**JAVA SPRING BOOT BOOTCAMP 1. HAFTA ÖDEVİ**

**SORU 1: Neden OOP’ye ihtiyaç duyarız? Önemli OOP dilleri nelerdir?**

OOP yazılım geliştirirken kullanılan ve gerçek dünya problemlerin daha organize ve yeniden kullanılabilir şekilde çözmek için kullanılan bir programlama paradigmasıdır.

**Nesne Yönelimli Programlamanın sağladığı kolaylıklar**

* Gerçek dünyadaki nesnelerin tasarımları sınıf içinde yapılır.
* Sınıftan nesne üretilip değişiklik yapılmak istendiğinde tüm programda değişiklik yapmak gerekmez, sadece oluşturulan nesnenin sınıf içinde değişiklik yapılması yeterlidir.
* Oluşturulan nesneler birbirinden bağımsız olduğu için bilgi gizleme olanağı artar.
* Nesne oluşturma, bir sınıf içerisinde gerçekleştirilir ve bu kodlar başka projelerde kullanılabilir.
* Oluşturulan sınıflar yardımıyla daha az kod ile daha fazla iş yapılıp kod tekrarı önlenir.
* Kod tekrarı önlendiği için geliştirme sürecinin verimliliği artar.

Avantajları ise :

1. **Modülerlik:** Kod, bağımsız sınıflar ve nesneler halinde organize edilir, bu da yönetimi kolaylaştırır.
2. **Yeniden Kullanılabilirlik:** Bir kez yazılan kod, miras alma (inheritance) ve bileşim (composition) sayesinde tekrar kullanılabilir.
3. **Ölçeklenebilirlik:** Büyük projelerde kodun düzenli olması, uygulamanın genişletilmesini kolaylaştırır.
4. **Bakım Kolaylığı:** Kodun modüler yapıda olması, hata ayıklama ve geliştirme işlemlerini kolaylaştırır.
5. **Güvenlik:** Kapsülleme sayesinde veriye yalnızca belirlenen metotlar üzerinden erişim sağlanır, bu da veri bütünlüğünü korur.

OOP’nin dört temel prensibi vardır:

* **Encapsulation (Kapsülleme):** Veriyi ve bu veriye erişim yöntemlerini (metotlar) bir arada tutar. Veriye yalnızca belirlenen metotlarla erişim sağlanır.
* **Inheritance (Miras Alma):** Bir sınıfın özelliklerinin ve metotlarının başka bir sınıf tarafından kullanılmasını sağlar.
* **Polymorphism (Çok Biçimlilik):** Aynı isimdeki metotların farklı davranışlar göstermesini sağlar.
* **Abstraction (Soyutlama):** Gereksiz detayları gizleyip yalnızca gerekli özelliklerin gösterilmesini sağlar.

**Önemli OOP Dilleri**

* Java: Platform bağımsız çalışması ve geniş kütüphane desteği ile popülerdir.
* C#: .NET platformunda kullanılan güçlü ve esnek bir OOP dilidir.
* Python: Basit sözdizimi ve güçlü kütüphane desteği ile hızlı geliştirme sağlar.
* C++: Yüksek performans gerektiren uygulamalarda sıkça tercih edilir.
* Ruby: Dinamik yapısı ve esnekliği ile web uygulamalarında kullanılır.

**SORU 2: Interface vs Abstract Class?**

İkiyapı da abstraction amacıyla kullanılmakla beraber iki yapı arasında ve kullanımları arasında farklar bulunmaktadır.

|  |  |  |
| --- | --- | --- |
| Özellik | Interface | Abstract Class |
| Metotlar | Metotlar public abstract olarak tanımlanır(java 8 ve sorasında default ve static metotlar eklenebilir) | Hem soyut hem de somut metotlar içerebilir |
| Değişkenler | Tüm değişkenler public static final olarak tanımlanır | Örnek değişkenler tanımlanabilir ve değerleri değiştirilebilir |
| Miras | Bir sınıf birden fazla interface’i implemente edebilir | Bir sınıf yalnızca bir tane abstract class extend edebilir |
| Constructor | Constructor içermez | Constructor içerebilir |
| Erişim belirleyiciler | Metotlar varsayılan olarak public kabul edilir | Metotlar herhangi bir erişim belirleyicisine sahip olabilir |

**Ne Zaman Kullanmalıyım?**

* **Interface:** Eğer birçok sınıfın ortak davranışları paylaşmasını istiyorsan ve birden fazla yapıdan miras alma ihtiyacın varsa kullanılır.
* **Abstract Class:** Eğer ortak davranışlar dışında bazı ortak özellikleri (fields) de paylaşmak istiyorsan tercih edilir.

