**Java Interview Soru ve Cevapları-@rehberkod**

1. **Java'da "static blocks" ve "static initializers" olarak adlandırılan yapılar nelerdir?**

Classın yüklendiği veya ilk kez kullanılmaya başladığı zaman çalıştırılan kod bloklarıdır. Bu yapılar, sınıfın genel başlatma işlemlerini gerçekleştirmek için kullanılır.

"Static blocks" veya "static initializers" olarak adlandırılan bu bloklar aynı şeyi ifade eder ve sınıfın yüklendiği an veya ilk kez kullanılmadan önce çalışır. Bu bloklar, genellikle sınıfın sabitlerini, statik değişkenlerini veya diğer başlangıç işlemlerini yapmak için kullanılır. Özellikle classın ilk yüklendiği anlarda bu tür işlemler yapmak önemlidir, çünkü bu işlemler yalnızca bir kez gerçekleşmelidir.

Aşağıda, Java'da static blokların nasıl kullanılacağına dair örnek bir kod parçası bulunmaktadır:

public class MyClass {

// Static initializer block

static {

// Bu blok sınıf yüklendiğinde çalışacak kodu içerir.

System.out.println("MyClass sınıfı yüklendi.");

}

// Diğer sınıf içeriği

// ...

}

1. **Bir constructor'ı başka bir constructor'dan nasıl çağırırız?**

Bir constructor içinden başka bir constructor'ı çağırmak için this() yöntemini kullanabiliriz. Ancak bu işlemi kullanırken bazı kısıtlamalara dikkat etmemiz gerekmektedir:

this() çağrısı, constructor içindeki ilk ifade olmalıdır: this() çağrısı, constructor içindeki diğer herhangi bir işlemden veya ifadeden önce gelmelidir. Yani, constructor içinde bir başka işlemi veya ifadeyi çalıştırmadan önce this() yöntemiyle diğer bir constructor'ı çağırmanız gerekmektedir.

Bir constructor içinde birden fazla this() çağrısı kullanılamaz: Bir constructor içinde sadece bir this() çağrısı kullanabilirsiniz. Aynı constructor içinde birden fazla this() çağrısı yapılmasına izin verilmez.

Örnek:

public class MyClass {

private int value;

public MyClass() {

this(0); // Bu, diğer constructor'ı çağırır

// Başka işlemler

}

public MyClass(int value) {

this.value = value;

}

}

Yukarıdaki örnekte, this(0) ifadesi, MyClass sınıfının parametresiz constructor'ından çağrı yapar ve bu çağrı, constructor içindeki ilk ifade olarak kullanılmıştır. Bu iki kurala uyulduğunda, constructor chaining (zincirleme çağrı) işlemi sorunsuz bir şekilde çalışır ve sınıfın farklı constructor'ları arasında kod tekrarı önlenmiş olur.

1. **Java’da method overriding nedir?**

Java'da method overriding, bir alt sınıfın, üst sınıfında zaten tanımlanmış bir metodun belirli bir uygulamasını sağlamasına izin veren bir nesne yönelimli programlama kavramıdır. Bir alt sınıf, üst sınıfındaki bir metodun adı, parametre türleri (imza) ve dönüş türü ile aynı olduğunda, bu metodun üzerine yazdığı söylenir.

Java'da method overriding'i neden ve ne zaman kullanırız:

1. **Özelleştirme:** Method overriding'i kullanırsınız çünkü bir alt sınıfta belirli bir metodun özelleştirilmiş bir uygulamasını sağlamak istiyorsunuz. Örneğin, bir **Araç** üst sınıfınızda bir **motoruCalistir** metodunuz olsun, ardından **Araç** sınıfının bir alt sınıfı olan **Araba** sınıfında, **motoruCalistir** metodunu bir arabanın özgü özelliklerine göre yeniden tanımlayabilirsiniz.
2. **Polimorfizm:** Method overriding, polimorfizmin önemli bir parçasıdır. Bu, farklı alt sınıfların nesnelerini genel bir üst sınıf olarak ele almanıza olanak tanır. Bir nesne üzerinde bir metodu bir üst sınıf referansı kullanarak çağırabilirsiniz ve alt sınıfta üzerine yazılmış bir metod çalıştırılır. Bu, daha genel ve esnek kod yazmak için kullanışlıdır.

Örneğin, bir **Araç** referansı kullanarak bir **Araba** nesnesinin **motoruCalistir** metodunu çağırdığınızda, bu metodun **Araba** sınıfındaki üzerine yazılmış hali çalışır.

Aşağıda örnek bir Java kodu bulunmaktadır:

class Araç {

void motoruCalistir() {

System.out.println("Bir aracın motoru çalıştırılıyor");

}

}

class Araba extends Araç {

@Override

void motoruCalistir() {

System.out.println("Bir arabanın motoru çalıştırılıyor");

}

}

public class Ana {

public static void main(String[] args) {

Araç araç = new Araba();

araç.motoruCalistir(); // Araba sınıfındaki üzerine yazılmış metod çalışır

}

}

Yukarıdaki örnekte, **motoruCalistir** metodu **Araba** sınıfında üzerine yazılmıştır ve bir **Araç** referansı kullanılarak bu metodu çağırdığınızda, **Araba** sınıfındaki üzerine yazılmış **motoruCalistir** metodu çalışır. Bu, Java'da method overriding ve polimorfizm kavramlarını gösterir.

1. **Java’da “süper” anahtar kelimesi nedir?**

Java'da "super" anahtar kelimesi, bir alt sınıfın üst sınıfının değişkenlerine, metodlarına veya constructor’ına erişmek için kullanılan bir mekanizmadır. Bu anahtar kelime, aşağıdaki iki şekilde kullanılabilir:

1. **İlk form, üst sınıf constructor’ını çağırmak için kullanılır:** Alt sınıfın kendi constructor’ının içinde, üst sınıfın constructor’ını çağırmak için "super" anahtar kelimesi kullanılabilir. Bu, alt sınıfın constructor’ının işini tamamladıktan sonra üst sınıfın constructor’ının çalışmasını sağlar.

Örnek:

**class ÜstSınıf {**

**ÜstSınıf() {**

**System.out.println("Üst sınıfın yapıcısı");**

**}**

**}**

**class AltSınıf extends ÜstSınıf {**

**AltSınıf() {**

**super(); // Üst sınıfın constructor’ını çağır**

**System.out.println("Alt sınıfın yapıcısı");**

**}**

**}**

1. **İkinci form, üst sınıf değişkenleri ve metodlarını çağırmak için kullanılır:** Bir alt sınıf içinde, üst sınıfın değişkenlerine veya metodlarına erişmek isterseniz "super" anahtar kelimesini kullanabilirsiniz. Bu, üst sınıfın değişkenleri veya metodları alt sınıf tarafından aynı isimle aynı isimle aşırı yazıldıysa dahi üst sınıfın üyelerine erişmenizi sağlar.

Örnek:

**class ÜstSınıf {**

**int sayı = 10;**

**void metot() {**

**System.out.println("Üst sınıfın metodu");**

**}**

**}**

**class AltSınıf extends ÜstSınıf {**

**int sayı = 20;**

**void metot() {**

**System.out.println("Alt sınıfın metodu");**

**}**

**void örnekMetod() {**

**System.out.println(super.sayı); // Üst sınıfın sayısını yazdır**

**super.metot(); // Üst sınıfın metodu çağrılır**

**}**

**}**

Yani "super" anahtar kelimesi, alt sınıfın üst sınıfının üyelerine erişmesine izin verir, aynı isimdeki üyelerin üzerine yazılmasına rağmen bu üyelere erişim sağlar. "super" her zaman birinci ifade olmalıdır, yani bir yöntemde veya yapıcıda ilk kullanılan ifade olmalıdır.

1. **Java’da metod overloading ve metod overriding arasındaki farklar nelerdir?**

|  |  |
| --- | --- |
| **Metod overloading** | **Metod overriding** |
| Metot Overloading aynı class içinde gerçekleşir. | Metot Overriding iki farklı class arasında gerçekleşir, yani bir üst class (superclass) ve bir alt class (subclass) arasında. |
| Miras (inheritance) kullanımı zorunlu değildir, yani aynı sınıf içindeki metotların farklı sürümlerini oluştururken, sınıfın kendisi üzerinde çalışılır. | Miras (inheritance) kullanılır, yani bir sınıfın diğer sınıfın özelliklerini alması ve bu özellikleri değiştirmesi gereklidir. |
| Overloading metotların dönüş türü (return type) aynı olmak zorunda değildir. | Overriding metotların dönüş türü aynı olmalıdır. |
| Parametreler farklı olmalıdır, yani aynı metot adı, ancak farklı parametre listeleri kullanılır. | Parametreler aynı olmalıdır, yani aşırı yazan metot aynı isimde ve aynı parametre listesi ile yazılmalıdır. |
| Statik çok biçimlilik (static polymorphism) metot overloading ile elde edilir. Hangi metotun çağrılacağı derleme zamanında belirlenir. | Dinamik çok biçimlilik (dynamic polymorphism) metot overriding ile elde edilir. Hangi metotun çağrılacağı çalışma zamanında belirlenir. |
| Overloading sırasında bir metot diğerini gizleyemez. Yani, aynı classta overloading metotlar birbirlerini gizleyemezler. | Overriding sırasında alt classın metodu, üst classın metotunu gizler. Yani, aynı isimde ve aynı parametre listesi ile yazılmışsa, alt classın metodu üst classın metotunu gizler ve çağrıldığında alt classın metodu çalışır. |

1. **Abstract Class ve interface arasındaki farklar nelerdir?**

|  |  |
| --- | --- |
| **Interface** | **Abstract Class** |
| İnterface yalnızca soyut (abstract) metotları içerir. | Abstract sınıf, soyut metotlar, somut metotlar veya her ikisini de içerebilir. |
| İnterfacedeki metotların erişim belirleyicileri (access specifiers) her zaman public olmalıdır. | Somut (concrete) sınıfın aksine, soyut sınıfın metotlarının erişim belirleyicileri (access specifiers) daha esnek olabilir; yalnızca private kullanılamaz. |
| İnterfacede tanımlanan değişkenler her zaman public, static ve final olmalıdır. | Abstract sınıf içindeki değişkenler, private dışında diğer erişim belirleyicilerini kullanabilir. |
| Java'da çoklu kalıtımı İnterfaceler kullanılarak uygularız. | Java'da abstract sınıflarla çoklu kalıtımı başaramayız, yani bir sınıf yalnızca bir soyut sınıftan türeyebilir. |
| Bir İnterface i uygulamak için "implements" anahtar kelimesi kullanılır. | Bir soyut sınıfı uygulamak için "extends" anahtar kelimesi kullanılır. |

1. **Java için neden platformdan bağımsızdır ifadesi kullanılır?**

Java'nın platform bağımsız olmasının en önemli özelliği, Java'nın kaynak kodunun herhangi bir platform üzerinde çalıştırılabilmesidir. Diğer birçok programlama dilinde, kaynak kodu derlendikten sonra platforma özgü bir yürütülebilir dosya oluşturulur. Bu dosyalar farklı işletim sistemleri ve platformlarda çalışmayabilir. Ancak Java'da, kaynak kodu "javac" derleyicisi tarafından derlendiğinde, ".class" uzantılı yürütülebilir bir dosya oluşturulur.

Bu ".class" dosyası, Java tarafından üretilen "byte code" adı verilen bir tür ara dil kodu içerir. Byte code, sadece Java Sanal Makineleri (JVM) tarafından yorumlanabilir. Sun Microsystems (şu anda Oracle Corporation) tarafından geliştirilen JVM'ler, farklı platformlar için sunulur. Bu nedenle, bir Java programının oluşturduğu byte code Windows ortamında üretildiyse, aynı byte code Linux ortamında da çalıştırılabilir. Bu, Java'nın platform bağımsızlığını sağlar.

Özetle, Java'nın platform bağımsız olmasının nedeni, byte code'un JVM'ler aracılığıyla farklı platformlarda yorumlanabilmesidir. Bu, aynı Java programının farklı işletim sistemlerinde ve donanım platformlarında çalıştırılabilmesini mümkün kılar.

1. **Java’da overloading methodu nedir?**

Java'da metod overloading, aynı isme sahip ancak farklı parametrelerle tanımlanmış iki veya daha fazla metodu ifade eder. Java'da metodlar aynı isme sahip olabilir, ancak parametrelerin türleri, sırası veya sayısı farklı olmalıdır. Metod overloading sayesinde aynı isim altında birden fazla metot tanımlayabiliriz ve bu metotlar farklı işlevler veya girdilerle çalışabilir.

Java'da static polymorphism, yani hangi metodu çağıracağınızın derleme zamanında belirlenmesi, metod overloading ile elde edilir. Derleyici, overloading metotların ismini, parametre sayısını ve parametre türlerini dikkate alarak hangi metodu çağıracağınızı belirler. Bu nedenle, aynı metot ismi altında birden fazla metot tanımlanabilir ve bu metotlar overload edilmiş olur.

Önemli bir not olarak, metotların dönüş türü (return type) metot signature ın bir parçası değildir. Yani, farklı dönüş türlerine sahip metotlar overload edilebilir, ancak bu dönüş türleri metodu çağırmak için yeterli değildir. Hangi metodu çağıracağınızı belirlemek için derleyici, metot ismi ve parametrelerin türleri ve sırası gibi faktörleri kullanır. Bu nedenle, overload edilmiş metotlar aynı ismi paylaşsa bile farklı parametrelerle çağrılabilir.

1. **JIT (Just-In-Time) derleyiciyisi nedir?**

JIT derleyici, Java bytecode'unu çalıştırılabilir kod haline getiren bir bileşen olarak JVM içinde bulunur. Ancak, JIT derleyici bir Java programını tamamen çalıştırılabilir kod haline dönüştürmez. Bunun yerine, Java programı çalıştırıldığında ihtiyaç duyulan kısımları bytecode'dan çalıştırılabilir kod haline çevirir. Böylece, Java programı daha etkili bir şekilde çalıştırılabilir.

1. **Java’da bytecode nedir?**

Bytecode, bir Java programının kaynak kodunun derlenmesi sonucunda oluşturulan .class dosyasında bulunan bir dizi talimat setidir. Bytecode, bir makine bağımsız dil olarak kabul edilir ve yalnızca Java Sanal Makinesi (JVM) tarafından yürütülmek üzere tasarlanmış talimatları içerir. Yani, Java kaynak kodu derlendikten sonra oluşturulan bytecode, JVM tarafından yorumlanabilir ve çalıştırılabilir. Bu, Java'nın platform bağımsızlığına katkıda bulunan önemli bir özelliktir, çünkü bytecode aynıdır, ancak farklı işletim sistemlerinde çalışan JVM'ler tarafından yürütülür.

1. **Java’da this() ve süper() keywords leri arasındaki farklar nelerdir?**

this():

this() anahtar kelimesi, aynı sınıf içerisindeki bir başka yapıyı (genellikle bir başka constructor'ı) çağırmak için kullanılır.

Genellikle overloading constructor'ları çağırmak veya aynı sınıf içindeki diğer constructor'larda tekrarlayan kodu önlemek için kullanılır.

this() anahtar kelimesi, aynı sınıf içindeki başka bir constructor'ı çağırmadan önce kullanılır ve constructor içinde sadece bir kere çağrılabilir.

İlgili constructor çağrıldıktan sonra bu constructor içindeki diğer işlemler devam eder.

super():

super() anahtar kelimesi, alt sınıfın constructor'ı içinde üst sınıfın constructor'ını çağırmak için kullanılır.

Java'da her alt sınıfın bir üst sınıfı vardır, ve bu nedenle alt sınıf constructor'ları, üst sınıf constructor'ını çağırmalıdır.

super() anahtar kelimesi, alt sınıf constructor'ının ilk satırında kullanılmalıdır.

1. **Java’da class kavramı nedir?**

Bir class, Nesne Yönelimli Programlama (OOP) içinde temel birimlerden biridir.

Bir class, nesnelerin tasarım şablonunu veya yapısını temsil eder.

Classlar, değişkenler (variables) ve metotları (methods) tanımlarlar.

Her class, belirli bir türde nesnelerin oluşturulabileceğini tanımlar. Örneğin, "Department" classı, departman türünde nesnelerin oluşturulabileceğini belirtir.

Java'da tüm program yapısı classlar içinde tanımlanır.

Bir Java uygulamasında en az bir class ve bir "main" metodu (main method) bulunmalıdır.

Bir class tanımı "class" anahtar kelimesi ile başlar.

Class tanımı, aynı isme sahip bir class dosyasında (".java" uzantılı) saklanmalıdır.

Class dosyalarının adı, classın adı ile aynı olmalıdır.

Class dosyaları derlendiğinde JVM (Java Sanal Makinesi), classı yükler ve bir ".class" dosyası oluşturur.

Program çalıştırıldığında, classı çalıştırır ve "main" metodu çalıştırır.

Yani, classlar, nesnelerin oluşturulmasını ve programların yapılandırılmasını tanımlamak için kullanılan temel yapı taşlarıdır.

1. **Java’da object nedir?**

Bir object, bir classın örneğidir. Yani, class bir türün (type) tasarımını veya şablonunu belirtirken, objectler bu türün somut örnekleridir.

Her object, bir classa aittir. Class, objectlerin temel yapısını ve davranışını tanımlar.

Bir object, iki temel bileşeni içerir: durum (state) ve davranış (behavior).

Durum, objectin özelliklerinin veya niteliklerinin değerlerini temsil eder. Bu özellikler, classın tanımladığı öznitelikler veya değişkenler olabilir.

Davranış, objectin yapabileceği işlemleri veya yöntemleri temsil eder. Bu yöntemler, classın tanımladığı işlevleri ifade eder.

Objectler ayrıca "örnek"(instance) olarak da adlandırılır.

Bir classın objectini oluşturmak için, new anahtar kelimesini kullanarak bir örnek (instance) oluşturulur. Örneğin, FirstClass classının bir örneğini oluşturmak için aşağıdaki gibi bir kod kullanılabilir:

FirstClass f = new FirstClass();

Burada f, FirstClass classının bir örneğini temsil eder ve bu örneğe erişmek için kullanılır.

Yani, bir object, bir classın belirlediği yapı ve davranışa sahip somut bir örneğidir. Objectler, classların özelliklerini ve işlevlerini kullanarak programın temel taşlarıdır ve programlar objectler aracılığıyla veri ve işlemler üzerinde çalışırlar.

1. **Method nedir?**

Java'da bir "metot" (method), bir class içindeki işlevsel bir bloktur ve belirli bir nesne üzerinde çalıştırılabilir. İşte

Metotlar, bir class içindeki işlevleri veya operasyonları tanımlarlar. Bir classın nesneleri, bu metotları çağırarak belirli işlemleri gerçekleştirebilir.

Bir metot, şu temel unsurları içerir:

Metot adı: Metodun adı, metodu tanımlayan bir isimdir. Bu isimle metodu çağırabilirsiniz.

Parametreler veya argümanlar: Metot, çalıştırılması sırasında aldığı verileri işlemek için parametreleri kullanabilir. Parametreler metot adının hemen sonra parantez içinde belirtilir.

Dönüş türü: Metot, bir sonuç döndürebilir veya dönüş yapmayabilir. Dönüş türü, metotun ne tür bir değer dönebileceğini belirtir. Örneğin, int veya float gibi bir veri türü olabilir.

