç işlemlerinin yapıldığı LongAdder objesi geldi. AtomicLong a gore daha fazla bellek kullanmaktadır.
3. Nesnelerin NullCheck işlemleri için “Optional” isimli özel bir utilityclass geldi.

# SERİLİZATİON NEDİR?

Serialization objeleri Byte streama( ObjectInputStream(), ObjectOutputStream()) dönüştürdüğümüz işlemlerdir.Ve böylece objelerimiz dosyaya ,veritabanına veya hafızaya kaydedebiliyoruz.

Bunun tersine ise Deserialization deniyor. Ve Serialization işlemi kullanımı önerilmemektedir.buna alternatif olarak json formatı kullanılabilir.

Eğer serial versiyon vermezisek sistem kendi yapısna göre kriptografik karma işlemiyle kendi oto üretir ve Maliyetli ,hata ve güvenlik açıkları oluşur İstisna sınıfımın neden serileştirilmesi gerekiyor?

Bunun nedeni, tüm istisnalar için kök sınıf olan Throwable'ın Serileştirilebilir arabirimi uygulamasıdır. Varsayılan olarak tüm istisnalar hale getirilebilir ve bu bir dil tasarımı kararıdır çünkü yazarlar

istisnaların herhangi bir özel konfigürasyon olmadan kablo üzerinden gönderilebilmesini istediler. Temel sınıf serileştirilebilir değilse, yerleşik istisna türleri üzerinde hiçbir kontrolünüz olmayacağından, uzak bir yöntemin başarısız olması durumunda tam olarak neyin yanlış gittiğini iletmekte zorlanabilirsiniz.

# Yazılım Prensibleri

## Clean Code

// Gereksiz verilerden,tekrarlardan uzak okunabiliği, yazılabilirliği yüksek, Kodun temiz olması, kodu yazan geliştirici dışında ekiptekilerin kodu kolay şekilde anlayabilmesi ve geliştirme yapabilmesidir.

* **Readability**, (Basitçe okunup anlaşılıyor )
* **Changeability**, (Basitçe değiştirilebiliyor )
* **Extensibility** (Basitçe genişletilebiliyor )
* **Maintainability**. (Basitçe bakım yapılabilsin)
* **Isimlendirme**: Net bir şekilde belirtmemiz lazım.

## Spagetti Codes:

Clean Code tam tersi durumuna denir. Gereksiz, okunabilirliği zor, düzensiz iç içe girmiş for-while.vs olan kodlara denir.

## KISS (Keep It Simple Stupid)

// Kodlarımızda her zaman için en basit yolla sonuça varmaktır. Kolay bir yol ile çözülebilecek problemleri ya da başka bir ifade ile bir kaç satır kod ile geliştirilebilecek bir metodu, gereksiz ifadelerden kaçınmaktır.

## DRY (Don't Repeat Yourself)

Kodlarımızda sürekli kendimizi tekrar yapmamız gereksiz ve sakıncalıdır.

Çözüm: Class, Metotları

## Dammy Code (Kukla Kod)

Classlarımızın içinde interface'leri implement ederek , eklenen gereksiz ve kullanılmayan kod bütüne denir.

## YAGNI – You Ain't Gonna Need It(Buna İhtiyacın Olmayacak)

YAGNI, ihtiyacımız olmayacak şeyleri sisteme dahil etmemeyi söyleyen bir prensip.

## RREP:Reuse-Release Equivalence Principle (REP)

Tekrar Kullanım ve Sürüm Eşitliği Projemizde paketlemeler çok önemlidir Classlarımızı paketler yardımıyla saklarız Common roles: paketler küçük harflerle yazılır.

## Monad:

Metotları iç içe kullanmaya denir. Metotun çıktısı , metotun girdisidir.

## İNTERFACE

önce söz veriyorum: kilo vereceğim imza (Gövdesiz metot)

spor salona gitmek (Gövdeli metottur)

gövdeli metot - gövdesiz metot yazabilirsiniz

keyword: implements

istediğimiz kadar implement yazıp virgülle ekleyebiliriz. Polimorphism yapabiliriz.

# ACCESS MODİFİER

Abstraction=interface+abstract

inheritance : public ,private ,protected , default(-) new ile

oluşturabilirsin , 1tane extends

abstract : public ,protected , default(-) new ile oluşturamazsın , 1tane

extends

interface : public ,protected , default(-) new ile oluşturamazsın , Ntane

implements

# inheritance-abstract-interface

temiz kod yazmamıza olanak sağlar

polymorphism sağlar

maliyeti düşürür.

kod karmaşıklığından kurtarır.

az kod çok iş

# abstract-interface ortak özellikleri

polymorphism yapabiliriz.

soyutmalama yapabiliriz

gövdeli metot yazabiliriz.

gövdesiz metot yazabiliriz.

# abstract-interface farklı özellikleri

abstract -interface farklı özellikleri

abstract ==>ortak özelliklerde kullanmak

extends(1),

IS-A

interface ==> farklı ortak özelliklerde

implements(N)

CAN-DO

100 % soyutlamadır

public static int sayi=44; //sabit değişken oluşturabilirsiniz

# Lambda expressions

işlevsel programlamayı kolaylaştırır ve geliştirmeyi çok basitleştirir. Koleksiyondan verilerin yinelenmesine, filtrelenmesine ve çıkarılmasına yardımcı olur.

Java'da lambda expression'ı şu şekilde tanımlayabiliriz.

(parameter list) -> {body}

lambda expression kendi başına yürütülmez. Bunun yerine, functional interface tarafından tanımlanan abstract bir yöntemin uygulamasını oluşturur. Bu nedenle, önce bir functional interface tanımlamamız gerekiyor.

# Stream API

Java 8 ile birlikte gelen Stream API fonksiyonel programlama yaklaşımına göre koleksiyon, dizi gibi veriler üzerinde filtreleme, döngü, dönüştürme, dönüşüm gibi işlemler yapmayı sağlar.

**Function interface**, içerisinde sadece bir tane abstract metodu olan **interface**'dir. Eğer ilgili **interface**'in türetildiği **interface**'de abstract metot varsa bu durumda da **functional interface** olur. **Functional interface**'ler, lambda expression'ların kullanılabilmesi için tanımlanırlar.

Pi değerini döndüren bir Java programı:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

import java.lang.FunctionalInterface;

// this is functional interface

@FunctionalInterface

interface MyInterface{

// abstract method

double getPiValue();

}

public class Main {

public static void main( String[] args ) {

// declare a reference to MyInterface

MyInterface ref;

// lambda expression

ref = () -> 3.1415;

System.out.println("Value of Pi = " + ref.getPiValue());

}

}

çıktısı:

Value of Pi = 3.1415

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_