Interface yapısı implements anahtar kelimesi ile uygulanır.

Abstract Class yapısı extends anahtar kelimesi ile genişletilir.

Interface çoklu kalıtıma izin verirken, Abstract Class yalnızca tek bir sınıf tarafından genişletilebilir.

Genel olarak abstract sınıfların özelliklerini bir araya toplarsak;

Abstract sınıfları genel olarak inheritance (kalıtım) uygularken kullanırız.

new anahtar sözcüğü ile nesneleri oluşturulamaz.

İçerisinde değişken ve metod bulundurabilir.

Abstract sınıflardan türetilen sınıfların abstract metodları implement etmesi zorunludur. Diğer metodları override etmeden de kullanabilir.

Constructors (yapıcı metodlar) ve destructors (yıkıcı metodlar) bulundurabilirler.

Static tanımlanamazlar. ( Tanımlanmaya çalışılırsa compiler “an abstract class cannot be sealed or static” hatası verir)

Bir sınıf yalnızca bir abstract sınıfı inheritance yoluyla implement edebilir. Çoklu kalıtım (multiple inheritance) desteklenmez.

Abstract olmayan metodları da bulundurabilir.  
  
Interface

Interface, içerisinde sadece kendisinden türeyecek olan sınıfların içini dolduracağı metod tanımlarının bulunduğu ve soyutlama yapmamıza olanak sağlayan bir yapıdır.

Interface hakkında internette bir arama yaptığınızda genellikle kendisini inherit alan sınıflar için bir kontrat olduğundan bahsedilir. Kontrat kelimesiyle anlatılmak istenen tam olarak budur. Türeyen sınıflar, interface’in içerisinde yer alan tüm metodları implement edeceğine dair bir söz vermekte, sözleşme yapmaktadır. Aksi halde derleyici hata verecek, kodunuz çalışmayacaktır.

Genel olarak Interface’lerin özelliklerinden bahsetmemiz gerekirse;

new keywordü ile nesneleri oluşturulamaz.

Bir sınıfın ne yapması gerektiğini belirtir, nasıl yapması gerektiğini değil.

Default olarak tüm Interface üyeleri abstract ve public olarak tanımlanır. Sizin özellikle belirtmeniz gerekmez.

Bir sınıf birden fazla interface’i inherit edebilir, çoklu kalıtım (multiple-inheritance) desteklenir.

İçerisinde yalnızca metodların imzaları yer alır, içi dolu metod bulunduramazlar.

**SORU 3: Neden equals ve hashcode'a ihtiyacımız var? Ne zaman override etmeli?**

Java'da equals() ve hashCode() metotları nesneler üzerinde karşılaştırma ve koleksiyon yapılarında (örneğin HashMap, HashSet) etkili çalışma açısından kritik öneme sahiptir.

**Neden equals() Metoduna İhtiyacımız Var?**

* equals() metodu, iki nesnenin **mantıksal eşitlik** durumunu kontrol etmek için kullanılır.
* Varsayılan olarak, Java'da equals() metodu Object sınıfından miras alınır ve == operatörü gibi çalışır. Yani bellekteki adreslerini karşılaştırır.

### Neden hashCode() Metoduna İhtiyacımız Var?

* hashCode() metodu, nesnenin bellekteki konumuna göre benzersiz bir tam sayı döndürür.
* HashMap, HashSet gibi veri yapıları, hashCode() metoduna dayanarak nesneleri gruplar. Eğer hashCode() düzgün çalışmazsa bu yapılar beklenmedik sonuçlar verebilir.

**equals() Ne Zaman Override Edilmeli?**

* Nesnelerin içeriklerine göre eşit olup olmadığını kontrol etmek istediğinizde.
* Özellikle HashMap, HashSet gibi koleksiyon yapılarında veri ekleme/çıkarmada sorun yaşamamak için gereklidir.