Metotların temel sözdizimi şu şekildedir:

dönüş\_türü metot\_adı(parametre\_listesi) {

// Metodun işlevselliğini tanımlayan kod burada bulunur

}

Örnek bir metot tanımı:

public float toplama(int a, int b, int c) {

int sonuc = a + b + c;

return sonuc;

}

Metotlar birden fazla parametre alabilir ve parametreler virgülle ayrılır. Metotun işlevselliğini tanımlayan kod, süslü parantezlerin içine yerleştirilir.

Sonuç olarak, bir metot Java programında bir işlemi veya işlevselliği temsil eder. Bu işlevselliği çağırmak için metot adını ve gerekli parametreleri kullanabilirsiniz. Metotlar, programlarınızın düzenli ve modüler olmasına yardımcı olan önemli yapı taşlarıdır.

1. **Encapsulation nedir?**

Encapsulation, verileri bir tekil birim olan class içine sarmak veya yerleştirmek ve verilerin yanlış kullanımından korumak işlemidir.

Encapsulation (kapsülleme), verileri ve bu verilere erişim yöntemlerini (metotları) bir class içinde toplama ve düzenleme sürecini ifade eder. Bu, verilerin dışarıdan doğrudan erişilmesini sınırlar ve kontrol altına alır.

Java'da encapsulation, verilere erişim kontrolü ile desteklenir. Bu kontrol, dört erişim kontrol belirleyicisi (modifier) kullanılarak sağlanır: public, private, protected ve default (veya package-private).

Bir örnek üzerinden açıklamak gerekirse, bir "araba" classını ele alalım. Bir arabanın içinde birçok parça ve veri bulunur, ancak sürücünün sadece arabayı nasıl çalıştıracağı ve nasıl durduracağı hakkında bilgiye ihtiyacı vardır. Bu nedenle, encapsulation kullanarak sürücüye sadece gerekli olan bilgileri sunabilir ve geri kalan bilgileri gizleyebiliriz.

Encapsulation, verilere erişimi kontrol altına alarak veri güvenliğini artırır ve bir classın iç yapısını gizleyerek daha sade ve anlaşılır bir arabirim sunar.

Sonuç olarak, encapsulation (kapsülleme), verilerin gizliliğini korumak ve erişimi kontrol altına almak için kullanılan bir nesne yönelimli programlama prensibidir. Bu prensip, daha güvenli ve modüler yazılım geliştirmeye yardımcı olur.

1. **Java'da main methodu neden public, static ve void olarak tanımlanmıştır?**

public: public bir erişim belirleyicisidir ve bu, main methodunun diğer sınıflardan (veya sınıf dışından) erişilebilir olmasını sağlar. Java programlarının başlangıç noktası olan main methodu, JVM (Java Virtual Machine) tarafından çağrılmalıdır, bu nedenle public olarak işaretlenmesi, başka sınıflardan veya JVM'den erişilebilir olmasını sağlar.

static: main methodu nesne oluşturmadan çağrılabilmesi gereken özel bir methoddur. Java programının başlangıcı için kullanıldığı için bu method, sınıf düzeyinde (yani sınıfın bir örneği oluşturulmadan) çağrılmalıdır. Bu nedenle static olarak işaretlenir. Bir method static olarak işaretlendiğinde, bu method sınıfın bir örneği (nesnesi) oluşturulmadan çağrılabilir.

void: main methodu bir sonuç döndürmez. Java'da methodlar belirli bir türde bir değer dönebilirler, ancak main methodu programın başlangıcı olduğu için herhangi bir değer döndürmesine gerek yoktur. Bu nedenle void olarak işaretlenir. Yani, main methodu işlemleri gerçekleştirir, ancak herhangi bir değer geri döndürmez.

Özetle, public static void main(String[] args) ifadesi, Java programlarının başlangıcı için özel olarak tanımlanan main methodunun imzasını belirtir ve bu imza, programın başlatılmasını ve çalıştırılmasını sağlar. Bu nedenle, main methodu programın giriş noktasıdır ve yukarıdaki imza, Java'nın bu noktayı bulmasına ve çalıştırmasına yardımcı olur.

1. **Java’da constructor nedir?**

Java'da bir constructor, bir sınıfın örneklerini oluştururken bu örneklerin başlangıç değerlerini ayarlamak için kullanılan özel bir metoddur. İnşa edici metodlar, bir sınıfın yapısını ve davranışlarını belirlemek için kullanılan bir tür yapıcı işlevi görürler. İnşa ediciler, yeni bir nesne oluşturulduğunda otomatik olarak çağrılır ve nesnenin başlangıç durumunu ayarlamak için kullanılır.

Java'da iki temel türde constructor bulunur:

Default Constructor (Parametresiz Constructor): Bir sınıfın herhangi bir constructor tanımlanmadığında, Java otomatik olarak bir varsayılan constructor sağlar. Bu varsayılan constructor, sınıfın nesnelerini oluştururken herhangi bir argüman almayan ve genellikle sınıfın içinde herhangi bir işlem yapmayan bir constructor'dır. Örnek olarak:

public class MyClass {

public MyClass() {

// Default constructor

}

}

Parameterized Constructor (Parametre Alan Constructor): Parametreli constructorlar, nesneleri oluştururken belirli parametreleri alabilir ve bu parametrelere dayalı olarak nesnenin başlangıç durumunu ayarlar. Bu tür constructorlar genellikle sınıfın özelliklerini başlangıç değerleri ile doldurmak için kullanılır. Örnek olarak:

public class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

}

Yukarıdaki örnek, Person sınıfı için bir parametreli constructor gösterir. Bu constructor, name ve age adlı iki parametre alır ve bu parametreleri kullanarak Person nesnesinin başlangıç değerlerini ayarlar.

Constructorlar, sınıfın yapısını tanımlamak ve sınıf örneklerini başlatmak için önemli bir rol oynarlar ve Java'da sınıf oluştururken yaygın olarak kullanılırlar.

1. **Java'da length ve length() ifadeleri arasındaki fark nedir?**

length(): Bu, bir String nesnesinin metin uzunluğunu döndüren bir metoddur. length() bir metin nesnesinin karakterlerinin sayısını verir. Örnek olarak:

String str = "Hello World";

int uzunluk = str.length(); // uzunluk, "Hello World" ifadesinin karakter sayısı olan 11'i döndürecektir.

length() yöntemi bir String nesnesi üzerinde çağrılır ve bu nesnenin uzunluğunu karakter sayısı olarak döndürür.

length: Bu, dizilerin veya dizilimlerin (arrays) uzunluğunu döndüren bir özelliktir. Dizilerin boyutunu (kaç eleman içerdiğini) belirtir. Örneğin:

String[] days = {"Sun", "Mon", "Wed", "Thu", "Fri", "Sat"};

int uzunluk = days.length; // uzunluk, "days" dizisinin içerdiği eleman sayısı olan 6'yı döndürecektir.

length özelliği, bir dizi veya dizilim üzerinde kullanılır ve bu yapıdaki elemanların sayısını verir.

Özetle, length() bir String nesnesinin karakter sayısını döndürürken, length özelliği bir dizi veya dizilimin içerdiği eleman sayısını döndürür. Bu iki ifade arasındaki fark, kullanıldıkları veri türüne ve kullanım bağlamına dayanır.

1. **ASCII (American Standard Code for Information Interchange) kodu nedir?**

ASCII, bilgisayarların ve iletişim sistemlerinin metin karakterlerini temsil etmek için kullanılan bir karakter kodlaması standardıdır. ASCII, 0'dan 127'ye kadar olan karakterleri tanımlayan yedi bitlik bir karakter setidir. 0 ile 127 arasındaki karakterler, İngilizce alfabesi, rakamlar, temel noktalama işaretleri ve bazı kontrol karakterlerini içerir.

Bu standart, özellikle eski bilgisayar sistemlerinde ve yazılım uygulamalarında yaygın olarak kullanılmıştır. ASCII karakter seti, İngilizce dilindeki metinlerin temsilinde işe yarar, ancak diğer dillerin karakterlerini veya sembollerini desteklemez. Bu nedenle, ASCII karakter seti sınırlı bir karakter yelpazesine sahiptir ve yalnızca temel İngilizce karakterleri içerir.

Açıklamada belirtildiği gibi, C programlama dili geleneksel olarak ASCII karakterlerini kullanır. Bu nedenle, C dilinde yazılmış programlar genellikle İngilizce karakterlerle yazılmalıdır. Başka dillerdeki karakterlerin veya sembollerin C programlarına dahil edilmesi zordur, çünkü ASCII karakter seti bu karakterleri desteklemez.

Günümüzde, daha geniş karakter kümelerini ve uluslararası dil desteğini sağlayan Unicode gibi karakter kodlamaları, ASCII'nin yerini almıştır. Unicode, dünya genelinde farklı dillerdeki karakterlerin ve sembollerin temsilini sağlar ve çok daha geniş bir karakter yelpazesi sunar.

1. **Unicode nedir?**

Unicode, dünya genelinde tüm dillerin karakterlerini ve sembollerini temsil etmek amacıyla oluşturulan bir karakter kümesidir. Unicode, Unicode Consortium adlı bir kuruluş tarafından geliştirilmiştir ve farklı dillerdeki karakterlerin ve sembollerin benzersiz bir kodlanmış temsilini sağlar.

Aşağıdaki noktalara dikkat edilmelidir:

Unicode Karakter Aralığı: Unicode karakterleri 16 bit (2 byte) olarak temsil edilir, bu da toplamda 65,536 (0'dan 65,535'e kadar) farklı karakteri içeren bir karakter kümesi sağlar. Bu, dünya genelindeki birçok dilin karakterlerini içerir.

Java ve Unicode: Java, Unicode karakterlerini destekler. Java programlama dilindeki String'ler, tanımlayıcılar (identifiers) ve yorumlar (comments) gibi metin verileri için Unicode kullanılır. Bu, Java'nın farklı dillerdeki karakterleri ve sembolleri doğru bir şekilde temsil etmesini sağlar. Örneğin, Java'da Telugu dilindeki karakterler kullanılarak tanımlayıcılar (değişken adları) oluşturulabilir ve Telugu dilinde yorumlar yazılabilir.

Java'nın Unicode'ı desteklemesi, çok dilli uygulamalar geliştirmek için kullanışlıdır. Java, uluslararası yazılım geliştirme için yaygın olarak tercih edilen bir dildir çünkü Unicode desteği sayesinde dünya genelinde kullanılabilir ve çoklu dil desteği sağlar.

1. **Java'da '>>' ve '>>>' operatörleri arasındaki fark nedir?**

'>>' (Sağa Kaydırma Operatörü):

'>>' operatörü, bir değerin tüm bitlerini belirtilen sayıda sağa kaydırmak için kullanılır.

Sağa kaydırma işlemi yapılırken, en sol taraftaki bitin değeri önceki değerine göre korunur ve sağa doğru belirtilen sayıda boşluk bırakılır.

Eğer kaydırma işlemi sonucunda boşalan bitler, kaydırma işlemi yapılan sayının işareti ile doldurulur. Yani, eğer işaretli bir tamsayıyı kaydırıyorsanız, sol taraftaki boşluklar işaret biti (negatiflik) ile doldurulur.

Örnek:

int a = 15;

a = a >> 3; // Sonuç: 1

Bu örnekte, 15 değeri ikili olarak '00001111' temsil edilir ve bu değer 3 kez sağa kaydırıldığında '00000001' elde edilir.

'>>>' (Sağa Kaydırma Operatörü - İşaretsiz):

'>>>' operatörü, sağa kaydırma işlemi yaparken işaretsiz (unsigned) bir kaydırma gerçekleştirir.

Bu operatör ile kaydırma işlemi sonucunda boşalan bitler her zaman sıfır ile doldurulur. Yani, işaret biti (negatiflik) göz ardı edilir.

Örnek:

int a = -15;

a = a >>> 3; // Sonuç: 536870909

Bu örnekte, -15 değeri ikili olarak '11111111111111111111111111110001' temsil edilir. Ancak '>>>' operatörü ile 3 kez sağa kaydırıldığında, işaret biti dikkate alınmaz ve sonuç '00011111111111111111111111111100' olur. Bu nedenle sonuç işaretsiz bir tam sayı olarak 536870909'dur.

'>>' ve '>>>' operatörleri arasındaki temel fark, işaret bitinin (negatiflik) nasıl işlendiği ve kaydırma işlemi sonucunda boşalan bitlerin nasıl doldurulduğudur.

1. **Java kodlama standartları veya Java kodlama kurallarını (Java Coding Conventions) açıklayınız?**

Java kodlama standartları veya Java kodlama kuralları (Java Coding Conventions) genellikle yazılım geliştiricilerin kodlarını daha okunaklı ve tutarlı bir şekilde yazmalarına yardımcı olur. Java sınıfları için kodlama standartlarına dair önemli noktalar:

Sınıf İsimleri Büyük Harfle Başlamalıdır:

Java sınıf isimleri büyük harfle başlamalıdır. Bu, sınıfları diğer öğelerden (değişkenler, metotlar vb.) ayırt etmek için kullanışlıdır.

Örnek: Çalışan veya MüşteriHesabı

Sınıf İsimleri İsimler Olmalıdır:

Sınıf isimleri genellikle isim (noun) olmalıdır ve sınıfın temsil ettiği nesnenin ne olduğunu açıkça yansıtmalıdır.

Örnek: Çalışan sınıfı bir çalışanı temsil ediyorsa, ismi Çalışan olabilir.

Birden Fazla Kelimeden Oluşan Sınıf İsimleri:

Eğer bir sınıf adı birden fazla kelimeden oluşuyorsa, iç kelimelerin ilk harfi büyük olmalıdır. Buna "Camel Case" denir.

Örnek: MüşteriHesabı, KişiProfilBilgisi, Kutuİşlemi

Java kodlama standartları, kodun okunabilirliğini artırmak, yazılım projelerini daha kolay yönetilebilir hale getirmek ve geliştiriciler arasında tutarlılık sağlamak amacıyla kullanılır. Bu kurallara uymak, kodun daha iyi anlaşılmasını ve bakımını kolaylaştırır ve daha profesyonel bir yazılım geliştirme süreci sunar.

1. **Java’da interfaceler için standartlar nelerdir?**

İnterface İsmi Büyük Harfle Başlamalıdır:

Java interface isimleri büyük harfle başlamalıdır. Bu, interfaceleri diğer öğelerden (sınıflar, değişkenler vb.) ayırt etmek için kullanışlıdır.

Örnek: Çalıştırılabilir, Serileştirilebilir

İnterface İsimleri Sıfat Olmalıdır:

İnterface isimleri genellikle sıfat (adjective) olmalıdır ve interface in temsil ettiği yeteneği, özelliği veya davranışı açıkça yansıtmalıdır.

Örnek: ÇoğulUyumluluk interface i, bir sınıfın çoğul uyumlu (Runnable) olduğunu ifade eder. Serileştirilebilir interface i, bir sınıfın serileştirilebilir (Serializable) olduğunu ifade eder. Klonlanabilir interface i, bir sınıfın klonlanabilir (Cloneable) olduğunu ifade eder.

Java'da bu tür kodlama standartları ve adlandırma kuralları, kodun okunabilirliğini artırır ve yazılım projelerini daha anlaşılır ve yönetilebilir hale getirir. Ayrıca, diğer geliştiriciler için daha açık ve tutarlı bir programlama deneyimi sağlar. Bu standartlara uyum, yazılım geliştirme sürecini daha profesyonel hale getirir ve işbirliğini kolaylaştırır.

1. **Java’da metod kullanım standartları nelerdir?**

Metot İsimleri Küçük Harfle Başlamalıdır:

Java metot isimleri küçük harfle başlamalıdır. Bu, metotları diğer öğelerden (sınıflar, arayüzler, değişkenler vb.) ayırt etmek için kullanışlıdır.

Örnek: toString(), hesapla().

Metot İsimleri Genellikle Fiil Olmalıdır:

Metot isimleri genellikle fiil (verb) olmalıdır çünkü metotlar bir eylemi gerçekleştirir.

Örnek: hesapla(), oluştur(), güncelle().

Birden Fazla Kelimeden Oluşan Metot İsimleri:

Eğer bir metot adı birden fazla kelimeden oluşuyorsa, iç kelimelerin ilk harfi büyük olmalıdır. Buna "Camel Case" denir.

Örnek: getArabaAdı(), getArabaNumarası(). Bu şekilde metot adları daha okunaklı ve anlaşılır olur.

Metot İsmi Genellikle Fiil ve İsimin Kombinasyonu Olmalıdır:

Metot ismi, metotun ne yaptığını ve neyi döndürdüğünü açıkça ifade etmelidir. Genellikle bir eylemi (fiil) ve bu eylemin üzerinde çalıştığı nesneyi (isim) içerir.

Örnek: getArabaAdı() metodu, bir arabanın adını döndürür. hesaplaToplam() metodu, bir toplamı hesaplar ve sonucu döndürür.

Java'da bu tür kodlama standartları ve adlandırma kuralları, kodun okunabilirliğini artırır ve yazılım projelerini daha anlaşılır ve yönetilebilir hale getirir. Ayrıca, diğer geliştiriciler için daha açık ve tutarlı bir programlama deneyimi sağlar. Bu standartlara uyum, yazılım geliştirme sürecini daha profesyonel hale getirir ve işbirliğini kolaylaştırır.

1. **Java’da variable(değişken) kullanım standartları nelerdir?**

Değişken İsimleri Küçük Harfle Başlamalıdır:

Java değişken isimleri küçük harfle başlamalıdır. Bu, değişkenleri diğer öğelerden (sınıflar, metotlar, arayüzler vb.) ayırt etmek için kullanışlıdır.

Örnek: string, değer, çalışanAdı, çalışanMaaşı.

Değişken İsimleri Genellikle İsim Olmalıdır:

Değişken isimleri genellikle isim (noun) olmalıdır ve değişkenin temsil ettiği verinin ne olduğunu açıkça yansıtmalıdır.

Örnek: sayı, müşteri, kitapAdı, öğrenciNotları.

Kısa ve Anlamlı İsimler Tavsiye Edilir:

Değişken isimleri kısa olmalıdır, ancak anlamlı olmalıdır. Değişkenin temsil ettiği veriyi açıkça ifade etmelidir.

Örnek: yaş, fiyat, ad, boyut.

Birden Fazla Kelimeden Oluşan Değişken İsimleri:

Eğer bir değişken adı birden fazla kelimeden oluşuyorsa, iç kelimelerin ilk harfi büyük olmalıdır. Buna "Camel Case" denir.

Örnek: çalışanAdı, öğrenciNotları, kitapListesi.

Java'da bu tür kodlama standartları ve adlandırma kuralları, kodun okunabilirliğini artırır ve yazılım projelerini daha anlaşılır ve yönetilebilir hale getirir. Ayrıca, diğer geliştiriciler için daha açık ve tutarlı bir programlama deneyimi sağlar. Bu standartlara uyum, yazılım geliştirme sürecini daha profesyonel hale getirir ve işbirliğini kolaylaştırır.

1. **Java’da sabitler (constant) ile ilgili standartlar nelerdir?**

Sabitler yalnızca büyük harflerle yazılmalıdır.

Örneğin:

public static final int MAX\_VALUE = 100;

public static final String DATABASE\_NAME = "MyDatabase";

Sabit isimleri iki kelimenin birleşiminden oluşuyorsa, bu kelimeler alt çizgi (\_) ile ayrılmalıdır. Bu, sabitlerin okunabilirliğini artırır.

Örneğin:

public static final int MAX\_LENGTH = 255;

public static final String API\_KEY = "my\_api\_key";

Sabit isimleri genellikle isimler veya nesnelerle ilişkilendirilen kelimeler olmalıdır. Bu, sabitin ne tür bir veriyi temsil ettiğini açıkça belirtir.