**hashCode() Ne Zaman Override Edilmeli?**

* equals() metodunu override ettiğiniz her durumda hashCode() da override edilmelidir.
* Koleksiyonlarda nesnelerin verimli depolanmasını ve hızlı erişilmesini sağlamak için gereklidir.

**SORU 4: Java’da Diamon Problemi ve çözümü nedir?**

Java'da **Diamond Problem** (Elmas Problemi), çoklu kalıtımın getirdiği bir çakışma problemidir ve özellikle **interface** yapılarında ortaya çıkabilir. Java'nın tasarımı gereği **class** yapılarında çoklu kalıtım desteklenmez; ancak **interface** yapılarında bu sorun ortaya çıkabilir ve özel çözümler gerektirir.

Diamond problemi, bir sınıfın aynı atadan gelen iki farklı yol üzerinden aynı metodu miras alması sonucu ortaya çıkan **çakışma** problemidir. Java hangi metodu çağıracağına karar veremez ve bu durum **diamond problemi** olarak adlandırılır.

Java'da **diamond problemi** super anahtar kelimesiyle çözülür. super anahtar kelimesi, hangi interface'in default metodunun çağrılacağını belirtir.

//Java'da bir sınıf yalnızca tek bir class'ı genişletebilir (extends).  
 //Bir sınıf birden fazla interface'i implemente edebilir (implements).  
 //Diamond problemi yalnızca interface'lerde default metotlarla ortaya çıkar.  
 //super anahtar kelimesi, hangi interface'in default metodunun çağrılacağını belirtmek için kullanılır.

**SORU 5: Neden Garbage Collector'a ihtiyacımız var? Nasıl çalışır?**

**Garbage Collector** (Çöp Toplayıcı), Java'nın bellek yönetiminden sorumlu olan mekanizmadır. Java'da bellek tahsisi (allocation) ve bellek temizliği (deallocation) işlemleri otomatik olarak gerçekleştirilir. Bu işlem, manuel bellek yönetiminin zorluklarını ortadan kaldırarak yazılımcının hata yapma riskini azaltır.

- **Garbage Collector**, Java'nın bellek yönetiminden sorumlu mekanizmasıdır.  
- Manuel bellek yönetimine gerek kalmadan Heap belleğindeki kullanılmayan nesneleri temizler.  
-Erişilemeyen nesneleri otomatik olarak tespit ederek bellekte yer açar ve performansı artırır.  
✅ Serial GC, Parallel GC, G1 GC, ZGC gibi farklı algoritmalarla çalışabilir.

**Garbage Collector'a Neden İhtiyacımız Var?**

Java'da **Garbage Collector** aşağıdaki nedenlerle kritik öneme sahiptir:

**1. Bellek Sızıntısını (Memory Leak) Önler**

* Bir nesneye referans kaybedildiğinde veya artık kullanılmayan nesneler bellekten silinmezse bellek sızıntısı oluşur. GC bu tür nesneleri tespit ederek belleği temizler.

**2. Geliştiriciyi Elle Bellek Yönetiminden Kurtarır**

* C ve C++ gibi dillerde bellek tahsisi ve serbest bırakma işlemleri manuel yapılır. Java'da bu süreç GC tarafından otomatik olarak gerçekleştirilir.

**3. Performansı Optimize Eder**

* Belleği etkili şekilde yöneterek uygulamanın daha verimli çalışmasını sağlar.

**3. Garbage Collector Nasıl Çalışır?**

Garbage Collector'ın temel çalışma prensibi **nesne erişilebilirliği** (object reachability) kontrolüne dayanır. Çalışma mantığı şu adımları izler:

**1. Root Set Tespiti**

* GC, bellekte erişilebilecek tüm kök referansları (Root Set) tespit eder.
* **Root Set** şunları içerir:
  + **Static değişkenler**
  + **Stack'teki yerel değişkenler**
  + **Aktif thread'lerin referansları**

**2. Reachability (Erişilebilirlik) Kontrolü**

* GC, kök referanslardan başlayarak erişilebilen tüm nesneleri işaretler. Erişilemeyen (referansı olmayan) nesneler "çöp" olarak işaretlenir.

**3. Nesnelerin Temizlenmesi (Cleanup)**

* Erişilemeyen nesneler bellekten temizlenir ve bu bellek geri kazanılır.

**4. Bellek Sıkıştırma (Compaction)**

* Temizlikten sonra boş alanların sıkıştırılması işlemi yapılır. Bu, belleğin daha verimli kullanılmasını sağlar.

**SORU 6: ‘static’ anahtar kelimesi kullanımı nasıldır?**

Java'da static anahtar kelimesi, bir sınıfın (class) belirli bileşenlerini (değişken, metot, blok veya iç sınıf) nesne oluşturmadan doğrudan erişilebilir hale getirmek için kullanılır.

Static kelimesi sınıfa (class) bağlıdır, yani bu bileşenler nesneye değil sınıfa aittir. Java'da static static metot, static blok, static değişken, static iç sınıf yapılarda kullanılabilir.

**-Static Kullanırken Dikkat Edilmesi Gerekenler**

\*static metotlar yalnızca static değişkenlere erişebilir.  
\* static bloklar yalnızca **bir kez** çalıştırılır.  
\* static iç sınıflar, dış sınıfın nesnesi oluşturulmadan kullanılabilir.  
\* static metotlar **this** ve **super** anahtar kelimelerini kullanamaz çünkü static metotlar nesneye değil, doğrudan sınıfa bağlıdır.

**Özetle:**

- static anahtar kelimesi, nesne oluşturmadan sınıf üyelerine erişim sağlar.  
- static değişkenler, tüm nesneler arasında paylaşılan tek bir kopyaya sahiptir.  
- static metotlar doğrudan sınıf adıyla çağrılabilir.  
- static bloklar, sınıf belleğe yüklendiğinde yalnızca bir kez çalışır.  
- static iç sınıflar, dış sınıfın nesnesine gerek kalmadan çağrılabilir.

**SORU 7: Immutability (Değişmezlik) nedir? Nerede, nasıl ve neden kullanılır?**

**Immutability** (Değişmezlik), bir nesnenin oluşturulduktan sonra **değiştirilemez** olması anlamına gelir. Java'da String sınıfı gibi bazı sınıflar immutable (değişmez) olarak tasarlanmıştır.

Eğer bir nesne immutable ise:

* Değiştirilemez.
* Yeni bir değer atanmak istendiğinde yeni bir nesne oluşturulur.

Immutable nesneler, **Thread-Safe** (çoklu iş parçacığı güvenliği) gibi avantajlar sağlar ve genellikle güvenli ve stabil bir yapı için tercih edilir.

**Avantajları:**

- **Güvenlik:** Immutable nesneler değiştirilemez olduğu için daha güvenlidir. Özellikle şifreleme, kullanıcı bilgisi gibi hassas verilerde tercih edilir.  
- **Thread-Safe:** Immutable nesneler aynı anda birden fazla iş parçacığı tarafından güvenle kullanılabilir.  
- **Kolay Hata Ayıklama:** Immutable nesneler değiştirilemez olduğu için beklenmedik veri değişimlerinin önüne geçer.  
- **Verimlilik:** Özellikle String gibi sık kullanılan veri türlerinde immutable yapılar bellek kullanımını optimize eder.

**Immutability Nasıl Sağlanır?**

Bir sınıfın immutable olması:

**1. Sınıfı final olarak tanımlanmalı.**

* Bu, sınıfın başka bir sınıf tarafından miras alınmasını engeller.

**2. Tüm alanları (fields) private ve final olarak tanımlanmalı.**

* private → Dışarıdan erişimi engeller.
* final → Alanın yalnızca bir kez atanabilmesini sağlar.

**3. Setter metotlarını tanımlanmamalı.**

* Eğer setter metodu tanımlanırsa, nesne içerisindeki değerler değiştirilebilir olur ve immutable yapı bozulur.

**4. Constructor (Yapıcı Metot) ile yalnızca başlangıç değerleri belirlenmeli.**

* Değerler yalnızca constructor (kurucu metot) aracılığıyla atanmalıdır.