Örneğin:

public static final int DEFAULT\_TIMEOUT = 5000;

public static final String ERROR\_MESSAGE = "An error occurred.";

Bu kurallar, sabitlerin kodunuzda daha iyi anlaşılabilir ve bakımı daha kolay olmasını sağlar. Ayrıca, sabitlerin değerlerinin değiştirilemez (final) olduğundan emin olur, böylece bu değerlerin yanlışlıkla değiştirilmesini engeller ve kodunuzun güvenilirliğini artırır.

1. **Java'da "IS-A" ilişkisi nedir?**

Java'da "IS-A" ilişkisi, genellikle miras (inheritance) olarak bilinir. "IS-A" ilişkisi, bir sınıfın başka bir sınıftan türetildiği veya kalıtım aldığı durumu ifade eder. Bu türetilen sınıf, türeten sınıfın özelliklerini ve davranışlarını devralır. Java'da "IS-A" ilişkisi, "extends" anahtar kelimesi kullanılarak gerçekleştirilir. Miras alınan sınıf, miras alan sınıfın bir alt sınıfıdır.

Örnek olarak, "Araç" sınıfını ele alalım:

public class Araç {

// Araç sınıfına ait özellikler ve davranışlar burada tanımlanır.

}

Bu durumda, "Araç" sınıfı bir üst sınıftır. Şimdi bu sınıftan türetilen "Motosiklet" ve "Araba" sınıflarını düşünelim:

public class Motosiklet extends Araç {

// Motosiklet sınıfının özellikleri ve davranışları burada tanımlanır.

}

public class Araba extends Araç {

// Araba sınıfının özellikleri ve davranışları burada tanımlanır.

}

Bu durumda, "Motosiklet" ve "Araba" sınıfları "Araç" sınıfından türetilmiştir ve "Araç" sınıfının özelliklerini ve davranışlarını miras almışlardır. Bu, "Motosiklet bir Araç'tır" ve "Araba bir Araç'tır" ifadelerini doğru bir şekilde ifade eder.

Miras (inheritance), kodun yeniden kullanılabilirliğini artırır ve sınıf hiyerarşileri oluşturarak nesne yönelimli programlamada (OOP) önemli bir konsepttir.

1. **Java'da "HAS-A" ilişkisi nedir?**

Java'da "HAS-A" ilişkisi, "kompozisyon" veya "birleştirme (aggregation)" olarak da bilinir. "HAS-A" ilişkisi, bir sınıfın başka bir sınıfı içerdiği veya ona sahip olduğu bir ilişkiyi ifade eder. Bu ilişki, bir sınıfın başka bir sınıfın nesnesini içermesini veya bu nesneye erişim sağlamasını içerir. "HAS-A" ilişkisi, Java'da "extends" anahtar kelimesi gibi özel bir anahtar kelime kullanılmadan gerçekleştirilir.

Örnek olarak, "Araba" sınıfının içinde bir "Motor" sınıfına sahip olduğunu düşünelim:

public class Motor {

// Motor sınıfına ait özellikler ve davranışlar burada tanımlanır.

}

public class Araba {

private Motor motor; // Araba sınıfı, Motor sınıfının bir nesnesini içerir.

public Araba() {

motor = new Motor(); // Araba nesnesi oluşturulduğunda bir Motor nesnesi de oluşturulur.

}

// Araba sınıfına ait diğer özellikler ve davranışlar burada tanımlanır.

}

Bu durumda, "Araba" sınıfı bir "Motor" nesnesini içeriyor. Bu, "Araba HAS-A Motor" ilişkisini ifade eder. "Araba" sınıfı, "Motor" sınıfının özelliklerine ve davranışlarına erişebilir ve onları kullanabilir.

"HAS-A" ilişkisi, kodun yeniden kullanılabilirliğini artırır, çünkü bir sınıfın başka bir sınıfın özelliklerini veya davranışlarını içermesi sayesinde, kod tekrarı önlenir ve kodun daha modüler ve anlaşılır hale gelmesine yardımcı olur. Bu tür ilişkiler, nesne yönelimli programlamanın (OOP) temel kavramlarından biridir.

1. **Java'da "IS-A" ilişkisi ile "HAS-A" ilişkisi arasındaki farklar nelerdir?**

|  |  |
| --- | --- |
| "IS-A" ilişkisi | "HAS-A" ilişkisi |
| Miras (inheritance) ilişkisi olarak bilinir. | Kompozisyon (composition) veya birleştirme (aggregation) ilişkisi olarak bilinir. |
| Bir sınıfın başka bir sınıftan türetildiği veya kalıtım aldığı bir ilişkiyi ifade eder. | Bir sınıfın başka bir sınıfı içerdiği veya ona sahip olduğu bir ilişkiyi ifade eder. |
| "extends" anahtar kelimesi kullanılarak gerçekleştirilir. Türetilen sınıf, türeten sınıfın bir alt sınıfıdır. | Özel bir anahtar kelime kullanılmadan gerçekleştirilir. Sınıf içinde diğer bir sınıfın nesnesi saklanarak veya bu nesneye erişilerek gerçekleştirilir. |
| Örnek: "Araba bir Taşıt'tır." ("Car is a Vehicle.") ifadesi doğru bir IS-A ilişkisini ifade eder. | Örnek: "Araba bir Motor'a sahiptir." ("Car has an Engine.") ifadesi doğru bir HAS-A ilişkisini ifade eder. |
| Mirasın başlıca avantajı kodun yeniden kullanılabilirliğini artırmaktır. | Burada dikkat edilmesi gereken önemli bir nokta, HAS-A ilişkisi ile türetilen sınıfın ayrı bir sınıf olduğudur. Yani Araba sınıfı Motor sınıfının bir alt sınıfı değil, yalnızca bir Motor nesnesine sahiptir. |

1. **Java’da instanceof operatörünü açıklayınız?**

instanceof operatörü, bir nesnenin hangi türde olduğunu test etmek için kullanılır. Bu operatör, bir referans ifadesini ve bir hedef türü alır ve referans ifadesinin belirtilen hedef türün bir alt türü olup olmadığını kontrol eder. instanceof ifadesi doğru (true) veya yanlış (false) bir değer döndürür.

instanceof operatörünün sözdizimi şu şekildedir:

<referans ifadesi> instanceof <hedef tür>

instanceof, aşağıdaki iki sonucu üretebilir:

true: Eğer referans ifadesi, hedef türün bir alt türü ise, instanceof operatörü true değeri döndürür.

false: Eğer referans ifadesi hedef türün bir alt türü değilse veya referans ifadesi null ise, instanceof operatörü false değeri döndürür.

Örnek olarak, aşağıdaki Java kodu instanceof operatörünü kullanır:

public class InstanceOfOrnegi {

public static void main(String[] args) {

Integer a = new Integer(5);

if (a instanceof java.lang.Integer) {

System.out.println(true);

} else {

System.out.println(false);

}

}

}

Bu örnekte, a değişkeni bir Integer türündedir ve a instanceof java.lang.Integer ifadesi true değeri döndürür çünkü a, java.lang.Integer türünün bir örneğidir.

instanceof operatörü, nesnelerin türlerini kontrol etmek, uygun işlemleri gerçekleştirmek ve tip güvenliğini sağlamak için kullanılır. Eğer bir referansın belirli bir türün bir alt türü olup olmadığını kontrol etmeniz gerekiyorsa, instanceof operatörü bu işlemi yapmanıza yardımcı olur.

1. **Java’da null nedir?**

"null" terimi Java'da bir referans değişkeninin hiçbir değeri göstermediğini ifade eder. Yani, bir referans değişkeni tanımlanmıştır, ancak herhangi bir nesneye işaret etmez. Bu, değişkenin bellekte bir konumu olmasına rağmen içeriğinin boş olduğu anlamına gelir.

Örneğin, şu kodda bir Employee (Çalışan) nesnesi oluşturmadan sadece bir referans değişkeni tanımlanmıştır:

Employee employee;

Bu durumda, employee değişkeni null olarak başlatılmıştır çünkü henüz hiçbir çalışan nesnesi oluşturulmamıştır. Bu değişkeni kullanmak istediğinizde, önce bir Employee nesnesi oluşturmanız ve employee değişkenine atamanız gerekecektir. Aksi halde, employee değişkeni hâlâ null değerini tutacaktır.

null değeri, bir referansın hiçbir nesneye işaret etmediğini belirtmek için kullanılır ve Java'da sıklıkla kontrol ifadelerinde ve nesne durumunu belirlemede kullanılır.

1. **Java'da bir kaynak dosyası içinde birden fazla sınıf tanımlanabilir mi?**

Evet, Java'da bir kaynak dosyası içinde birden fazla sınıf tanımlayabilirsiniz, ancak bu nadiren kullanılır ve genellikle tavsiye edilmez. Ancak, bu sınıflardan yalnızca bir tanesi public olarak işaretlenebilir. Başka bir deyişle, kaynak dosyanızda yalnızca bir tane public anahtar kelimesiyle belirtilen sınıf olabilir. Diğer sınıflar public anahtar kelimesiyle belirtilmez.

Eğer aynı kaynak dosyasında birden fazla public sınıf tanımlamaya çalışırsak, aşağıdaki derleme hatasını alırız:

“The public type must be defined in its own file”

"public türü kendi dosyasında tanımlanmalıdır."

Yani, her public sınıfın kendi ayrı kaynak dosyasına sahip olması gerekmektedir. Diğer sınıflar ise public anahtar kelimesi olmadan aynı dosyada tanımlanabilir ve aynı paket içinde kullanılabilir. Bu, sınıfları düzenli ve daha okunaklı tutmak için genellikle daha iyi bir uygulama yöntemidir.

1. **Java'da, en üst düzey sınıflar (top level class), hangi erişim belirleyiciler (access modifier) kullanılabilir?**

Java'da, en üst düzey sınıflar (top level class), yani bir dosyanın ana sınıfları için yalnızca iki erişim belirleyici (access modifier) kullanılabilir: public ve varsayılan (default). İşte bu erişim belirleyicilerin anlamları:

public: Eğer bir sınıf public olarak işaretlenirse, bu sınıf her yerden erişilebilir hale gelir. Başka paketlerdeki sınıflar da bu sınıfa erişebilirler.

Varsayılan (default): Eğer bir sınıf herhangi bir erişim belirleyici belirtilmeden (yani public, private, protected veya package-private olmaksızın) tanımlanırsa, bu sınıf sadece aynı paket içindeki diğer sınıflar tarafından erişilebilir. Başka paketlerdeki sınıflar bu sınıfa erişemezler.

Ancak, bir en üst düzey sınıfı private veya protected gibi diğer erişim belirleyicileri ile işaretlemeye çalışırsanız, aşağıdaki derleme hatasını alırsınız:

Illegal Modifier for the class only public, abstract and final are permitted.

"Sınıf için yalnızca public, abstract ve final erişim belirleyicileri izinlidir."

Bu hatanın anlamı, en üst düzey sınıfların yalnızca public, abstract ve final erişim belirleyicileri ile işaretlenebileceğidir. Bu sınıfları diğer erişim belirleyicileriyle işaretlemek geçerli değildir.

1. **Java’da “package” neyi ifade eder?**

Java'da "package" (paket), ilgili sınıfları, arayüzleri ve enumları tek bir modül içinde gruplamak için kullanılan bir mekanizmadır. Bir paket, Java programınızı düzenlemek ve sınıfları mantıklı bir şekilde gruplandırmak için kullanabileceğiniz bir araçtır. Paketler, kodunuzu daha düzenli, okunaklı ve sürdürülebilir hale getirmenize yardımcı olur.

Paketlerin kullanımı şu şekilde tanımlanır:

Bir paket, package anahtar kelimesi ile tanımlanır, örneğin: package com.example.myapp;

Paket adı küçük harfle yazılmalıdır, bu bir kodlama kuralıdır.

Paketlerin ana amaçları şunlardır:

İsim Çakışmalarını Çözmek: Java'da farklı paketlerde aynı isme sahip sınıflar veya arabirimler olabilir. Paketler, bu tür çakışmaları önlemek veya çözmek için kullanılır. Her paket, kendi isim alanını tanımlar ve bu sayede isim çakışmaları önlenir.

Görünürlük Kontrolü: Paketler aynı zamanda sınıflar ve arabirimlerin başka sınıflar tarafından erişimini kontrol etmek için kullanılabilir. Varsayılan erişim (package-private) olarak işaretlenen sınıflar ve üyeler, yalnızca aynı paket içindeki diğer sınıflardan erişilebilir. Bu, sınıflarınızın daha iyi bir şekilde kapsülasyonunu sağlar ve kodunuzu daha güvenli hale getirir.

Özetlemek gerekirse, Java'da paketler, kodunuzu düzenlemek, isim çakışmalarını önlemek ve sınıflarınızın erişimini kontrol etmek için kullanılan önemli bir araçtır. Bu sayede daha büyük ve karmaşık Java uygulamalarını daha iyi yönetebilirsiniz.

1. **Java'da bir kaynak dosyası içinde birden fazla "package" ifadesi (paket ifadesi) bulunabilir mi?**

Java'da bir kaynak dosyası içinde birden fazla "package" ifadesi (paket ifadesi) bulunamaz. Bir Java programında yalnızca en fazla bir "package" ifadesi bulunabilir. Eğer kaynak dosyanızda birden fazla "package" ifadesi bulunursa, derleme hatası alırsınız. Yani, aynı kaynak dosyasında yalnızca bir kez "package" ifadesi kullanabilirsiniz. İşte bu kuralın Türkçe açıklaması:

Bir Java programında sadece bir "package" ifadesi bulunabilir. Eğer kaynak dosyasında birden fazla "package" ifadesi bulunursa, derleme hatası alırsınız. Yani, bir kaynak dosyası sadece bir paket içinde tanımlanabilir ve bu paket ifadesi dosyanın en başında yer almalıdır. Bu, Java'nın paketleme ve isim alanı yönetimi kurallarına uygunluğu sağlamak için gereklidir.

1. **Java'da import ifadesinden sonra package ifadesini tanımlayabilir miyiz?**

Java dilinde package ifadesi her zaman import ifadesinden önce gelmelidir ve package ifadesi bir Java kaynak dosyasındaki ilk ifade olmalıdır. Yani import ifadesinden önce bir package ifadesi tanımlanamaz. Ancak package ifadesinden önce yorum satırları (comment lines) bulunabilir. Java kaynak dosyalarında yorum satırları herhangi bir yerde bulunabilir ve kod tarafından görmezden gelinirler.

Örnek bir Java kaynak dosyası şu şekilde olabilir:

// Bu bir yorum satırıdır

package com.example.mypackage; // Bu geçerli bir package ifadesidir

import java.util.List;

public class MyClass {

// Sınıfın kodu burada gelir

}

Yukarıdaki örnekte, yorum satırları package ifadesinden önce bulunabilir, ancak package ifadesi import ifadesinden önce gelmelidir. Bu, Java dilindeki sözdizimi kurallarına uyar.

1. **Java’da identifiers lar nelerdir?**

* Tanımlayıcılar harf, alt çizgi (\_) veya dolar işareti ($) ile başlamalıdır.
* Tanımlayıcılar rakamla başlayamaz.
* Tanımlayıcıların uzunluğunda bir sınırlama yoktur, ancak 15 karakterden fazla kullanmak önerilmez.
* Java tanımlayıcıları büyük-küçük harf duyarlıdır. Yani büyük harfle başlayan bir tanımlayıcı ile küçük harfle başlayan bir tanımlayıcı farklı kabul edilir.
* İlk harf bir harf (alfabe karakteri), alt çizgi (\_) veya dolar işareti ($) olabilir. İkinci harften itibaren rakam da kullanabilirsiniz.
* Java'da tanımlayıcılar için ayrılmış (rezerve) kelimeleri kullanmamalıyız. Yani özel anlam taşıyan Java anahtar kelimeleri, sınıf adları, metod adları veya değişken adları olarak kullanılmamalıdır.

Örnek birkaç tanımlayıcı:

Doğru tanımlanmış bir sınıf adı: public class Araba { }

Doğru tanımlanmış bir metod adı: public void hesaplaSonucu() { }

Doğru tanımlanmış bir değişken adı: int ogrenciSayisi = 42;

Bu kurallara uyarak tanımlayıcılarınızı oluşturmalı ve Java kodunuzu daha anlaşılır ve düzenli hale getirmelisiniz.

1. **Java'da erişim değiştiricileri (access modifiers) nelerdir?**

Java'da erişim değiştiricileri (access modifiers), sınıfların, metodların ve üyelerin (değişkenlerin veya alanların) erişim seviyelerini kontrol etmek için kullanılan önemli özelliklerden biridir. Erişim kontrolü, bir sınıfın, metodun veya üyenin nasıl kullanılabileceğini belirler ve bu sayede yanlış kullanımların önüne geçilir.

Java'da üç tür erişim değiştirici bulunur:

public: Bu değiştirici, bir sınıfın, metodun veya üyenin herhangi bir yerden (başka bir sınıf, paket veya proje içinden) erişilebilir olduğunu belirtir. Yani herkes bu öğeye erişebilir.

private: Bu değiştirici, bir sınıfın, metodun veya üyenin sadece kendi sınıfı içinden erişilebilir olduğunu belirtir. Başka sınıflar bu öğeye erişemez.

protected: Bu değiştirici, bir sınıfın, metodun veya üyenin aynı paket içinde veya alt sınıflardan erişilebilir olduğunu belirtir. Diğer paketlerden gelen sınıfların erişimi sınırlanır.

Ayrıca, bir erişim değiştirici belirtilmezse, varsayılan bir erişim düzeyi kullanılır. Varsayılan erişim, sadece aynı paketten erişimi sağlar. Başka bir paketten gelen sınıflar bu öğeye erişemezler.

Özetle, erişim değiştiricileri Java'da sınıfların, metodların ve üyelerin kimler tarafından erişebileceğini kontrol etmek için kullanılır ve bu, kodun daha güvenli ve düzenli olmasına yardımcı olur.

1. **Java'da sınıflar için kullanılabilecek erişim değiştiricileri (access modifiers) nelerdir?**

* public: Bir sınıfın erişim düzeyi "public" olarak belirtilirse, bu sınıf her yerden erişilebilir hale gelir. Yani, bu sınıfın neredeyse her yerden erişilebilir olur. İşte erişim seviyesi "public" olan bir sınıfın hangi yerlerden erişilebileceği:

Aynı sınıf içinde (kendi sınıfının içinde).

Aynı paket içinde yer alan başka sınıflar ve alt sınıflar tarafından erişilebilir.

Aynı paket içinde yer alan başka sınıflar ve alt sınıflar olmayan sınıflar tarafından erişilebilir.

Farklı bir paket içinde yer alan alt sınıflar tarafından erişilebilir.

Farklı bir paket içinde yer alan alt sınıflar olmayan sınıflar tarafından erişilebilir.

* Varsayılan (Default): Erişim değiştirici belirtilmediğinde, sınıfın erişim düzeyi "varsayılan" olarak kabul edilir. Varsayılan erişim, yalnızca aynı paket içinde bulunan sınıflar tarafından erişilebilir. Bu, paket düzeyinde erişim sağlar. Yani, sınıfın aynı paket içinde yer alan diğer sınıflar tarafından erişilebilir olduğu anlamına gelir.