**5. Değiştirilebilir (mutable) nesneleri defensive copying (savunma kopyalama) tekniğiyle korunmalı.**

* Eğer sınıf List, Map, Date gibi değiştirilebilir nesneler içeriyorsa, bu nesnelerin kopyasını oluşturup kullanılmalı.

**Immutability'nin Kullanım Alanları**

-**String Sınıfı:** Java'da String immutable'dır ve bu sayede performans ve güvenlik sağlar.  
- **Wrapper Sınıflar:** Integer, Double, Boolean gibi primitive veri türlerinin karşılığı olan sınıflar immutable'dır.  
- **Hassas Veriler:** Kullanıcı şifreleri, token'lar gibi güvenlik açısından kritik veriler immutable yapılarla korunur.  
- **Concurrent Uygulamalar:** Thread-safe yapılarda immutable nesneler verimli çalışır.

**Özetle:**

**\***Immutability, bir nesnenin oluşturulduktan sonra değiştirilmemesidir.  
\*Immutable nesneler güvenli, performanslı ve hatasız çalışmayı sağlar.  
\* Immutable yapılar final, private alanlar ve defensive copying teknikleriyle oluşturulur.  
\* String, Integer, Boolean gibi sınıflar zaten immutable'dır.

**SORU 8: Composition ve Aggregation nedir? Farkları nelerdir?**

**Composition**, iki nesne arasında **güçlü** bir ilişkiyi temsil eder. Bir nesne olmadan diğer nesne **varlığını sürdüremez**. Bu, "kısmi sahiplik" veya "yaşam döngüsü bağımlılığı" olarak da tanımlanır.

-**Güçlü İlişki (Strong Relationship)**  
- **Yaşam Döngüsü Bağımlılığı** — Biri yok edilirse diğeri de yok edilir.  
- **"Has-A" İlişkisi** ("Sahip Olma" ilişkisi)

**Aggregation**, iki nesne arasında **zayıf** bir ilişkiyi temsil eder. Bir nesne olmadan diğer nesne varlığını sürdürebilir. Bu ilişki "dışsal sahiplik" olarak da tanımlanır.

- **Zayıf İlişki (Weak Relationship)**  
- **Bağımsız Yaşam Döngüsü** — Bir nesne yok edilse bile diğeri varlığını sürdürebilir.  
- **"Has-A" İlişkisi** ("Sahip Olma" ilişkisi)

|  |  |  |
| --- | --- | --- |
| Özellik | Composition | Agrgregation |
| İlşiki türü | Güçlü İlişki | Zayıf ilişki |
| Yaşam Döngüsü | Bir nesne yok edilirse bağlı nesne de yok edilir. | Bir nesne yok edilse bile bağlı nesne varlığını sürdürür. |
| Bağımlılık | Bir nesne, diğerine tamamen bağımlıdır. | Nesneler birbirinden bağımsız çalışabilir. |
| Has-a İlişkisi | "Sahip Olma" ilişkisidir. | "Sahip Olma" ilişkisidir. |

**SORU 9:** **Cohesion (Tutarlılık) ve Coupling (Bağlılık) nedir? Aralarındaki farklar nelerdir?**

**Cohesion (Tutarlılık)**, bir sınıfın veya modülün **tek bir sorumluluğa** odaklanma derecesini ifade eder. Yüksek cohesion'a sahip bir sınıf yalnızca belirli bir görevi yerine getirir ve belirli bir amacı vardır.

**Yüksek Cohesion**

* Sınıfın yalnızca tek bir sorumluluğu vardır.
* Kod daha anlaşılır ve yönetilebilir olur.
* Bakım yapmak ve geliştirme daha kolaydır.

**Düşük Cohesion**

* Sınıf birden fazla sorumluluğa sahiptir.
* Kod karışır ve bakımı zorlaşır.
* Kodun okunabilirliği azalır ve hata yapma olasılığı artar.

**Coupling (Bağlılık)**, iki veya daha fazla sınıfın **birbirine bağımlı** olma derecesini ifade eder.

**Gevşek (Loose) Coupling**

* Sınıflar birbirinden bağımsız çalışır.
* Değişiklik yapıldığında diğer sınıflar üzerinde minimum etki bırakır.
* Kod daha esnek ve genişletilebilir olur.

**Sıkı (Tight) Coupling**

* Sınıflar birbirine sıkı şekilde bağlıdır.
* Bir sınıfta yapılan değişiklik diğer sınıfları etkileyebilir.
* Bakım yapmak ve genişletmek zorlaşır.

|  |  |  |
| --- | --- | --- |
| **Özellik** | **Cohesion (Tutarlılık)** | **Coupling (Bağlılık)** |
| **Tanım** | Bir sınıfın tek bir sorumluluğa odaklanma derecesidir. | Sınıflar arasındaki bağımlılık seviyesidir. |
| **Amaç** | Tek bir amaca odaklanarak kodun düzenli olmasını sağlar. | Sınıflar arasındaki bağımlılığı azaltarak esnekliği artırır. |
| **Yüksek Olması** | Yüksek cohesion tercih edilir. | Gevşek coupling tercih edilir. |
| **Düşük Olması** | Düşük cohesion kodun karmaşıklaşmasına yol açar. | Sıkı coupling kodun bakımını zorlaştırır. |

***Yüksek cohesion sağlandığında, sınıflar belirli bir sorumluluğa odaklandığı için doğal olarak diğer sınıflarla bağımlılığı azalır ve bu da gevşek coupling'e katkı sağlar. Bu nedenle, yüksek cohesion sağlandığında gevşek coupling elde etme olasılığı artar***

**SORU 10:** **Heap ve Stack bellek alanları nedir? Aralarındaki farklar nelerdir?**

**Heap Bellek**, Java uygulamalarında **nesnelerin** ve **dinamik veri** yapılarının depolandığı bellek alanıdır. Heap bellek, Java Sanal Makinesi (JVM) tarafından yönetilir ve Garbage Collector (Çöp Toplayıcı) ile temizlenir.

**// Heap Belleğin Özellikleri:**

✅ **Nesneler Heap'te saklanır.** Tüm nesneler new anahtar kelimesiyle oluşturulduğunda Heap'e yerleştirilir.  
- **Global erişim mümkündür.** Heap bellekte oluşturulan nesnelere tüm iş parçacıkları (threads) erişebilir.  
- **Daha geniş bellek alanı sağlar.** Heap bellek genellikle daha büyük ve uzun ömürlü nesneler için kullanılır.  
-**Garbage Collector (GC) temizliği sağlar.** Kullanılmayan nesneler GC tarafından otomatik olarak temizlenir.

**Stack Bellek**, Java'da her iş parçacığına (thread) ait olan ve metot çağrılarıyla çalışan **geçici** bellek alanıdır. Her metot çağrısı Stack bellekte bir "frame" oluşturur ve metot tamamlandığında bu frame bellekten kaldırılır.

**// Stack Belleğin Özellikleri:**

- **Yerel (Local) değişkenler Stack'te saklanır.** Metotların içerisine tanımlanan değişkenler bu alanda tutulur.  
- **Fonksiyon (Metot) çağrıları Stack'te çalışır.** Her metot çağrıldığında yeni bir "frame" oluşturulur ve bu "frame" metot tamamlandığında bellekten silinir.  
- **Daha hızlı çalışır.** Stack bellekte veriler son giren ilk çıkar (LIFO - Last In First Out) mantığıyla yönetilir, bu da işlemleri hızlandırır.  
- **Küçük ve geçici veri saklama için uygundur.** Kısa ömürlü ve hızlı erişim gerektiren veriler Stack'te tutulur.

|  |  |  |
| --- | --- | --- |
| **Özellik** | **Heap Bellek** | **Stack Bellek** |
| **Depolama Türü** | Nesneler ve dinamik veriler | Yerel değişkenler ve metot çağrıları |
| **Boyut** | Daha geniş alan sağlar | Daha küçük ve hızlıdır |
| **Yaşam Süresi** | Garbage Collector tarafından temizlenir | Metot tamamlandığında otomatik olarak silinir |
| **Erişim Hızı** | Stack'e göre daha yavaştır | Daha hızlı çalışır |
| **Erişim Kapsamı** | Global erişime açıktır | Yalnızca çağrıldığı metot kapsamında çalışır |

Java'da **Heap** ve **Stack** birbirini tamamlayan bellek alanlarıdır. Örneğin:

* Bir nesne oluşturulduğunda bu nesne **Heap** bellekte saklanır.
* Bu nesneye olan referans (değişken) ise **Stack** bellekte tutulur.