Özetle, Java'da sınıflar için iki tür erişim değiştirici kullanılabilir: "public" ve "varsayılan" (default). Bu değiştiriciler, sınıfların hangi yerlerden erişilebileceğini belirler ve kodun kontrol edilmesine yardımcı olur.

1. **Java'da, metodlar için kullanılabilecek erişim değiştiricileri (access modifiers) nelerdir?**

* public: Bir metodun erişim düzeyi "public" olarak belirtilirse, bu metod her yerden erişilebilir hale gelir. Yani, bu metodun neredeyse her yerden erişilebilir olur. İşte erişim seviyesi "public" olan bir metodun hangi yerlerden erişilebileceği:

Aynı sınıf içinde (kendi sınıfının içinde).

Aynı paket içinde yer alan başka sınıflar ve alt sınıflar tarafından erişilebilir.

Aynı paket içinde yer alan başka sınıflar ve alt sınıflar olmayan sınıflar tarafından erişilebilir.

Farklı bir paket içinde yer alan alt sınıflar tarafından erişilebilir.

Farklı bir paket içinde yer alan alt sınıflar olmayan sınıflar tarafından erişilebilir.

* Varsayılan (Default): Erişim değiştirici belirtilmediğinde, bir metodun erişim düzeyi "varsayılan" olarak kabul edilir. Varsayılan erişim, yalnızca aynı paket içinde bulunan sınıflar ve alt sınıflar tarafından erişilebilir. Bu, paket düzeyinde erişim sağlar. Yani, metodun aynı paket içinde yer alan diğer sınıflar ve alt sınıflar tarafından erişilebilir olduğu anlamına gelir.
* protected: Bir metodun erişim düzeyi "protected" olarak belirtilirse, bu metod aynı sınıf, aynı paket içindeki sınıflar, alt sınıflar ve farklı paket içindeki alt sınıflar tarafından erişilebilir. Ancak, farklı bir paket içinde yer alan non-subclass (alt sınıf olmayan) sınıflar bu metoda erişemez.
* private: Bir metodun erişim düzeyi "private" olarak belirtilirse, bu metod sadece kendi sınıfı içinde erişilebilir hale gelir. Yani, başka sınıfların veya alt sınıfların bu metoda erişmesine izin verilmez.

Bu erişim değiştiricileri, metodların hangi yerlerden erişilebileceğini ve kimlerin bu metodları kullanabileceğini belirlemek için kullanılır. Bu, kodun güvenliğini ve düzenini sağlamak için önemlidir.

1. **Java’da final access modifier nedir?**

Final Class (Final Sınıf):

Bir sınıfın "final" olarak işaretlenmesi, bu sınıfın başka bir sınıf tarafından alt sınıf olarak genişletilmesini veya miras alınmasını önler.

Final bir sınıfın avantajı, güvenliği artırmaktır, yani bu sınıfın davranışı değiştirilemez.

Dezavantajı ise Java'nın nesne yönelimli programlama (OOP) kavramlarını kullanırken bu sınıfın miras alınamayacak olmasıdır.

Örnek:

final class FinalClass {

// ...

}

Final Method (Final Metod):

Bir metodu "final" olarak işaretlemek, bu metodun alt sınıflar tarafından ezilememesini sağlar. Yani bu metodun davranışı değiştirilemez.

Metodların override edilmesini istemediğiniz durumlarda final metotlar kullanışlıdır.

Örnek:

class BaseClass {

public final void finalMethod() {

// ...

}

}

Final Variable (Final Değişken):

Bir değişkeni "final" olarak işaretlemek, bu değişkenin bir kez değer atanabilen ve sonrasında değiştirilemeyen bir sabit gibi davranmasını sağlar.

Herhangi bir deneme değişkenin değerini değiştirmeye çalışmak derleme hatası verir.

Örnek:

public class FinalVariableExample {

public static final int MY\_CONSTANT = 42;

}

Bu final erişim belirleyicileri, kodunuzun güvenliğini artırmak ve belirli durumlarda OOP kavramlarına (örneğin miras) karşı koruma sağlamak için kullanışlıdır. Ancak, bu belirleyicileri dikkatli bir şekilde kullanmalısınız, çünkü aşırı kullanıldığında kodunuzu esnekliğini kaybetmesine neden olabilirler.

1. **Java’da Abstract Classı açıklayınız?**

Java'da soyut sınıflar (abstract classes), bazı durumlarda bir sınıftaki tüm metodlara uygulama sağlayamayabileceğimiz durumlarla karşılaştığımızda kullanılırlar. Bu durumda, ilgili metodların uygulamasını, bu soyut sınıfı genişleten bir başka sınıfa bırakmak isteriz. Bir sınıfı soyut yapmak için "abstract" anahtar kelimesini kullanırız. Bir veya daha fazla soyut metod içeren her sınıf soyut olarak işaretlenir. Eğer soyut metotlar içeren bir sınıfı soyut olarak işaretlemezsek, derleme zamanında hata alırız ve şu hata mesajını alırız:

"Soyut metodları tanımlamak için <sınıf adı> sınıfı soyut bir sınıf olmalıdır."

Örnek olarak, bir araç sınıfını ele alalım. Bu sınıfa uygulama sağlayamayabiliriz çünkü iki tekerlekli, dört tekerlekli vb. gibi farklı araç türleri olabilir. Bu durumda, araç sınıfını soyut yaparız. Araç sınıfında araçların ortak özellikleri soyut metodlar olarak tanımlanır. Araç sınıfını genişleten herhangi bir sınıf, bu metodları kendi uygulayacaktır. Alt sınıfın, bu soyut metodları uygulamakla yükümlüdür.

Soyut sınıfların önemli özellikleri şunlardır:

* Soyut sınıfların örnekleri oluşturulamaz.
* Soyut sınıflar, soyut metotlar, somut metotlar veya her ikisini içerebilir.
* Bir soyut sınıfı genişleten her sınıf, soyut sınıfın tüm metodlarını geçersiz kılmak (override) zorundadır.
* Bir soyut sınıf 0 veya daha fazla soyut metot içerebilir.

Örnek bir soyut sınıf:

abstract class Shape {

// Soyut metotlar

public abstract double getArea();

public abstract double getPerimeter();

// Somut metot

public void displayInfo() {

System.out.println("This is a shape.");

}

}

Soyut sınıflar, nesne yönelimli programlama (OOP) kavramlarını uygularken çok kullanışlıdır çünkü alt sınıfların belirli davranışları uygulamalarına olanak tanır.

1. **Abstract class larda constructor oluşturabilir miyiz?**

Evet, soyut sınıflarda da constructor oluşturabiliriz. Derleme hatası vermez. Ancak soyut sınıfların kendileri doğrudan örneklendirilemeyeceği (yani instantiate edilemeyeceği) için soyut bir sınıfın kendi constructor metodu, soyut sınıfın alt sınıfları için kullanılacaktır. Soyut sınıfın alt sınıfları bu constructor metodu çağırdığında, alt sınıfların başlatılmasına yardımcı olur.

Soyut sınıfların constructor metotları, alt sınıfların başlatılmasını ve inşa edilmesini sağlamak amacıyla kullanışlıdır. Bu constructor metotlar, alt sınıfın kendisine özgü inşa sürecini başlatmak için soyut sınıfın özelliklerine erişebilir.

Örneğin, aşağıdaki Java kodunda soyut bir sınıfın constructor metodu örneklendirilemeyen bir soyut sınıfın alt sınıfı tarafından çağrılmaktadır:

abstract class Animal {

private String name;

public Animal(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

class Dog extends Animal {

public Dog(String name) {

super(name); // Soyut sınıfın kurucu metodu çağrılıyor

}

// Dog sınıfının kendi davranışları burada tanımlanabilir.

}

Soyut sınıfın constructor metodu, alt sınıfın başlangıç durumunu ayarlamak için kullanıldığı için soyut sınıfların constructor metotları önemli bir işleve sahip olabilir. Ancak doğrudan bir soyut sınıf örneği oluşturamazsınız. Soyut sınıfın alt sınıflarını oluşturarak bu sınıfları kullanabilirsiniz.

1. **Java’da abstract methodlar nelerdir?**

Java'da soyut metodlar (abstract methods), herhangi bir işlem gövdesine sahip olmayan metodlardır. Soyut metodlar, "abstract" anahtar kelimesi kullanılarak ve metodun gövdesi yerine noktalı virgül (;) ile tanımlanır. Bir soyut metodun imzası şu şekildedir:

public abstract void <metod adı>();

Örnek olarak:

public abstract void getDetails();

Soyut metodların en önemli özelliği, soyut sınıflarda tanımlanmış olmalarıdır. Bir soyut sınıfta tanımlanan soyut metodlar, bu sınıfı genişleten alt sınıflar tarafından uygulama sağlanması gereken metotlardır. Soyut sınıfta tanımlanan soyut metodlar, alt sınıfların bu metodları kendi özel ihtiyaçlarına ve mantığına göre uygulamalarına olanak tanır.

Soyut metodlar, soyut sınıfın kalıtım yoluyla türetilen alt sınıflarının davranışını belirlemek için kullanılır ve bu şekilde soyut sınıfın davranışının şekillenmesine katkıda bulunurlar. Soyut sınıflar ve soyut metodlar, nesne yönelimli programlama (OOP) konseptlerinin uygulanmasına yardımcı olur.

1. **Java’da exception nedir?**

Java'da exception bir nesnedir. Exception, programımızda anormal durumlar ortaya çıktığında oluşturulurlar. Bu anormal durumlar, JVM (Java Sanal Makinesi) veya uygulama kodumuz tarafından oluşturulabilir. Tüm exception sınıfları "java.lang" paketi içinde tanımlanır. Başka bir deyişle, exception ları çalışma zamanı hataları olarak düşünebiliriz.

Exception, bir programın düzgün çalışmasını engelleyebilecek veya beklenmeyen bir durumu belirlemek için kullanılır. Örneğin, bir dosya okuma hatası, bir diziye erişim hatası veya sıfıra bölme hatası gibi durumlar istisnalara örnektir. Exception, bu tür hataların işlenmesine ve programın çalışmaya devam etmesine olanak tanır. Java'da, istisna işleme için try, catch, ve throw gibi anahtar kelimeler kullanılır.

1. **Java’da exceptionların çıkabileceği bazı durumları söyleyiniz?**

Java'da exceptionlar, programın normal akışının dışında ortaya çıkan ve işlenmesi gereken hatalı durumları temsil eder. İşte Java'da exceptionların ortaya çıkabileceği bazı durumlar ve bu durumların kısa açıklamaları:

Dizide Var Olmayan Bir Elemana Erişme: Bir dizinin belirli bir dizini veya elemanı bulunmuyorsa ve bu elemana erişmeye çalışırsanız**, ArrayIndexOutOfBoundsException** ortaya çıkabilir.

Sayıyı Dizeye veya Dizeden Sayıya Hatalı Dönüştürme: Bir sayıyı dize olarak temsil eden bir metin veya tam tersini dönüştürmeye çalışırken hatalı bir biçim kullanırsanız, **NumberFormatException** ortaya çıkabilir.

Geçersiz Sınıf Türü Dönüşümü: Bir sınıfın bir nesnesini başka bir sınıf türüne dönüştürmeye çalışırken uyumsuz bir tür dönüşüm yapmaya çalışırsanız, **ClassCastException** ortaya çıkabilir.

Arayüz veya Soyut Sınıf İçin Nesne Oluşturma: Java'da doğrudan bir arayüz veya soyut sınıf için bir nesne oluşturulamaz. Bu durumda, **InstantiationException** ortaya çıkabilir.

Bu exceptionlar, Java programlarının çalışma zamanında hatalarla başa çıkmasına yardımcı olur. Exception işleme mekanizması, bu tür hataları ele almak için try-catch blokları veya exceptionın yukarı yönlendirilmesini (throwing) içerir. Java programcıları, bu tür exceptionları tanımlayabilir, işleyebilir ve uygun şekilde yanıt verebilir.

1. **Java’da exception handle etme nedir?**

Java'da exception handling, bir programın çalışması sırasında normalden sapma veya hata durumlarının nasıl ele alınacağını belirleyen bir mekanizmadır. Programın normal akışı, bir hata durumu ortaya çıktığında kesintiye uğramamalıdır.

Programların çalışması sırasında birçok hata veya özel durum (exception) ortaya çıkabilir. Bu exceptionlar, veritabanına bağlanma hatası, dosya bulunamama hatası, dizi sınırının aşılması, matematiksel işlemlerde bölme sıfıra bölme hatası gibi çeşitli nedenlerle ortaya çıkabilir. Bu tür exceptionların programın çalışmasını durdurmasını veya çökmesini önlemek için exception handling kullanılır.

Exception işleme, programcılara hata durumlarını tanımlama, ele alma ve uygun bir şekilde yanıtlama olanağı sunar. Bu, programın daha güvenli ve daha düzenli bir şekilde çalışmasını sağlar. Java'da exception handling, try, catch, finally, ve throw anahtar kelimeleri gibi özgül yapılar kullanılarak gerçekleştirilir.

Örneğin, bir dosyanın okunmaya çalışıldığı bir programda, dosya bulunamazsa veya okuma hatası olursa, bu tür exceptionları yakalayarak programın düzgün bir şekilde çalışmasını sağlamak için exception handling kullanılabilir.

Java'da exception handling, programlarınızın daha güvenilir ve kullanıcı dostu olmasına yardımcı olur ve istenmeyen çökme durumlarını önler.

1. **Java’da error nedir?**

Java'da "hata" (error), Throwable sınıfının bir alt sınıfıdır. Hatalar, programın normal akışını durduran ve genellikle geri dönülemez olan ciddi sorunları temsil eder. Hataların programın akışını durdurmasının nedeni, genellikle bu hataların programın kontrolünün dışında olması veya programın düzeltemeyeceği kritik sorunlar olmasıdır. Hatalar programın çalışmasını sonlandırabilir.

Hatalar genellikle çeşitli çevresel faktörler veya sistem sorunları nedeniyle ortaya çıkar. Örnek olarak, bellek tükenmesi hatası (OutOfMemoryError), programın kullanılabilir belleği aşması nedeniyle ortaya çıkar. Bu tür hataların program tarafından düzeltilmesi veya ele alınması mümkün değildir çünkü programın kontrolü dışındaki faktörlerden kaynaklanır.

Bu nedenle, hatalar genellikle programlar tarafından ele alınmaz ve programın çalışmasını sonlandırır. İstisnalar (exceptions) genellikle program tarafından ele alınabilen ve düzeltilebilen hata durumlarını temsil ederken, hatalar genellikle programın kontrolünün dışında olan ve müdahale edilemeyen sorunları ifade eder. Özetle, hatalar programın çalışmasını sonlandırır ve çoğunlukla programcılar tarafından düzeltilmesi veya ele alınması gerekmeyen sorunlardır.

1. **Java’da exceptionı handle etmenin avantajları nelerdir?**

Java'da exception handle etmenin avantajları şunlar olabilir:

Normal Kodu İstisna İşleme Kodundan Ayırma: Java'da exception handle etme, programın normal akışını, hata durumlarını ele almak için ayrılmış kod bloklarından ayırmanıza olanak tanır. Bu, hata durumlarında programın ani ve abur-cubur şekilde sonlandırılmasını önler. Programın normal işlevselliği, istisna durumlarının kontrolünde etkilenmez.

Exception Türlerini Kategorize Etme: Java'da farklı türde exception durumları için özel exception sınıfları bulunur. Bu, programcıların her exception türüne özgü hata işleme kodları yazmasını sağlar. Bu, hataları daha spesifik ve anlamlı bir şekilde ele almayı mümkün kılar. Örneğin, bir dosya bulunamadığında "FileNotFoundException" gibi belirli bir exception türünü ele almak, daha açıklayıcı ve düzgün hata işleme sağlar.

Çağrı Yığını Mekanizması(Call stack mechanism): Bir metot bir exception fırlattığında ve bu exception hemen ele alınmazsa, exception çağrıldığı metodu çağıran yere (çağrı yığını) geri gönderilir. Bu, exception handle etme hiyerarşisini ve işlemeyi kolaylaştırır. Exception bir uygun işleyici buluncaya kadar yukarı doğru taşınır. Bu, programın exceptionları düzgün bir şekilde ele almasını sağlar ve programın ani bir şekilde sonlandırılmasını önler.

Bu avantajlar, Java'da exception handle etmenin programların daha güvenli, düzenli ve hata durumlarına karşı daha dirençli hale gelmesine yardımcı olduğu nedenleri içerir. Exception handle etme, hata durumlarını açıkça tanımlayabilir, izole edebilir ve uygun bir şekilde ele alabilir. Bu da yazılımın daha güvenilir ve bakımı daha kolay olmasını sağlar.

1. **Java’da kaç yolla exeptionları handle edebiliriz?**

Java'da exception handling yapmanın iki ana yolu vardır:

try-catch Bloğu Kullanarak exception handling: Bir try-catch bloğu, belirli bir kod bloğunu çevreler ve bu kod bloğunda ortaya çıkabilecek exceptionları yakalamak ve işlemek için kullanılır. İşte bu yöntemin kullanımı:

try: Exception fırlatabilecek kodun bu bloğun içine yerleştirilir.

catch: try bloğunda ortaya çıkan belirli bir exception türünü yakalamak ve işlemek için kullanılır.

Örnek bir try-catch bloğu:

try {

// İstisna fırlatabilecek kod

} catch (ExceptionType e) {

// İstisna yakalandığında yapılacak işlemler

}

try-catch bloğu, belirli bir exception türünü belirterek exceptionların yakalanmasını ve uygun bir şekilde işlenmesini sağlar.

throws Bildirimi Kullanarak exception handling: Bir metot, içinde exception fırlatabileceğini belirten bir throws bildirimi kullanabilir. Bu yöntem, bir metotun içindeki kod, exception fırlatabilir, ancak exceptionları yakalamak veya işlemek yerine, bu metodu çağıran kodun sorumlu olmasını bekler.

Örnek bir throws bildirimi:

public void someMethod() throws SomeException {

// İstisna fırlatabilecek kod

}

Bu yaklaşım, exception handlingi metot seviyesinde taşır ve exception yakalama sorumluluğunu çağıran kodun üzerine bırakır.

Her iki yaklaşımın da kendine özgü kullanım alanları vardır, ve hangi yöntemin tercih edileceği, işlemek istenen exception türüne, programın tasarımına ve gereksinimlerine bağlı olarak değişebilir.

1. **Java’da try catch bloğunu açıklayınız?**

Java'da "try" ve "catch" anahtar kelimeleri, exeption durumlarının ele alınması için kullanılır.

"try" Bloğu:

"try" bloğu içinde, exception oluşturabilen veya fırlatabilecek kodları tanımlarız. Exception meydana geldiğinde bu blok çalışır ve kodunuzu değerlendirmeye alır. Eğer exception oluşmazsa, "try" bloğu normal bir şekilde çalışır.

Örnek "try" blok kullanımı:

try {

// İstisna oluşturabilen kodlar burada bulunur.

} catch (Exception e) {

// İstisna yakalandığında buradaki kod çalışır.

}

"catch" Bloğu:

"catch" bloğu, bir "try" bloğunda meydana gelen exception durumlarını yakalar. Exception durumu "try" bloğundan "catch" bloğuna iletilir ve burada işlenir. "catch" bloğu, exception türüne göre çalışacak kodu içerir. Böylece, exception oluştuğunda hangi işlemin yapılacağını belirleriz.

Örnek "catch" blok kullanımı:

try {

// İstisna oluşturabilen kodlar burada bulunur.

} catch (Exception e) {

// İstisna yakalandığında buradaki kod çalışır.