**Heap** bellekte oluşturulan nesneler Garbage Collector tarafından temizlenir; bu nedenle bellek sızıntılarını önlemek için referansların dikkatlice yönetilmesi gerekir.

**SORU 11:** **Exception (İstisna) nedir? Türleri nelerdir?**

**Exception (İstisna)**, Java'da program çalışırken oluşan **beklenmedik hataları** temsil eder. Java'da Exception'lar, programın akışını kesintiye uğratarak uygulamanın çökmesine neden olabilir. Bu nedenle Java, istisnaları ele almak için güçlü bir **exception handling** (istisna yönetimi) mekanizması sağlar.

**Java'da Exception Hiyerarşisi**

Java'daki tüm istisnalar, Throwable adlı ana sınıftan türemektedir.

Throwable

├── Error (Hata)

│ ├── OutOfMemoryError

│ ├── StackOverflowError

│

└── Exception (İstisna)

├── Checked Exception (Kontrollü İstisnalar)

│ ├── IOException

│ ├── SQLException

│

└── Unchecked Exception (Kontrolsüz İstisnalar)

├── NullPointerException

├── ArithmeticException

**1. Checked Exception (Kontrollü İstisnalar)**

* Derleme (compile-time) aşamasında tespit edilen istisnalardır.
* Java derleyicisi (compiler) bu tür istisnaların ele alınmasını **zorunlu** kılar. try-catch bloğu veya throws anahtar kelimesiyle ele alınmalıdır.

Örn: IOException, SQLException, FileNotFoundException

**2. Unchecked Exception (Kontrolsüz İstisnalar)**

* Çalışma zamanı (runtime) aşamasında meydana gelen istisnalardır.
* RuntimeException sınıfından türetilir ve derleyici tarafından kontrol edilmez.
* Genellikle programlama hatalarından kaynaklanır.

**Örnek:** NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException

**3. Error (Hata)**

* Error sınıfından türetilir ve genellikle ciddi sistem hatalarını ifade eder.
* **Garbage Collector (GC)** problemleri, bellek taşmaları veya JVM (Java Virtual Machine) ile ilgili sorunlardan kaynaklanır.
* try-catch bloğuyla genellikle ele alınmaz çünkü programın akışını tamamen kesebilir.

**Örnek:** OutOfMemoryError, StackOverflowError, VirtualMachineError

**----- Java'da Exception Handling (İstisna Yönetimi)-----**

Java'da istisnaları ele almak için şu yöntemler kullanılır:

**1. try-catch Bloğu**

İstisnaları ele almak için kullanılır. catch bloğu hatayı yakalar ve ilgili işlemi gerçekleştirir.

**2. finally Bloğu**

finally bloğu, hata oluşsa da oluşmasa da çalışır. Genellikle dosya kapatma, kaynak serbest bırakma gibi işlemler için kullanılır.

**3. throws Anahtar Kelimesi**

Metot tanımında kullanılarak belirli bir istisnanın fırlatılabileceği belirtilir.

**4. throw Anahtar Kelimesi**

Belirli bir istisnayı manuel olarak fırlatmak için kullanılır.

**SORU 12:** **Clean Code (Temiz Kod) nasıl özetlenir?**

1. Anlamlı ve Açık İsimlendirme (Meaningful Naming)
2. Fonksiyonlar Tek Sorumluluk Üstlenmelidir (Single Responsibility Principle - SRP)
3. Kısa ve Net Fonksiyonlar (Short Functions)
4. Yorumları (Comments) Gereksiz Kullanmamak
5. Kuru (DRY - Don't Repeat Yourself) İlkesi(Aynı kodu tekrar tekrar yazmak yerine, ortak kodları fonksiyon veya metodlara taşınmalı)
6. Anlamlı Hata Mesajları (Descriptive Error Messages)

**SORU 13:** **Java'da Method Hiding (Metot Gizleme) nedir?**

Java'da **Method Hiding** (Metot Gizleme), bir **üst sınıfın (parent class)** static metotlarının, **alt sınıf (child class)** tarafından **aynı isim ve aynı parametre listesi** ile tanımlanması durumudur.