// "e" değişkeni, yakalanan istisnayı temsil eder.

// Burada istisnayı işleyebilir veya raporlayabiliriz.

}

Java'da bir "try" bloğu, bir "catch" bloğu ile tamamlanır ve bu ikisi birlikte bir exception işlem birimi oluşturur. Bir "catch" bloğu, yalnızca kendisinden önceki "try" bloğunda meydana gelen exceptionları yakalayabilir. Başka bir "try" bloğunun oluşturduğu exception, bir "catch" bloğu tarafından yakalanamaz.

Eğer programınızda exception oluşturan kod yoksa veya exception oluşturulmazsa, JVM (Java Sanal Makinesi) "try-catch" bloğunu görmezden gelir ve normal program akışı devam eder.

1. **Java’da catch bloğu kullanmadan try bloğu kullanılır mı?**

Bir "try" bloğunu, Java'da exception bir durumu ele almak veya kaynakları temizlemek için kullanabilirsiniz. Ancak bu "try" bloğunun yalnızca kendisi olmaz; ya bir "catch" bloğu ile hata işleme veya bir "finally" bloğu ile kaynak temizleme gibi bir sonuç sağlamalıdır.

Yani, bir "try" bloğu oluşturuyorsanız, aynı zamanda en az bir "catch" bloğu veya "finally" bloğu eklemeniz gerekmektedir. Sadece "try" bloğu kullanmak veya her ikisini birden atlamak (ne "catch" ne de "finally") derleme hatasına yol açar. Bu nedenle, exception durumları ele almak veya kaynakları düzgün şekilde temizlemek için "try-catch" veya "try-finally" bloklarını kullanmak önemlidir.

1. **Java'da bir "try" bloğu için birden fazla "catch" bloğu kullanılabilir mi?**

Bir "try" bloğu içinde birden fazla "catch" bloğu kullanılabilir çünkü kodunuz birden çok farklı türde exception oluşturabilir. Her "catch" bloğu, belirli bir exception türünü ele alır ve ilgili işlemi gerçekleştirir. Bu şekilde, programınızın farklı türde hatalarla başa çıkabilmesini sağlayabilirsiniz.

Örnek bir "try" bloğu ile birden fazla "catch" bloğu kullanımı:

try {

// İstisna oluşturabilen kodlar burada bulunur.

} catch (ArithmeticException e) {

// ArithmeticException türündeki istisnaları ele al

// Bu blok, bölen sıfır hatası gibi matematiksel hataları işleyebilir.

} catch (NullPointerException e) {

// NullPointerException türündeki istisnaları ele al

// Bu blok, bir nesnenin olmadığı durumları işleyebilir.

} catch (IOException e) {

// IOException türündeki istisnaları ele al

// Bu blok, dosya okuma/yazma hatalarını işleyebilir.

} catch (Exception e) {

// Genel olarak tüm istisna türlerini ele al

// Bu blok, yukarıdaki catch bloklarının yakalayamadığı istisnaları işleyebilir.

}

Bu "catch" blokları, sırasıyla çalışır. Yani, bir exception meydana geldiğinde, JVM her bir "catch" bloğunu sırayla kontrol eder ve ilk eşleşen "catch" bloğunu çalıştırır. Geri kalan "catch" blokları atlanır. Bu nedenle, "catch" bloklarının sırası çok önemlidir.

Ayrıca, "catch" bloklarının sırasının alt sınıf exceptionlarından üst sınıf exceptionlarına doğru olması gerekir. Yani, daha özel exception türleri daha önce gelmelidir, sonra daha genel olanlar sıralanmalıdır. Bu, herhangi bir "catch" bloğunun çalıştırılmasını sağlar ve genel işlem sırasında daha özel durumları ele almanıza olanak tanır. Bu nedenle, "try" bloğu ile birden fazla "catch" bloğu kullanırken bu sıralamayı dikkate almak önemlidir.

1. **Java’da finally bloğunun önemini açıklayınız.**

"finally" bloğu, Java'da önemli bir rol oynar çünkü şu amaçlar için kullanılır:

Kaynak Temizleme: Genellikle "try" bloğu içinde kaynakların (veritabanı bağlantıları, dosya işlemleri, soketler vb.) kullanıldığı durumlarda, "finally" bloğu bu kaynakların temizlenmesi için kullanılır. Bu, kaynakların güvenli bir şekilde kapatılmasını ve kaynak sızıntılarının önlenmesini sağlar.

Hata Durumlarında İşlem Tamamlama: Eğer "try" bloğu exception oluşturmadan başarıyla çalışırsa, "finally" bloğu "catch" bloğunu atlamadan hemen sonra çalışır. Bu, temizleme veya sonlandırma işlemlerini gerçekleştirmeniz gereken durumlar için kullanışlıdır.

İstisna Durumlarında İşlem Tamamlama: Eğer "try" bloğunda bir exception oluşursa ve bu exception bir "catch" bloğu tarafından ele alınmazsa, yine de "finally" bloğu çalıştırılır. Bu, beklenmeyen hatalar durumunda kaynakları serbest bırakmanız veya temizlemeniz gerektiğinde kullanışlıdır.

"finally" bloğu, "try-catch-finally" yapısı içinde bulunur ve bu yapı, exceptionların düzgün bir şekilde ele alınması ve kaynakların güvenli bir şekilde temizlenmesi için çok önemlidir. Bu sayede, programlarınız daha güvenli ve sağlam hale gelir ve kaynak yönetimi konusundaki hatalar minimize edilir. Bu nedenle, "finally" bloğu Java'da önemli bir yapısal öğedir ve programların daha güvenilir olmasına yardımcı olur.

1. **Java'da "try" ve "catch" blokları arasında herhangi bir kod yazılabilir mi?**

"try" bloğu ve "catch" bloğu arasında herhangi bir kod bulunmamalıdır. "try" bloğu içindeki kod çalıştırılır ve bir exception oluşursa, bu exception "catch" bloğuna yönlendirilir. Eğer "try" bloğu ve "catch" bloğu arasında kod bulunursa, derleme hatası alırsınız. İstisna oluştuğunda, bu kod parçası atlanır ve "catch" bloğu çalıştırılmadan hemen sonra exception işlenir.

Doğru kullanım şu şekildedir:

try {

// try bloğunda istisna oluşturabilen kodlar burada bulunur

} catch (Exception e) {

// istisna yakalandığında bu blok çalışır

}

Yani, "try" bloğu ve "catch" bloğu birbirine bitişik olmalı ve aralarına herhangi bir kod parçası eklenmemelidir. Bu kurala uymak, kodunuzun beklenmedik hatalara karşı daha güvenli ve düzenli olmasına yardımcı olur.

1. **Java'da "try" ve "finally" blokları arasında herhangi bir kod yazılabilir mi?**

Java'da "try" bloğu ve "finally" bloğu arasında herhangi bir kod parçası yer alamaz. "finally" bloğu, "try" ve (varsa) "catch" blokları tamamlandıktan hemen sonra çalışır. Eğer "catch" bloğu yoksa, "try" bloğu tamamlandıktan hemen sonra "finally" bloğu çalışır.

Doğru kullanım şu şekildedir:

try {

// try bloğunda işlem yapılacak kodlar burada bulunur

} catch (Exception e) {

// istisna yakalandığında bu blok çalışır

} finally {

// finally bloğu, try ve catch tamamlandıktan sonra çalışır

}

1. **Java'da aynı "catch" bloğunda birden fazla exception türü ele alınır mı?**

Java 7'den itibaren, aynı "catch" bloğunda birden fazla istisna türünü ele alabiliriz. Örneğin, aşağıdaki gibi bir yaklaşım kullanabiliriz:

try {

// İstisna oluşturabilen kodlar burada bulunur

} catch (ArrayIndexOutOfBoundsException | ArithmeticException e) {

// Birden fazla istisna türünü aynı "catch" bloğunda ele alabiliriz

// e değişkeni, her iki istisna türünün de istisnasını temsil eder

// Bu blok, hem ArrayIndexOutOfBoundsException hem de ArithmeticException istisnalarını işleyebilir.

}

Ancak, aynı "catch" bloğunda birden fazla exception türünü ele alırken dikkate almanız gereken bazı önemli noktalar şunlardır:

"catch" parametresi zaten final olarak kabul edilir. Yani, bu değişkene başka bir değer atayamazsınız veya değiştiremezsiniz.

Aynı "catch" bloğunda ele alınan exception türleri arasında bir hiyerarşi olmalıdır. Yani, bir üst sınıfın exceptionsını bir alt sınıfın exceptionsı olarak ele almak mümkün değildir. Örneğin, "catch (Exception e)" ve "catch (RuntimeException e)" aynı "catch" bloğunda kullanılamaz.

Bu tür bir yaklaşım, kodunuzu daha temiz ve daha az tekrarlı hale getirebilir, ancak exception durumları mantıklı bir şekilde ele almak ve kodunuzun okunabilirliğini korumak için dikkatli olmalısınız.

1. **Checked Exceptions nelerdir?**

Kontrol Edilen Exceptionlar:

* "Throwable" sınıfının alt sınıfları arasında, "error", "RuntimeException" ve onun alt sınıfları haricinde kalan tüm exception türleri "kontrol edilen exception" olarak kabul edilir.
* Kontrol edilen exception türleri, programcıların bu exceptionları işlemek veya bunları bildirmek için özel bir işlem yapmalarını gerektirir.
* Bu exception türlerini işlemek için "throws" anahtar kelimesi ile metot imzasında veya "try-catch" blokları kullanılmalıdır, aksi takdirde derleme hatası alınır.

Örnekler:

IOException: Dosya giriş/çıkış işlemleri sırasında oluşan hataları temsil eder.

SQLException: Veritabanı işlemleri sırasında oluşan hataları temsil eder.

FileNotFoundException: Belirtilen dosya bulunamadığında oluşan exceptionı temsil eder.

InvocationTargetException: Bir yordamın yürütülmesi sırasında exception oluştuğunda bu tür bir exception fırlatılır.

CloneNotSupportedException: Bir nesnenin klonlanmaya çalışıldığında ve klonlama işlemi desteklenmiyorsa oluşan exceptionı temsil eder.

ClassNotFoundException: Bir sınıfın yüklenmeye çalışıldığında ve bu sınıf mevcut değilse fırlatılan exceptionı temsil eder.

InstantiationException: Bir sınıfın bir örneği oluşturulamadığında fırlatılan exceptionı temsil eder.

Bu exception türleri, programın güvenliğini ve istikrarını sağlamak için önemlidir ve programcılar bu tür exceptionları yakalamak ve işlemek için gerekli önlemleri almalıdır.

1. **Unchecked Exceptions nelerdir?**

Kontrol Edilmemiş İstisnalar (Unchecked Exceptions):

"RuntimeException" sınıfının alt sınıfları, "unchecked exceptions" olarak adlandırılır. Bu tür exception türleri, programın derleyici tarafından exception yönetimi için kontrol edilmesini gerektirmez.

Programlar, bu tür exceptionları ele almasalar bile derlenebilir. Bu, "catch" veya "throws" kullanımına gerek olmadığı anlamına gelir.

Eğer bir kontrol edilmemiş exception meydana gelirse, programın çalışması sona erir. Bu nedenle bu tür exceptionların dikkatli bir şekilde ele alınması önemlidir.

Örnekler:

ArithmeticException: Matematiksel işlemlerde sıfıra bölme veya geçersiz işlemler nedeniyle oluşan exceptionları temsil eder.

ArrayIndexOutOfBoundsException: Dizi erişimi sırasında dizinin sınırlarının dışına çıkma exceptionlarını temsil eder.

ClassCastException: Nesne tür dönüşümü sırasında uyumsuz türler arasında dönüşüm yapma girişimleri nedeniyle oluşan exceptionları temsil eder.

IndexOutOfBoundsException: İndeks sınırları dışına çıkma exceptionlarını temsil eder.

NullPointerException: Null bir nesneye erişme girişimleri nedeniyle oluşan exceptionları temsil eder.

NumberFormatException: Bir sayısal dönüşüm hatası nedeniyle oluşan exceptionları temsil eder.

StringIndexOutOfBounds: Bir dize içinde belirtilen indeksin sınırları dışına çıkma exceptionlarını temsil eder.

UnsupportedOperationException: Bir koleksiyon üzerinde desteklenmeyen bir işlemi yapma girişimleri nedeniyle oluşan exceptionları temsil eder.

Bu tür exceptionlar, programların güvenliğini ve istikrarını sağlamak için önemlidir ve programcılar bu tür exceptionları öngörmeye ve ele almaya çalışmalıdır.

1. **Java’da varsayılan handle etme işlemi nedir?**

Java'da varsayılan exception işleme, Java Sanal Makinesi (JVM) tarafından otomatik olarak yürütülen bir mekanizmadır. Bir exception oluşturulduğunda veya tespit edildiğinde, JVM aşağıdaki bilgileri içeren yeni bir exception işleme nesnesi oluşturur:

Exception Türü Adı

Exception Hakkında Açıklama

Exception Oluşma Konumu

JVM, bu exception işleme nesnesini oluşturduktan sonra, programın bu exception ile başa çıkıp çıkamayacağını kontrol eder. Eğer programda exception işleme kodu varsa (yani "try-catch" veya "try-finally" blokları), bu kod çalışır ve program devam eder. Ancak, eğer programda uygun bir exception işleme kodu bulunmuyorsa, JVM bu sorumluluğu varsayılan bir işlemciden bekler ve programı beklenmedik bir şekilde sonlandırır.

Varsayılan exception işleyici, exception hakkında bilgi veren bir açıklama, bir exceptionın stacktrace'ini ve exception oluşma konumunu görüntüler. Programın aniden sonlandırılması, bu işlemin dezavantajlarından biridir çünkü programlar düzgün bir şekilde sona ermeli ve kullanıcıya hata mesajları verilmelidir. Bu nedenle, Java programlarında uygun exception işleme kodu eklemek, programın daha güvenilir ve kullanıcı dostu olmasına yardımcı olur.

1. **Java’da throw keywordunu açıklayınız?**

Java'da "throw" kelimesi, genellikle JVM'nin exceptionları fırlatmasına izin verdiği ve bu exceptionları try-catch blokları kullanarak ele aldığımız durumların dışında, kullanıcı tanımlı exceptionları veya çalışma zamanı exceptionlarını açıkça fırlatmak için kullanılır.

"throw" kelimesinin kullanımı için genel sözdizimi:

throw throwableInstance;

Burada, throwableInstance bir Throwable veya onun alt sınıflarından biri olmalıdır.

"throw" ifadesi çalıştırıldığında, bu ifadeden sonraki ifadeler çalıştırılmaz ve işlem durur. Ardından, JVM fırlatılan exceptionı ele alacak bir catch bloğu olup olmadığını kontrol eder. Eğer yoksa, bir sonraki catch ifadesine geçer ve uygun bir işleyici bulana kadar devam eder. Eğer uygun bir işleyici bulunamazsa, varsayılan exception işleyici programı durdurur ve exception açıklamasını ve konumunu yazdırır.

Genellikle "throw" kelimesini, kullanıcı tanımlı veya özel exceptionları fırlatmak için kullanırız. Bu şekilde, programın belirli durumları ele almasını sağlayabilir ve hata durumlarını düzgün bir şekilde işleyebiliriz.

1. **Java’da “throw” kullanımından sonra kod yazılır mı?**

“throw” ifadesi çalıştırıldığında, kontrol akışı ilgili catch bloğuna veya uygun bir işleyiciye geçer. throw ifadesinden sonraki ifadeler, exception fırlatıldığı için ulaşılamaz kod olarak kabul edilir. Bu nedenle, throw ifadesinden sonra herhangi bir kod yazmaya çalışırsanız, derleme zamanında "ulaşılamaz kod" hatası alırsınız. Derleyici, throw ifadesinden sonraki kodun hiçbir zaman çalıştırılamayacağını belirler ve bu durumu size bildirmek için bir hata mesajı oluşturur.

1. **Java’da throw keywordunun önemini açıklayınız?**

Java'da throws anahtar kelimesinin önemi :

Metodun İmzasında Belirtme: throws ifadesi, bir metodun imzasının sonunda kullanılır ve belirli bir türde bir exception bu metodun içinden fırlatılabileceğini belirtir.

İstisna İşleme Sorumluluğunu Delege Etme: throws ifadesinin temel amacı, kontrol edilen exceptionların durumunda exception işleme sorumluluğunu çağıran metotlara devretmektir. Yani, bir metodun içinde bir kontrol edilen exception (checked exception) oluşursa, bu durumu ele almaktan ziyade, metot throws ifadesi ile çağırana bu exception ile başa çıkma sorumluluğunu atar.

Unchecked (Kontrolsüz) İstisnalarda Gerekli Değil: Kontrolsüz exceptionlar (unchecked exceptions) için throws ifadesini kullanmak zorunlu değildir. Bu tür exceptionların genellikle programcı hatası veya beklenmeyen durumlar olduğu düşünülerek, bu exceptionları ele almak yerine programın çökmesine izin verilir.

Sadece Throwable Türleri İçin Kullanılabilir: throws ifadesi sadece Throwable türleri veya bu sınıflardan türetilmiş exception türleri için kullanılabilir. Aksi takdirde, uyumsuz türlerle ilgili bir derleme hatası alırsınız.

Öneri: Yalnızca Belirli İstisna Türlerini Belirtme: Metot içinde fırlatılan exceptionların alt sınıfları yerine, belirli exception türlerini belirtmek daha iyidir. Bu, kodun daha esnek ve anlaşılır olmasına katkıda bulunur.

class Test {

public static void main(String args[]) throws SpecificException {

// ...

}

}

Bu örnekte, SpecificException veya bu sınıfın alt sınıflarından biri olabilecek bir istisna fırlatılabileceği belirtilmiştir.

1. **Finally ve return kullanımı nasıl olur?**

finally bloğu ve return ifadesi birlikte kullanıldığında, finally bloğu her iki durumda da (try veya catch bloğu içinde olsun veya olmasın) önemli bir rol oynar. Özellikle finally bloğu, programın normal akışını bozmaksızın son temizleme veya kaynak salma işlemlerini gerçekleştirmek için kullanılır. Bu, programın her durumda düzgün bir şekilde kapatılmasını sağlar.

Önemli bir senaryo şudur:

Return İfadesi Inside Try veya Catch Bloğu:

public int exampleMethod() {

try {

// Some code that may throw an exception

return 42;

} catch (Exception e) {

// Handle the exception

return -1;

} finally {

// This block will be executed no matter what

System.out.println("Finally block executed");

}

}

Bu örnekte, try bloğu içinde veya catch bloğu içinde bir return ifadesi bulunmaktadır. Ancak, bu return ifadesi çağrıldığında bile, finally bloğu yine de çalışacaktır.

Finally Bloğu Önceliklidir:

Eğer bir finally bloğu ve return ifadesi birlikte kullanılıyorsa, finally bloğu her zaman önceliklidir. Yani, finally bloğu tamamlandıktan sonra return ifadesi işlenir.

Örneğin, yukarıdaki exampleMethod örneğinde, eğer try bloğu içindeki return 42; satırı çalışırsa, bu değeri döndürmeden önce finally bloğu çalışacaktır. Dolayısıyla, sonuç olarak önce "Finally block executed" yazısı görülecektir, ardından 42 değeri döndürülecektir.