**Önemli Fark:** Method Hiding yalnızca static metotlarda gerçekleşir ve **polimorfizm (override)** gibi çalışmaz.

**Method Hiding Kuralları**

\* static metotlar **override edilemez**, ancak aynı isimde yeniden tanımlanarak **gizlenebilir (hidden)**.  
\* Method Hiding sırasında @Override anotasyonu kullanılmaz.  
\* Üst sınıfın static metodu gizlenirken **erişim belirleyicisi (access modifier)** daha kısıtlayıcı olamaz (örneğin public olan metot private olarak gizlenemez).  
\* Method Hiding'de hangi metodun çalışacağı **nesne türüne değil, referans türüne** bağlıdır.

|  |  |  |
| --- | --- | --- |
| **Özellik** | **Method Hiding** | **Method Overriding** |
| **Metot Türü** | static metotlar için geçerlidir. | non-static metotlar için geçerlidir. |
| **@Override Kullanımı** | @Override anotasyonu kullanılmaz. | @Override anotasyonu kullanılır. |
| **Çalışma Zamanı Davranışı** | Referans türüne göre hangi metot çalışır. | Nesne türüne göre hangi metot çalışır. |
| **Erişim Belirleyici** | Alt sınıfın metodu, üst sınıftan daha dar erişim belirleyiciye sahip olamaz. | Erişim belirleyiciler daha geniş olabilir. |

**Kritik Nokta:**

* **Referans türü** → Hangi **static** metotların çalışacağını belirler.
* **Nesne türü** → Hangi **instance (örnek)** metotların çalışacağını belirler.

**SORU 14:** **Java'da Abstraction ve Polymorphism arasındaki farklar nedir?**

**Abstraction (Soyutlama)**, nesnenin yalnızca **gerekli özelliklerini** gösterip, **gereksiz detaylarını gizleme** işlemidir. Java'da abstraction, genellikle:

✅ **Abstract sınıflar** (abstract class) ✅ **Arayüzler** (interface)

yardımıyla gerçekleştirilir.

**Polymorphism (Polimorfizm)**, aynı isimdeki metotların **farklı davranışlar** sergilemesidir. Java'da polymorphism genellikle:

✅ **Metot Override (Geçersiz Kılma)**  
✅ **Metot Overload (Aşırı Yükleme)**  
yöntemleriyle sağlanır.

|  |  |  |
| --- | --- | --- |
| **Özellik** | **Abstraction (Soyutlama)** | **Polymorphism (Polimorfizm)** |
| **Amaç** | Gereksiz detayları gizleyerek yalnızca gerekli fonksiyonları ortaya çıkarır. | Aynı metot adı altında farklı davranışlar sağlar. |
| **Uygulama** | abstract sınıflar ve interface yapılarıyla gerçekleştirilir. | Override ve Overload yöntemleriyle gerçekleştirilir. |
| **Nesne Üretimi** | abstract sınıflardan doğrudan nesne üretilemez. | Polimorfizm, genellikle Parent referansı üzerinden Child nesnesi üretildiğinde uygulanır. |
| **Kullanım Amacı** | Yalnızca belirli özelliklerin görünür olmasını sağlar. | Farklı nesneler için aynı metot çağrılarında farklı davranış sağlar. |

**Override ve Overload Karşılaştırma Tablosu**

|  |  |  |
| --- | --- | --- |
| **Özellik** | **Override (Geçersiz Kılma)** | **Overload (Aşırı Yükleme)** |
| **Amaç** | Mevcut metodu yeniden tanımlama | Aynı metodu farklı parametrelerle tanımlama |
| **Metot Adı** | Aynı olmak zorunda | Aynı olmak zorunda |
| **Parametre Listesi** | **Aynı olmalı** | **Farklı olabilir** |
| **Dönüş Tipi** | Aynı veya alt tür (subtype) | Aynı veya farklı olabilir |
| **Erişim Belirleyici** | Daha kısıtlayıcı olamaz | Serbest |
| **Zorunlu Anotasyon** | @Override kullanılmalı | @Override gerekmez |