Bu, kaynakları serbest bırakma, temizlik işlemleri veya her iki durumda da yapılması gereken işlemler gibi durumlar için kullanışlıdır. finally bloğu, hata oluşsa bile belirli işlemlerin her zaman gerçekleştirilmesini sağlar. Bu nedenle, finally bloğu, özellikle kaynak yönetimi gibi durumlarda, programın düzgün çalışmasını sağlamak için önemlidir.

1. **Checked exceptions için catch kullanabilir miyiz?**

Java'da, kontrol edilen exceptionların (checked exceptions) işlenmesi için catch blokları eklemek gereklidir. Ancak, bir kod bloğunda kontrol edilen bir exception olasılığı yoksa, yani belirli bir blokta bir hata oluşma ihtimali yoksa, bu durumda bu exceptionı işlemek için bir catch bloğu eklemek zorunlu değildir.

Örneğin:

public class Example {

public static void main(String[] args) {

try {

// Kodunuz burada

} catch (CheckedException ex) {

// Eğer buraya hiçbir zaman gelinmiyorsa, bu bloğa gerek yok.

// Bu durumda, bu catch bloğu "unreachable code" hatası alabilir.

}

}

}

Yukarıdaki örnekte, catch (CheckedException ex) bloğu, eğer CheckedException türünde bir istisna fırlatma olasılığı yoksa gereksiz olacaktır. Bu durumda, bu catch bloğu "ulaşılamaz kod" hatası alabilir.

Bu nedenle, kodunuzda bir istisna olasılığı olup olmadığını dikkatlice değerlendirmek önemlidir. Eğer bir kod bloğunda belirli bir istisna türü olmayacağı kesinse, o istisnayı işlemek için gereksiz bir catch bloğu eklemekten kaçınılmalıdır. Aksi takdirde, derleme sırasında "ulaşılamaz kod" hatası alabilirsiniz.

1. **Kullanıcı tanımlı istisnalar (user-defined exceptions) ne demektir?**

Kullanıcı tanımlı exceptionlar (user-defined exceptions), programcıların özel durumları belirtmek ve özel hata mesajları oluşturmak için oluşturdukları exception türleridir. Bu tür exceptionlar, hem kontrol edilen (checked) exception türleri olarak hem de kontrolsüz (unchecked) exception türleri olarak oluşturulabilir.

Kontrol Edilen (Checked) User-Defined Exception:

Kullanıcı tanımlı exception türleri, Exception sınıfını genişleterek veya kontrol edilen exception sınıflarından birini genişleterek oluşturulabilir. Kontrol edilen bir exception, programcının bu exception türünü belirtmek için throws ifadesini kullanması gerektiği bir türdür.

public class MyCheckedException extends Exception {

// ...

}

Bu durumda, MyCheckedException sınıfı, Exception sınıfını genişleterek kontrol edilen bir exception haline gelir.

Kontrolsüz (Unchecked) User-Defined Exception:

Kullanıcı tanımlı exception türleri, RuntimeException sınıfını genişleterek kontrolsüz bir exception haline getirilebilir. Bu tür exception türleri, programcının throws ifadesini kullanmak zorunda olmadığı, yani yakalanma veya işlenme zorunluluğu olmayan exception türleridir.

public class MyUncheckedException extends RuntimeException {

// ...

}

Bu durumda, MyUncheckedException sınıfı, RuntimeException sınıfını genişleterek kontrolsüz bir exception haline gelir.

Not:

Genellikle özel bir durumu temsil etmek için kontrolsüz exception türlerini kullanmak daha yaygındır. Bu, programcının herhangi bir throws ifadesi eklemek zorunda kalmadan exception türünü kullanabilmesine olanak tanır. Bu nedenle, özel exception sınıflarını oluştururken, mümkünse RuntimeException sınıfını genişletmek önerilir.

1. **Java'da iç içe try bolakları kullanabilir miyiz ?**

Java'da try ifadelerini iç içe yerleştirebiliriz. Bir try ifadesini başka bir try ifadesinin bloğu içinde tanımlayabiliriz. Bu, belirli bir kod bloğunda farklı seviyelerde hata işleme stratejileri uygulamamıza olanak tanır.

Bir örnek:

public class NestedTryExample {

public static void main(String[] args) {

try {

// Dış try bloğu

System.out.println("Outer Try Block");

try {

// İç try bloğu

System.out.println("Inner Try Block");

int result = 10 / 0; // Bu satır bir ArithmeticException fırlatabilir

} catch (ArithmeticException innerException) {

// İç try bloğundaki ArithmeticException'ı işle

System.out.println("Inner Catch Block: " + innerException.getMessage());

}

System.out.println("After Inner Try-Catch Block");

} catch (Exception outerException) {

// Dış try bloğundaki genel istisnayı işle

System.out.println("Outer Catch Block: " + outerException.getMessage());

}

}

}

Bu örnekte, dış try ifadesi ile iç içe geçmiş bir iç try ifadesi bulunmaktadır. Eğer iç try bloğunda bir hata oluşursa, iç catch bloğu bu hatayı ele alacaktır. Eğer dış try bloğunda bir hata oluşursa, dış catch bloğu bu hatayı ele alacaktır. Bu, programcılara farklı durumları belirli bloklarda işlemek için esneklik sağlar.

1. **Throwable class ve methotlarının önemini açıklayınız?**

Throwable sınıfı, Java'daki exceptionların temel sınıfıdır. Bütün exceptionlar, bu Throwable sınıfından türetilir. Throwable sınıfının iki ana alt sınıfı Exception ve Error sınıflarıdır. Throwable sınıfının tanımladığı üç önemli metot şunlardır:

printStackTrace() Metodu:

Bu metot, exception bilgilerini aşağıdaki formatta yazdırır:

İstisna adı

Açıklama

Stack trace (yığın izi)

try {

// ... bir exception fırlatabilecek kod

} catch (Exception e) {

e.printStackTrace();

}

Bu metot, hatanın nerede oluştuğunu belirlemek ve hatayı ayıklamak için çok kullanışlıdır.

getMessage() Metodu:

Bu metot, exception nesnesinin sadece açıklama (description) kısmını yazdırır. Bu, exception hakkında daha spesifik bilgiye ihtiyaç duyulmadığında kullanışlıdır.

try {

// ... bir exception fırlatabilecek kod

} catch (Exception e) {

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

}

Bu şekilde, sadece exception açıklamasını elde edebilirsiniz.

toString() Metodu:

Bu metot, exception nesnesinin adını ve açıklamasını yazdırır. printStackTrace() metodu gibi detaylı bir yığın izini yazdırmaz, sadece exception adı ve açıklamasını döndürür.

try {

// ... bir exception fırlatabilecek kod

} catch (Exception e) {

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

}

Bu metot, exception hakkında temel bilgi sağlamak için kullanışlıdır.

1. **ClassNotFoundException ne zaman fırlatılır, açıklayınız?**

ClassNotFoundException, JVM'nin bir sınıfı yüklemeye çalıştığında sınıfın bulunamaması durumunda ortaya çıkan bir exceptiondır. Bir sınıf adı, metin dizesi olarak verildiğinde ve JVM bu sınıfı yüklemeye çalıştığında, ancak belirtilen isimde bir sınıf bulunamadığında bu exception fırlatılır.

Örneğin, aşağıdaki gibi bir durumda ClassNotFoundException ortaya çıkabilir:

public class Main {

public static void main(String[] args) {

try {

// Belirtilen sınıf adıyla bir sınıf yüklemeye çalışıyoruz

Class.forName("BilinmeyenSinif");

} catch (ClassNotFoundException e) {

// ClassNotFoundException durumunda buraya düşeriz

e.printStackTrace();

}

}

}

Bu örnekte, Class.forName("BilinmeyenSinif"); satırı, belirtilen sınıf adını yüklemeye çalışır. Ancak, "BilinmeyenSinif" adında bir sınıf bulunmadığı için JVM ClassNotFoundException fırlatır. Bu durum, sınıf adının yanlış yazıldığı veya var olmadığı bir durumu temsil eder.

ClassNotFoundException durumu, sınıf adlarını dinamik olarak belirleyerek veya yükleme işlemlerini yürütürken kullanıcı tarafından tanımlanan sınıfların yüklenmesi sırasında sıkça karşılaşılır. Programlar dinamik olarak sınıfları yüklerken veya sınıf adlarını kullanıcı girişi veya dış kaynaklardan alırken bu tür bir durum ortaya çıkabilir.

1. **NoClassDefFoundError ne zaman fırlatılır, açıklayınız?**

NoClassDefFoundError, JVM'nin bir sınıfı yüklemeye çalıştığında sınıfın tanımının bulunamaması durumunda ortaya çıkan bir hatadır. Bu hatanın sebepleri şunlar olabilir:

Sınıf Tanımının Bulunamaması:

Sınıfın tanımı compile (derleme) aşamasında var olabilir, ancak runtime (çalışma) aşamasında bu tanım bulunamazsa NoClassDefFoundError hatası ortaya çıkar.

Yanlış Sınıf Adı:

Sınıf adının yanlış yazılması veya belirtilen sınıf adının geçerli bir sınıfı işaret etmemesi durumunda bu hata ortaya çıkabilir.

Classpath Problemleri:

Sınıfın byte kodlarını içeren sınıf dosyasının veya sınıf dizininin, classpath üzerinde bulunamaması durumunda bu hatayla karşılaşılabilir.

Byte Kod Dosyasının Kaldırılması veya Değiştirilmesi:

Sınıfın byte kodlarını içeren dosyanın silinmesi veya değiştirilmesi durumunda, JVM sınıfı yükleyemez ve NoClassDefFoundError hatası alınabilir.

Örnek bir durumu göstermek için:

public class Main {

public static void main(String[] args) {

try {

// BilinmeyenSınıf adındaki sınıfı yüklemeye çalışıyoruz

BilinmeyenSinif obj = new BilinmeyenSinif();

} catch (NoClassDefFoundError e) {

// NoClassDefFoundError durumunda buraya düşeriz

e.printStackTrace();

}

}

}

Bu örnekte, BilinmeyenSinif adında bir sınıfı yüklemeye çalıştık, ancak bu sınıfın tanımı bulunamadığı için NoClassDefFoundError hatası alırız. Bu hatanın genellikle sınıfın adının yanlış yazılması, classpath sorunları veya sınıf dosyasının eksik veya bozuk olması gibi sebepleri vardır.

1. **Process(işlem) nedir?**

Bir işlem (process), bir programın yürütülen halidir. Yani, bir uygulama başlatıldığında, bu uygulama bir işlem olarak adlandırılır. Her işlem, kendi bellek alanına (adres alanına) sahiptir, bu nedenle bir işlem kendi veri, kod ve çalışma zamanı verilerini içerir.

İşlemler genellikle ağır işlemlerdir çünkü her biri kendi bellek alanını, kaynaklarını, ve diğer işletim sistemi kaynaklarını kullanır. İşlemler arasında izolasyon sağlamak için ayrı ayrı çalışırlar. Eğer bir işlem çökerse, diğer işlemler bundan etkilenmez.

İşlemler genellikle bir veya daha fazla "thread"(işlem parçacığı) içerir. Bir "thread", bir işlem içindeki bir yürütme yolunu temsil eder. İşlemin içindeki thread'ler, aynı bellek alanını paylaşır, bu nedenle aynı işlem içindeki thread'ler birbirleriyle veri paylaşabilirler.

Bu nedenle, bir işlem, bir programın yürütülen hali olarak düşünülebilir ve bu işlem, kendi bellek alanına ve kaynaklara sahip, bağımsız bir yürütme birimidir.

1. **Java’da thread (iş parçacığı) nedir?**

İş parçacığı, bir programın içindeki ayrı bir yürütme yolu ya da işlevidir. Her iş parçacığı, programın içinde bağımsız bir yürütme akışını temsil eder.

İş Parçacıkları Hafif (Lightweight) ve Aynı Adres Alanını Paylaşır:

İş parçacıkları, hafif yani düşük kaynak tüketen yürütme birimleridir. Bir programın içindeki iş parçacıkları aynı adres alanını paylaşırlar, bu da demektir ki aynı bellek alanını kullanabilirler.

İş Parçacığı Oluşturmak, İşlem Oluşturmaktan Daha Basittir:

İş parçacıkları oluşturmak, işlemleri oluşturmaktan daha basittir çünkü iş parçacıkları daha az kaynak tüketirler. Bu nedenle, bir program içinde yeni bir iş parçacığı oluşturmak daha hızlı ve daha etkili bir yöntemdir.

İş Parçacıkları Bir İşlem İçinde Varlığını Sürdürür:

Bir işlem (process) en az bir iş parçacığı içerir. Yani, bir program çalıştığında, en az bir iş parçacığı bulunur. İş parçacıkları, bir işlem içinde varlığını sürdürür ve bu işlemin içinde birbirleriyle etkileşimde bulunabilirler.

İş parçacıkları, eş zamanlılık ve çok görev yetenekleri sağlayarak programların daha etkili ve hızlı çalışmasına katkıda bulunurlar. Bu nedenle, Java ve diğer çoklu görev destekleyen programlama dillerinde iş parçacıkları önemli bir kavramdır.

1. **İşlem (Process) ve İş Parçacığı (Thread) arasındaki farklar nelerdir?**

|  |  |
| --- | --- |
| process | thread |
| Yürütülen bir programdır. | Program içinde ayrı bir yürütme yolunu temsil eder. Bir veya daha fazla iş parçacığına sahip olabilir ve bu bir veya daha fazla iş parçacığı bir işlemi oluşturur. |
| İşlemler daha ağırdır | İş parçacıkları daha hafiftir. |
| İşlemler, kendi bağımsız adres alanlarına sahiptir. | İş parçacıkları, aynı işlem içinde aynı adres alanını paylaşır. |
| İşlemler arasındaki iletişim genellikle maliyetlidir. | İş parçacıkları arasındaki iletişim, işlemlere göre daha az maliyetlidir. |
| İşlemler arasında bağlam değiştirmek maliyetlidir. | İş parçacıkları arasında bağlam değiştirmek daha düşük maliyetlidir. |

1. **Çoklu görev (multitasking) nedir?**

Çoklu görev, bilgisayar kullanımında aynı anda birden fazla işlemi gerçekleştirme anlamına gelir. Örneğin, bir elektronik tablo programını kullanırken aynı anda bir hesap makinesi uygulamasını da kullanmak gibi. Bu, bilgisayarın kaynaklarını etkili bir şekilde kullanarak kullanıcılara farklı görevleri aynı anda yerine getirme yeteneği sağlar. Bu, iş verimliliğini artırabilir ve kullanıcıların daha efektif bir şekilde çalışmasına olanak tanır.

1. **Çoklu görev (multitasking) türleri nelerdir?**

Çoklu görevin farklı türleri şunlardır:

*1.İşlem Tabanlı Çoklu Görev (Process Based Multitasking):*

Bu tür çoklu görev, iki veya daha fazla programın aynı anda çalıştırılmasına olanak tanır.

İşlem tabanlı çoklu görevde, bir işlem, kodun en küçük parçasını temsil eder.

Örnek: Ms Word ve Ms PowerPoint programlarını aynı anda çalıştırmak.

*2.İş Parçacığı Tabanlı Çoklu Görev (Thread Based Multitasking):*

Bu tür çoklu görev, bir programın farklı parçalarının aynı anda çalıştırılmasına olanak tanır.

İş parçacığı tabanlı çoklu görevde, bir programın farklı işlevsel parçaları olan iş parçacıkları aynı anda çalışabilir.

Örnek: Bir belgedeki metni biçimlendirirken aynı anda belgeyi yazıcıya göndermek.

Java, iş parçacığı tabanlı çoklu görevi destekler ve çoklu iş parçacığına dahili destek sağlar.

1. **Multithread(çoklu iş parçacıklı) programlamanın avantajları nelerdir?**

Çoklu iş parçacıklı programlama, CPU'nun boş zamanını başka bir iş parçacığına kullanma imkanı sağlar ve bu da programın daha hızlı bir şekilde çalışmasına neden olur. Tek iş parçacıklı bir ortamda her görevin tamamlanması gerektiği için bir sonraki göreve geçilmeden önce CPU boşta bekler.

Örneğin, bir programda tek bir iş parçacığı kullanılıyorsa, bir görevin tamamlanması beklenmeden bir sonraki görev başlatılamaz. Bu durumda CPU, bir görevin tamamlanmasını beklerken boşta durur ve bu süre içinde başka bir iş yapmaz. Ancak çoklu iş parçacıklı bir programda, bir iş parçacığı bir görevle meşgulken diğer iş parçacıkları boşta kalan CPU zamanını değerlendirebilir ve başka görevleri gerçekleştirebilir. Bu, işlemcinin daha etkin bir şekilde kullanılmasına olanak tanır ve programın daha hızlı çalışmasını sağlar.

1. **Java’da thread (iş parçacığı) kavramını açıklayınız?**

Thread Nedir?

Bir program içinde bağımsız bir yürütme yolunu temsil eder. Yani, bir program içinde ayrı bir çalışma akışını ifade eder.

İş parçacıkları, programın içinde aynı anda birden fazla görevi yerine getirmesine olanak tanır.

Thread Yapısı:

Bir iş parçacığı, üç ana bölümden oluşur: Sanal CPU (Virtual CPU), kod ve veri.

Sanal CPU, iş parçacığının yürütme sürecini kontrol eden sanal bir işlemciyi temsil eder.

Kod, iş parçacığının yürütülen programı ifade eder.

Veri, iş parçacığının kullanımındaki bellek alanını temsil eder.

Çalışma Zamanında Paylaşım:

Çalışma zamanında, iş parçacıkları aynı kodu ve veriyi paylaşır; yani, aynı adres alanını kullanırlar.

Bu, iş parçacıklarının birbirleriyle iletişim kurmasını ve veri paylaşmasını sağlar.

Java'da Thread:

Java'da her iş parçacığı, "java.lang.Thread" sınıfının bir nesnesidir.

Java, çoklu iş parçacığına olanak tanıyan bir programlama dilidir ve iş parçacığı oluşturmak ve yönetmek için Java'da özel destek sağlanmıştır.

1. **Java’da threadi destekleyen API lar nelerdir?**

java.lang.Thread:

Thread sınıfını kullanarak bir iş parçacığı oluşturmanın bir yoludur. Thread sınıfını genişleterek ve run() metodunu override ederek bir iş parçacığı oluşturulabilir.

java.lang.Runnable:

Java'da bir arayüz olan Runnable'ı uygulayarak ve run() metodunu override ederek bir iş parçacığı oluşturmanın başka bir yoludur.

java.lang.Object:

Object sınıfı, tüm Java sınıflarının üst sınıfıdır. Bu sınıfta yer alan wait(), notify() ve notifyAll() gibi metodlar, iş parçacıkları arasında senkronizasyonu sağlamak için kullanılır.

java.util.concurrent:

Bu paket, eş zamanlı programlamayı destekleyen sınıf ve arabirimleri içerir.

Örneğin, Executor arabirimi ve FutureTask sınıfı gibi sınıflar, çoklu iş parçacıklarını daha etkili bir şekilde yönetmeye yardımcı olur.

Bu API'lar, Java dilinde çoklu iş parçacığı programlamasını kolaylaştırmak ve yönetmek için kullanılır. "java.lang.Thread" ve "java.lang.Runnable" gibi temel sınıflar, iş parçacıkları oluşturmak ve yönetmek için kullanılan temel araçlardır. "java.util.concurrent" paketi ise daha gelişmiş çoklu iş parçacığı programlama özelliklerini sunar.

1. **Java'da "main thread" (ana iş parçacığı) kavramını açıklayınız?**

Main thread, bir program başlatıldıktan hemen sonra başlayan ilk iş parçacığıdır(thread). Main thread'in önemli olmasının nedenleri şunlardır:

Tüm Alt İş Parçacıkları Ana İş Parçacığından Oluşturulur:

Program başladığında, ana iş parçacığı (main thread) otomatik olarak oluşur ve bu ana iş parçacığından tüm diğer iş parçacıkları türetilir.

Main Metodu, Yürütmenin Son İş Parçacığıdır:

Java programları genellikle main metodunu içerir. Main metodunun tamamlanması, programın yürütmesinin sona ermesi anlamına gelir.

Diğer tüm iş parçacıkları ana iş parçacığından türetildiği için, ana iş parçacığı tamamlandığında diğer iş parçacıkları da sona erer.

JVM, main Metodunu Çağırarak Yeni Bir İş Parçacığı Başlatır:

JVM (Java Virtual Machine), bir Java programını çalıştırdığında, main metodunu çağırarak yeni bir iş parçacığı başlatır.

main() metodunun içindeki kodlar çalışmaya başlar, ancak bu sırada JVM, yeni bir iş parçacığı oluşturur ve bu yeni iş parçacığı üzerinden programın geri kalanını yürütür.

Bu nedenle, main() metodunun içindeki kodlar çalışırken, aslında ana iş parçacığı geçici bir süre duraklar ve yeni iş parçacığı çalışmaya başlar.

Java programları genellikle çoklu iş parçacığı kullanımını destekler ve ana iş parçacığı, programın başladığı noktadır ve diğer iş parçacıklarının temelini oluşturur. Bu, programın eşzamanlı ve etkili bir şekilde çalışmasını sağlamak için kullanılır.

1. **Java'da iş parçacıkları (threads) oluşturmanın kaç yolu vardır?**

Java'da iş parçacıkları (threads) oluşturmanın iki temel yolu vardır.

1.Thread Sınıfını Extend etme:

Bir iş parçacığı oluşturmanın bir yolu, Thread sınıfını extend ederek yeni bir sınıf oluşturmaktır.

Bu sınıf, Thread sınıfından türetilir ve run metodunu override ederek iş parçacığının ana mantığını tanımlar.

Örneğin:

public class MyThread extends Thread {

public void run() {

// İş parçacığının ana mantığı burada tanımlanır.

}

}

// İş parçacığını başlatmak için

MyThread myThread = new MyThread();

myThread.start();

Runnable Arayüzünü Uygulayarak:

Başka bir yol, Runnable arayüzünü implemets ederek bir iş parçacığı oluşturmaktır.

Runnable arayüzü, run metodunu içeren bir arayüzdür. Bu metod, iş parçacığının ana mantığını içerir.

Yeni bir sınıf oluşturulur ve bu sınıf Runnable arayüzünü implements eder. Ardından, bir örneği oluşturulan bu sınıf, bir Thread nesnesine parametre olarak verilerek çalıştırılır.

Örneğin:

public class MyRunnable implements Runnable {

public void run() {

// İş parçacığının ana mantığı burada tanımlanır.

}

}

// İş parçacığını başlatmak için

Thread myThread = new Thread(new MyRunnable());

myThread.start();

Bu iki yöntem de iş parçacığı oluşturmanın yaygın ve temel yollarını temsil eder.

1. **Thread oluşturmanın en iyi yolu hangisidir?**

Java'da iş parçacığı oluşturmanın en iyi yaklaşımı genellikle Runnable arabirimini uygulamaktır. Nedeni şu şekildedir:

Esneklik:

Runnable arabirimini uygulayarak iş parçacığı oluşturmak, sınıfınızın başka bir sınıfı genişletmesine olanak tanır.

Java'da çoklu kalıtım desteklenmez, bu nedenle Thread sınıfını genişlettiğinizde başka bir sınıfı genişletemezsiniz. Bu durum, sınıfınızın özellikle başka bir sınıfı genişletmesi gerektiğinde sorun yaratabilir.

Arayüz Odaklı Programlama:

Runnable arabirimini uygulamak, arayüz odaklı programlamayı teşvik eder. Bu, sınıflar arasındaki bağımlılığı azaltır ve kodun daha modüler olmasını sağlar.

Thread Pool ve Executor Framework:

Java'da Runnable arayüzünü uygulayan sınıfları kullanarak, Executor framework ve thread pool gibi gelişmiş çoklu iş parçacığı yönetim araçlarından daha iyi yararlanabilirsiniz.

Kodun Daha Okunabilir Olması:

Runnable arayüzünü uygulamak, kodun daha okunabilir ve bakımı daha kolay olmasını sağlar.

Örnek:

public class MyRunnable implements Runnable {

@Override

public void run() {

// İş parçacığının ana mantığı burada tanımlanır.

}

}

// İş parçacığını başlatmak için

Thread myThread = new Thread(new MyRunnable());

myThread.start();

Yukarıdaki örnekte, MyRunnable sınıfı, Runnable arabirimini uygular ve yeni bir iş parçacığı başlatmak için Thread sınıfını kullanır. Bu, kodun daha esnek ve okunabilir olmasını sağlar.

1. **Java'da thread scheduler'ın önemini açıklayınız?**

Thread scheduler (iş parçacığı planlayıcısı), JVM'in bir anda birden fazla iş parçacığı olduğunda hangi iş parçacığının çalıştırılacağını belirlemek için kullandığı bir bölümdür. Sadece "runnable" (çalışabilir) durumdaki iş parçacıkları, thread scheduler tarafından seçilir.

Thread scheduler, öncelikli olan iş parçacıklarına önce mikroprosesör zamanı tahsis eder. Aynı önceliğe sahip iş parçacıkları arasında mikroprosesör zamanını tahsis etmek için ise thread scheduler round robin (dönüş sırası) yöntemini takip eder.

Önemli Noktalar:

Öncelikli İş Parçacıkları:

Thread scheduler, öncelikli iş parçacıklarına öncelik verir. Yüksek öncelikli iş parçacıkları, düşük öncelikli iş parçacıklarından önce mikroprosesör zamanı alır.

Round Robin Yöntemi:

Aynı önceliğe sahip iş parçacıkları arasında zaman paylaşımı yapmak için round robin yöntemini kullanır. Bu, iş parçacıkları arasında adil bir dağılım sağlar.

Runnable Durumdaki İş Parçacıkları:

Sadece "runnable" durumdaki iş parçacıkları, thread scheduler tarafından seçilir. Bu, iş parçacıklarının sadece çalışmaya hazır olduğu durumda mikroprosesör zamanı almalarını sağlar.

Java'da thread scheduler, çoklu iş parçacıklarının etkili bir şekilde çalışmasını sağlamak ve mikroprosesör kaynaklarını adil bir şekilde dağıtmak için kullanılır. Bu sayede öncelikli ve çalışabilir durumdaki iş parçacıkları, belirli bir düzen ve adil bir şekilde mikroprosesör zamanı alabilirler.

1. **Java’da thread life cycle ( yaşam döngüsünü) açıklayınız?**

Yeni (New) Durumu:

Bir iş parçacığı örneği oluşturulduğunda, iş parçacığı "New" durumundadır.

Örneğin:

Thread t = new Thread();

Yukarıdaki örnekte, t isimli iş parçacığı "new" durumundadır. İş parçacığı oluşturulmuş, ancak henüz aktif durumda değildir. Aktif hale getirmek için start() metodunu çağırmamız gerekir.

Çalışabilir (Runnable) Durumu:

İş parçacığı, iki şekilde "runnable" durumuna geçebilir:

a) start metodu çağrıldığında.

b) Bir iş parçacığı, bloke olmuş veya uyku durumundan döndükten sonra da "runnable" durumuna geçebilir.

Çalışan (Running) Durumu:

İş parçacığı, thread scheduler tarafından CPU zamanı tahsis edildiğinde "running" durumuna geçer.

Bekleme/Bloke/Uyku (Waiting/Blocking/Sleeping) Durumu:

İş parçacığı, kısa bir süre için pasif hale getirilebilir. Bu durumda iş parçacığı, aşağıdaki durumlardan birinde "bekleme" durumuna geçer:

İş parçacığı, bir nesnenin kilidini almayı bekliyor.

İş parçacığı, başka bir iş parçacığının tamamlanmasını bekliyor.

İş parçacığı, diğer bir iş parçacığının bildirimini bekliyor.

Bitmiş (Dead) Durumu:

İş parçacığı, run metodunun çalışması tamamlandığında "dead" durumuna geçer.

İş parçacığı, run metodunun tamamlanmasıyla otomatik olarak sona erer ve iş parçacığı nesnesi çöp toplama tarafından yok edilir.

Java'daki iş parçacığı yaşam döngüsü, iş parçacıklarının oluşturulması, çalıştırılması, bekletilmesi ve sonlandırılması gibi farklı durumları içerir. Bu durumlar, iş parçacıklarının etkileşimli ve efektif bir şekilde çalışmasını sağlar.

1. **Dead thread tekrar başlatılabilir mi?**

Eğer bir iş parçacığı (thread) ölmüş (dead) durumdaysa ve start methodu kullanılarak tekrar başlatmaya çalışılırsa, bu durumda çalışma zamanı hatası alınır. Çünkü bir kez ölen bir iş parçacığı, bir daha başlatılamaz.

İş parçacığı öldüğünde, run metodunun yürütülmesi tamamlanmış ve iş parçacığı sona ermiştir. Bu durumda, aynı iş parçacığı nesnesi üzerinde tekrar start metodunu çağırmak geçerli değildir ve bir IllegalThreadStateException hatası alınır.

Thread myThread = new Thread(() -> {

// İş parçacığının çalışma mantığı

System.out.println("İş parçacığı çalışıyor.");

});

myThread.start(); // İş parçacığı başlatıldı

// ...

myThread.start(); // Hata! İş parçacığı öldüğü için tekrar başlatılamaz

Yukarıdaki örnekte, myThread adlı iş parçacığı başlatıldıktan sonra tekrar başlatılmaya çalışıldığında IllegalThreadStateException hatası alınacaktır. İş parçacığı bir kez çalışma sürecini tamamladığında, başka bir iş parçacığı nesnesi oluşturup onu başlatmak gereklidir.

1. **Bir thread başka threadi bloke edebilir mi?**

Java'da bir iş parçacığı (thread), başka bir iş parçacığını bloke edemez. Ancak, bir iş parçacığı kendi yürütme akışını geçici olarak durdurabilir (bloke edebilir).

Java'da iş parçacıkları arasında koordinasyon ve senkronizasyon mekanizmaları kullanılarak iş parçacıkları arasında bir tür etkileşim sağlanabilir. Bu mekanizmalar şunları içerir:

Locks (Kilitler): İş parçacıkları arasında ortak bir kaynağa erişimi kontrol etmek için kilitler kullanılabilir. Ancak, bir iş parçacığı bir kaynağa kilit koyduğunda, diğer iş parçacıkları bu kaynağa erişemez, ancak başka kaynaklara erişebilir.

Wait ve Notify: İş parçacıkları arasında iletişim sağlamak için wait ve notify metodları kullanılabilir. Bir iş parçacığı, bir koşulu kontrol ederken wait metodunu kullanarak bekleyebilir ve diğer bir iş parçacığı, bu koşulu sağladığında notify metodunu kullanarak diğer iş parçacığını uyandırabilir.

Thread.sleep: Bir iş parçacığı, belirli bir süre boyunca uykuda kalabilir (Thread.sleep kullanarak). Bu, başka bir iş parçacığı için kaynağa erişimi geçici olarak durdurabilir, ancak bu blokaj koordinasyon amacıyla değil, zamanlamayla ilgilidir.

Yani, bir iş parçacığı doğrudan başka bir iş parçacığını bloke edemez, ancak senkronizasyon ve koordinasyon mekanizmaları kullanılarak iş parçacıkları arasında bir tür etkileşim sağlanabilir.

1. **Başlatılan bir thread tekrar başlatılabilir mi?**

Bir thread bir kere başlatıldıktan sonra ikinci kez start yöntemi çağrılırsa (bir kere başlatılan bir iş parçacığı bir daha başlatılamaz), bu durumda RuntimeException türünden bir IllegalThreadStateException hatası oluşacaktır.

1. **Thread pool nedir?**

Thread pool, bir uygulamada aynı anda çalışan çok sayıda iş parçacığını etkili bir şekilde yönetmek için kullanılan bir tasarım desenidir. Thread pool, genellikle aşağıdaki avantajları sağlar:

Performans İyileştirmesi:

Her iş parçacığı oluşturulduğunda bir miktar sistem kaynağı (bellek, işlemci zamanı) harcanır. Thread pool, bu maliyeti azaltarak performansı artırır. Zira önceden oluşturulan iş parçacıkları tekrar kullanılır.

Kaynak Yönetimi:

Thread pool, aynı anda çok sayıda iş parçacığının çalışmasına izin verirken, aşırı iş parçacığı oluşturmanın önüne geçer. Bu, sistem kaynaklarının daha etkili kullanılmasını sağlar.

İstikrarlı Performans:

Sabit bir sayıda iş parçacığı ile çalıştığı için, thread pool, sistemde aşırı iş yükü oluşturarak performans düşüklüğüne yol açan durumları engeller.

İş Parçacığı Yönetimi:

Thread pool, iş parçacıklarını oluşturma, başlatma, sonlandırma ve takip etme gibi görevleri kolaylaştırır.

1. **Volatile keywordu ne işe yarar?**

volatile kelimesi, bir değişkenin değerini hafıza modelindeki bazı özelliklere uygun bir şekilde kullanmayı sağlar. volatile anahtar kelimesinin kullanıldığı bir değişken, bir iş parçacığı tarafından yapılan değişikliklerin diğer iş parçacıkları tarafından hemen görülmesini sağlar. Bu, değişkenin bellek modelindeki ana hafıza konumunu temsil eder.

volatile anahtar kelimesi kullanıldığında, bir değişkenin değeri sadece okuma ve yazma işlemleri sırasında önbellek kullanımı minimize edilerek ana belleğe doğrudan erişimle güncellenir. Bu, diğer iş parçacıklarının güncel değeri hemen görmelerini sağlar.

Bu özellik, özellikle çoklu iş parçacıklı ortamlarda bir değişkenin güncellenmiş değerini diğer iş parçacıklarına hemen göstermesi gerektiği durumlarda kullanılır. Ancak, volatile kullanımı bazı durumlarda yeterli olmayabilir ve daha karmaşık senkronizasyon mekanizmaları (örneğin, synchronized bloklar veya java.util.concurrent kütüphanesindeki araçlar) gerekebilir.

Örnek kullanım:

public class SharedResource {

private volatile boolean flag = false;

public void setFlagTrue() {

flag = true;

}

public boolean checkFlag() {

return flag;

}

Bu örnekte, flag adlı değişken volatile olarak tanımlanmıştır. setFlagTrue metodu, flag değişkeninin değerini true yapar. Diğer iş parçacıkları, checkFlag metodu aracılığıyla flag değişkeninin güncel değerini anında görebilirler. volatile kullanılmış olsaydı, diğer iş parçacıkları bu değişikliği hemen fark edemeyebilirdi.

1. **Thread oluştururken run methodunu override etmezsek ne olur?**

Eğer run metodunu override etmezsek, Thread sınıfının varsayılan run metodunun uygulaması çalıştırılır. Bu durumda, Thread sınıfının run metodu içinde herhangi bir iş yapılması beklenmez ve bu metot boştur.

Eğer bir iş parçacığı (thread) oluştururken kendi run metodumuzu sağlamazsak, sadece Thread sınıfının run metodunun içindeki varsayılan boş işlem gerçekleşir. Bu durumda, iş parçacığı hiçbir özel görev gerçekleştirmeyecek ve derlenen programın davranışı değişmeyecektir.

1. **Java’da run metodunu overload edebilir miyiz?**

Evet, run metodunu overload edebiliriz ancak Thread sınıfının start metodunun her zaman parametre almayan run metodunu çağıracağını unutmamamız önemlidir. Overload run metodunu doğrudan start metodu tarafından çağırmaz.

Yani, eğer bir iş parçacığı sınıfında birden fazla run metodunu tanımlarsak, Thread sınıfının start metodu yalnızca parametre almayan (run() şeklinde olanı) run metodunu çağırır.

1. **Java'da kilit (lock) veya kilitlerin amacı nedir?**

Kilit (lock), Java'da paylaşılan bir kaynağa birden fazla iş parçacığının erişmesini önlemek için kullanılır. Bir kilidin amacı, paylaşılan bir kaynağa erişmek isteyen bir iş parçacığının, öncelikle o kaynağın kilidini alması ve bu kaynağa erişmek için izin almasıdır. Eğer başka bir iş parçacığı tarafından zaten kilit alınmışsa, o zaman diğer iş parçacığı, bu kaynağa erişmek için beklemek zorunda kalır ve kilidin serbest bırakılmasını bekler.

Java'da kilit, synchronized anahtar kelimesi veya Lock arabirimini kullanarak elde edilir. Bir nesneyi kilitlemek için synchronized anahtar kelimesi kullanılır ve bu, ilgili kod bloğunun sadece bir iş parçacığı tarafından aynı anda yürütülmesine izin verir.

Kilitler, kodun belirli bölümlerini koruyarak, sadece bir iş parçacığının aynı anda yürütebileceği şekilde düzenlenir.

Örneğin:

*public class SharedResource {*

*private int count = 0;*

*public synchronized void increment() {*

*count++;*

*}*

*public synchronized int getCount() {*

*return count;*

*}*

*}*

Yukarıdaki örnekte, increment ve getCount metodlarına synchronized anahtar kelimesi uygulanmıştır. Bu, bu metotların aynı anda yalnızca bir iş parçacığı tarafından erişilmesini sağlar ve böylece count değişkenine güvenli bir şekilde erişilir.

1. **Java'da senkronizasyon (synchronization) yapmanın kaç yolu vardır?**

Java'da senkronizasyon yapmanın iki temel yolu vardır:

* 1. Senkronize metotlar (Synchronized methods)
  2. Senkronize bloklar (Synchronized blocks)

Senkronizasyon, aynı anda birden fazla iş parçacığının aynı kod bloğuna veya metoda erişmesini engellemek için kullanılır. Senkronizasyon, paylaşılan kaynaklara güvenli bir şekilde erişimi sağlar ve veri bütünlüğünü korur.

1. **Java’da Senkronize metotlar (synchronized methods) nedir?**

Senkronize metotlar, Java'da bir nesnenin yönteminin aynı anda yalnızca bir iş parçacığı tarafından erişilmesini sağlamak için kullanılır. Bu, paylaşılan kaynaklara güvenli ve koordineli erişim sağlamak için kullanılır. Senkronize bir metot tanımlamak için synchronized anahtar kelimesi kullanılır.

Senkronize bir metot tanımlamak için kullanılan imza şu şekildedir:

*public synchronized void methodName() {*

*// Metot içeriği*

*}*

Senkronize bir metot çağrıldığında, ilgili nesnenin üzerinde otomatik olarak bir kilidin alınması gerekir. Başka bir deyişle, senkronize bir metot çağrıldığında, o metot üzerindeki kilidin otomatik olarak alınması ve metotun yalnızca tek bir iş parçacığı tarafından aynı anda yürütülmesi sağlanır.

Senkronize bir metot, yalnızca o metoda ilişkin kilidin serbest olduğu durumda çalışabilir. Eğer bir başka iş parçacığı tarafından o metoda ilişkin kilidin alındığı durumda, diğer iş parçacığı o metoda erişmek için beklemek zorunda kalır. Böylece, senkronize metotlar paylaşılan kaynaklara güvenli bir şekilde erişim sağlar ve veri bütünlüğünü korur.

1. **Java’da Senkronize metotları ne zaman kullanırız?**

Java'da senkronize metotlar, birden fazla iş parçacığının aynı nesnenin yöntemine erişmeye çalıştığı durumlarda kullanılır. Özellikle, bir yöntem nesnenin durumunu değiştirebiliyorsa ve bu durumun tutarlılığının korunması gerekiyorsa senkronizasyon kullanılır.

Örneğin, bir nesnenin belirli bir yöntemi, nesnenin iç durumunu değiştiriyorsa ve birden fazla iş parçacığı bu yönteme aynı anda erişmeye çalışıyorsa, bu durumda senkronize metotlar kullanılabilir. Senkronize bir metot, sadece bir iş parçacığının aynı anda o metoda erişmesine izin verir, böylece nesnenin durumu tutarlı kalır ve beklenmeyen sonuçlar oluşmaz.

Özetle, senkronize metotlar, bir nesnenin iç durumunu güvenli bir şekilde değiştirmek ve veri bütünlüğünü korumak için kullanılır. Eğer birden fazla iş parçacığı aynı anda bu yönteme erişmeye çalışıyorsa, senkronize bir metot kullanarak bu durumu güvenli bir şekilde ele alabiliriz.

1. **Java’da bir thread senkronize yöntemleri yürütürken, diğerlerini yürütmek mümkün müdür?**

**Yöntemler diğer threadler tarafından aynı anda senkronize edilebilir mi?**

Bir thread bir senkronize metodu çalıştırırken, o metot içindeyken diğer threadlerin aynı nesnenin başka senkronize metotlarını aynı anda çalıştırması mümkün değildir. Bu durumda, bir threadin senkronize bir metodu çalıştırması, o nesnenin üzerinde kilidi alır ve bu nesnenin başka senkronize metotlarına diğer threadlerin erişimini engeller.

Senkronize metotlar, bir nesnenin belirli bir kısmını tek bir thread tarafından aynı anda işlenmesini sağlamak için kullanılır. Dolayısıyla, bir threadin bir senkronize metodu çalıştırırken, o nesnenin diğer senkronize metotlarını başka threadlerin aynı anda çalıştırmasına izin verilmez.

Bu mekanizma, veri bütünlüğünü sağlamak ve senkronizasyonu yönetmek için kullanılır. Eğer birden fazla thread aynı nesneyi değiştirmeye çalışıyorsa, senkronizasyon olmadan beklenmedik sonuçlar oluşabilir. Bu nedenle, senkronize metotlar ve diğer senkronizasyon mekanizmaları, paylaşılan kaynaklara güvenli bir şekilde erişmek için önemlidir.

1. **Java'da senkronize bloklar nelerdir?**

Java'da senkronize bloklar, sadece belirli kod satırlarını veya kod bloklarını senkronize etmek için kullanılır. Bununla birlikte, tüm bir metodu senkronize etmek yerine, yalnızca belirli bir kod parçasını senkronize etmek istediğimiz durumlarda kullanılırlar.

Senkronize bir blok tanımlamak için synchronized anahtar kelimesi kullanılır. Senkronize blokun içine girildiğinde, belirtilen nesnenin kilidi alınır ve bu nesne üzerindeki diğer senkronize bloklara ve senkronize metotlara erişim engellenir. Senkronize bloklar, aynı anda yalnızca bir iş parçacığının belirli kod bloğuna erişmesini sağlar.

Senkronize bir bloğun imzası şu şekildedir:

*synchronized (objectReference) {*

*// Senkronize edilecek kod bloğu*

*}*

objectReference, senkronize bloğun kilidini alacak nesneyi belirtir. Bu nesne, senkronize edilen kod bloğuna aynı anda erişmeye çalışan tüm iş parçacıkları için bir referans noktası sağlar. Genellikle, bu nesne, paylaşılan bir kaynağı temsil eden bir nesnedir.

1. **Senkronize blokları ne zaman kullanırız ve senkronize blok kullanmanın avantajları nelerdir?**

Eğer çok az kod satırı senkronize edilmesi gerekiyorsa, o zaman senkronize blokların kullanılması önerilir. Senkronize blokların senkronize metotlara göre başlıca avantajı, bekleyen iş parçacıklarının bekleme süresini azaltması ve sistem performansını artırmasıdır.

Senkronize bloklar, yalnızca belirli bir kod bloğunu senkronize etmek için kullanılır. Bu, senkronizasyon gerektiren kodun sadece küçük bir kısmının senkronize edilmesine olanak tanır. Bu durum, senkronize blokların senkronize metotlara göre daha esnek olmasını sağlar.

Senkronize blokların avantajları şunlardır:

* Esneklik: Senkronize bloklar, sadece belirli kod parçalarını senkronize etmemize olanak tanır. Böylece, senkronizasyonu yalnızca ihtiyaç duyulan yerlerde uygulayabiliriz.
* Daha Az Bekleme Süresi: Senkronize bloklar, senkronize metotlara göre daha küçük kilitlenme alanlarına sahiptir. Bu, diğer iş parçacıklarının belirli bir bloğa erişimini beklerken daha az bekleme süresi geçirilmesini sağlar.
* Performans Artışı: Senkronize bloklar, senkronize edilmesi gereken kod bloğunun tamamını senkronize etmek yerine, yalnızca belirli bir bölümünü senkronize eder. Bu, iş parçacıklarının daha az kilitlenme durumu yaşamasını sağlar ve böylece genel performansı artırır.

Özet olarak, senkronize bloklar, senkronizasyon gereksinimlerini daha hassas bir şekilde kontrol etmek ve performansı artırmak için kullanılır. Bu nedenle, çok az kodun senkronize edilmesi gerektiğinde veya senkronizasyonun sadece belirli bir kod bloğunda uygulanması gerektiğinde senkronize bloklar tercih edilir.

1. **Java’da class düzeyi kilit nedir?**

Sınıf seviyesi kilidi (class level lock), bir sınıfın örneği yerine sınıfın kendisi üzerinde kilidi almayı ifade eder. Sınıf seviyesi kilidi, bir sınıfın tüm örneklerine uygulanır ve bu nedenle sınıfın tüm örneklerine erişen threadlerin senkronize edilmesini sağlar.

Sınıf seviyesi kilidi ile nesne seviyesi kilidi arasındaki fark şudur: Sınıf seviyesi kilidi, sınıfın kendisi üzerinde alınırken, nesne seviyesi kilidi, sınıfın belirli bir örneği üzerinde alınır.

Örneğin, bir sınıf seviyesi kilidi aşağıdaki gibi tanımlanabilir:

*public class MyClass {*

*public static synchronized void myStaticMethod() {*

*// Class level lock, sınıfın tüm örneklerine uygulanır*

*}*

*}*

Burada, myStaticMethod metodu static olarak tanımlanmış ve synchronized anahtar kelimesiyle işaretlenmiştir. Bu nedenle, bu metoda erişen tüm iş parçacıkları, sınıf seviyesi kilidi üzerinde senkronize edilirler.

Sınıf seviyesi kilidi, genellikle paylaşılan kaynaklara erişim sağlamak ve veri bütünlüğünü korumak için kullanılır. Bununla birlikte, sınıf seviyesi kilidinin kullanımı, gereksinimlere ve uygulamanın doğasına bağlı olarak değişir.

1. **Java'daki statik metodları senkronize edebilir miyiz?**

Java'da static metotları senkronize edebiliriz. Her sınıfın bir kilidi (lock) vardır ve bir thread, statik bir senkronize yöntemi çalıştırmak istiyorsa önce sınıf seviyesindeki kilidi elde etmelidir. Bir thread, statik bir senkronize yöntemini yürütürken, diğer bir thread aynı sınıfın statik senkronize yöntemini çalıştıramaz çünkü kilidin sınıf üzerinde alınmış olması gereklidir.

Ancak, aynı anda aşağıdaki yöntemlerin çalıştırılmasına izin verilir:

Normal static methods

Normal instance methods

Senkronize instance methods

Bir sınıf seviyesi kilidi almak için kullanılan imza şu şekildedir:

*synchronized(ClassAdı.class) {*

*// Kod bloğu*

*}*

Bu yapı, sınıfın kendisi üzerinde bir kilidin alınmasını sağlar. Bu, sınıfın tüm örneklerine aynı anda erişimi kontrol etmek için kullanılır ve sınıf seviyesinde senkronizasyon sağlar.

1. **Primitive için senkronize bloğu kullanabilir miyiz?**

Java'da senkronize bloklar yalnızca nesneler için geçerlidir. Senkronize bloklar, belirli bir nesne üzerinde senkronize edilir ve bu nesne, senkronizasyonun gerçekleştirildiği kilit mekanizmasını sağlar.

Eğer senkronize blokları ilkel veri türleri için kullanmaya çalışırsak, derleme zamanında hata alırız. Çünkü senkronize bloklar yalnızca nesneler üzerinde çalışır ve ilkel veri türleri nesne değildir.

Örneğin, aşağıdaki kod derleme zamanında hata verecektir:

*int counter = 0;*

*synchronized(counter) {*

*// Kod bloğu*

*}*

Çünkü counter bir ilkel veri türüdür (int) ve senkronize bloklar yalnızca nesneler üzerinde çalışır. Dolayısıyla, senkronize blokları ilkel veri türleriyle kullanmaya çalıştığımızda derleme zamanında hata alırız.

1. **Java'da threadlerin öncelikleri ve önceliklerinin önemi nelerdir?**

Thread prioriteleri, bekleyen birçok thread arasında hangi threadin çalıştırılacağını belirler. Java programlama dilinde her threadin bir önceliği vardır. Bir thread, ebeveyn threadin önceliğini miras alır. Varsayılan olarak, bir thread normal bir önceliğe sahiptir.

Thread planlayıcısı (thread scheduler), her threadin ne zaman çalıştırılacağına karar vermek için threadin önceliklerini kullanır. Thread planlayıcısı, daha yüksek öncelikli threadi önce çalıştırır.

Thread prioriteleri, sistemdeki threadlerin çalışma sırasını belirlemekte ve öncelikli threadlerin daha hızlı yanıt vermesini sağlamakta önemlidir. Ancak, thread prioritelerinin mutlak davranışları, Java'nın çalıştığı platforma ve JVM'in yapılandırmasına bağlı olabilir. Bu nedenle, thread prioritelerini kullanırken dikkatli olunmalı ve platform bağımlılıkları göz önünde bulundurulmalıdır.

1. **Farklı threadlerin öncelik türlerini açıklayın?**

Java'da her threadin öncelikleri 1 ile 10 arasındadır. Varsayılan olarak, bir threadin önceliği 5'tir (Thread.NORM\_PRIORITY olarak adlandırılır). Maksimum öncelik 10, minimum öncelik ise 1'dir. Thread sınıfı aşağıdaki sabitleri (static final değişkenler) özelliklerini tanımlamak için tanımlar:

Thread.MIN\_PRIORITY = 1; En düşük öncelik değeri

Thread.NORM\_PRIORITY = 5; Normal öncelik değeri (varsayılan)

Thread.MAX\_PRIORITY = 10; En yüksek öncelik değeri

Öncelik değerleri, threadlerin çalışma sırasını belirler. Öncelik daha yüksek olan threadleri, öncelikleri daha düşük olanlardan önce çalıştırılır. Ancak, işletim sistemi ve JVM'nin uygulama davranışını etkileyebilecek faktörler olduğunu unutmamak önemlidir. Bu nedenle, thread önceliklerini ayarlamak ve kullanmak programcıların dikkat etmesi gereken bir konudur.

1. **Threadin önceliği nasıl değiştirilir veya threadin önceliği nasıl ayarlanır?**

Thread sınıfı, threadin önceliğini ayarlamak için bir set metodu ve threadin önceliğini almak için bir get metodu içerir.

İşte öncelik ayarlamak için kullanılan setPriority metodunun imzası:

*final void setPriority(int value);*

setPriority() metodu, threadin önceliğini ayarlamak için JVM'e bir istek gönderir. Ancak, JVM bu isteği kabul edip etmeme konusunda serbesttir.

Ayrıca, mevcut threadin önceliğini almak için Thread sınıfının getPriority() metodunu kullanabiliriz:

*final int getPriority();*

Bu metodun çağrılması, mevcut threadin öncelik değerini döndürür.

Özetlemek gerekirse, setPriority() metoduyla bir threadin önceliğini ayarlayabilir ve getPriority() metoduyla mevcut threadin önceliğini alabiliriz. Ancak, threadin önceliğini ayarlamak, işletim sistemi ve JVM tarafından belirli koşullara bağlı olduğu için kesin bir sonuç garanti edilmez.

1. **İki threadin önceliği aynıysa hangi thread önce yürütülür?**

Eğer eşit önceliğe sahip iki thread varsa, hangi threadin önce çalıştırılacağı konusunda garanti verilemez. Bu durum, thread planlayıcısına, hangi threadin çalıştırılacağına karar verme yetkisi verir. Thread planlayıcısı şu işlemleri yapabilir:

Havuzdaki herhangi bir threadi seçip çalıştırabilir ve tamamlanana kadar onu çalıştırabilir.

Zaman dilimlemesi (time slicing) yöntemiyle tüm iş parçacıklarına eşit fırsatlar verebilir.

Yani, eğer eşit önceliğe sahip iki thread varsa, hangisinin önce çalıştırılacağı işletim sistemine ve thread planlayıcısının belirlediği faktörlere bağlıdır. Bu nedenle, öncelikler eşit olduğunda hangi threadin öncelikli olduğunu kesin olarak söylemek mümkün değildir.

1. **Threadin yürütülmesini önlemek, durdurmak için hangi yöntemler kullanılır?**

Thread'lerin çalışmasını durduran üç yöntem vardır:

1. yield(): Bir threadin çalışmasını durdurarak, thread planlayıcısına diğer threadlere öncelik vermesi için bir işarettir. Ancak, bu sadece bir öneridir ve thread planlayıcısı, threadnın çalışmasını durdurup durdurmayacağına karar verir.
2. join(): Bir threadin diğer bir threadi tamamlanmasını beklemesini sağlar. Örneğin, bir threadin diğer bir threadin tamamlanmasını beklemesi gereken senaryolarda kullanılır.
3. sleep(): Belirli bir süre boyunca bir threadi uyutur. Bu işlem, belirli bir süre boyunca threadin çalışmasını durdurur ve sonra tekrar devam eder.

Bu yöntemler, threadlerin kontrolünü sağlamak ve senkronizasyonu düzenlemek için kullanılır.

1. **Java’da thread sınıfında yield() metodunu açıklayınız?**

yield() yöntemi, mevcut çalışan threadi bekleyen durumundaki diğer eşit önceliğe sahip threadlere fırsat vermek için çalışma durumundan Runnable durumuna geçirir. yield(), mevcut threadi belirli bir süre için uyutmak için kullanılmaz. Mevcut çalışan threadin çalışma durumunu Runnable durumuna geçirir, böylece thread planlayıcısı diğer eşit önceliğe sahip threadleri çalıştırmak için seçenek sahibi olur.

Ancak, yield() yöntemini çağırmak, threadin bir kilitleme durumunda olduğunda herhangi bir etkiye sahip değildir. Eğer bir thread daha önce bir kilidi almışsa, yield() yöntemi threadin kilidini kaybetmesine neden olmaz.

Yani, yield() yöntemi, mevcut threadi diğer eşit önceliğe sahip threadlere fırsat vermek için Runnable durumuna geçirir. Ancak, bu durumun işleyişi, thread planlayıcısına bağlıdır ve kesinlikle garantilenmez.

1. **yield() yöntemi kullanılan threadin tekrar çalıştırılma şansını yakalaması mümkün mü?**

yield() metodu, mevcut threadi belirli bir süre için uyutmak ve eşit önceliğe sahip diğer iş parçacıklarının çalışmasına fırsat vermek için kullanılır. Ancak, yield() metodunun çağrılmasından sonra uyuyan threade tekrar çalışma fırsatı verilip verilmeyeceği, thread planlayıcısına bağlıdır.

Yani, yield() yöntemi çağrıldıktan sonra uyuyan threadin tekrar çalışma fırsatı alıp almayacağı tamamen thread planlayıcısının insafına bağlıdır. Thread planlayıcısı, uygun olduğunu düşündüğü zamanda uyuyan threadi tekrar çalıştırabilir veya başka bir threade öncelik verebilir. Dolayısıyla, yield() yöntemi çağrıldıktan sonra uyuyan threadin tekrar çalışma fırsatı alıp almayacağı, thread planlayıcısının takdirine bağlıdır.

1. **Java’da thread sınıfında join() yönteminin önemini açıklayınız?**

Thread sınıfındaki join() yönteminin önemi şudur:

Bir thread, diğer bir threadin tamamlanmasını beklemek için join() yöntemini çağırabilir. Örneğin, t1 ve t2 adında iki thread olduğunu varsayalım. Çalışan bir thread t1, join() yöntemini t2 thread üzerinde çağırırsa, t1 thread t2'nin tamamlanmasını bekler ve bu süre boyunca bekleyen durumunda kalır. t2 thread tamamladığında, t1 thread çalışmasını sürdürür.

join() yöntemi, bir threadin başka bir threadi tamamlanmasını beklemesini sağlar. Ancak, join() yöntemi, Interrupted Exception (Kesinti İstisnası) fırlatabilir. Dolayısıyla, join() yöntemini kullanırken Interrupted Exception'ı yönetmek için try-catch bloğu veya throws kullanarak istisnai durumları ele almak önemlidir.

join() yönteminin imzaları şunlardır:

*public final void join() throws InterruptedException {}*

*public final synchronized void join(long millis) throws InterruptedException {}*

*public final synchronized void join(long millis, int nanos) throws InterruptedException {}*

Bu yöntemler, threadin belirli bir süre veya süre ve nanosaniye cinsinden beklemesini sağlar.

1. **Java’da thread sınıfında sleep() yönteminin amacını açıklayınız?**

sleep() yöntemi, mevcut çalışan threadi belirli bir süre için uyutmak için kullanılır. sleep() yöntemi, mevcut threadin çalışmasını belirtilen süre boyunca duraklatır. Ancak, sleep() yöntemi, mevcut threadin tam olarak belirtilen süre boyunca uyumasını garanti etmez. Belirtilen sürenin alt sınırıdır ve thread uyandırıldığında, belirtilen süreden önce uyanabilir.

sleep() yönteminin imzaları şunlardır:

*public static native void sleep(long millis) throws InterruptedException {}*

*public static void sleep(long millis, int nanos) throws InterruptedException {}*

İlk imzada, sadece millisaniye cinsinden bir süre belirtilir ve thread bu süre boyunca uyur. İkinci imzada, millisaniye ve nanosaniye cinsinden bir süre belirtilir. Bu yöntemler, mevcut threadi belirtilen süre boyunca uyutur ve belirli bir süre boyunca çalışmasını duraklatır.

1. **sleep() yöntemi başka bir threadin uyumasına neden olabilir mi?**

Hayır, sleep() yöntemi sadece mevcut threadi uyutmak için kullanılır, başka bir threadi uyutmaz. sleep() yöntemi, sadece çağrıldığı threadi belirli bir süre boyunca uyutmak için kullanılır. Diğer threadler üzerinde herhangi bir etkisi yoktur.

1. **Thread sınıfının Interrupt() yöntemini açıklayın?**

Thread sınıfının interrupt() yöntemi, mevcut threadi veya başka bir threadi kesmek için kullanılır. Bu, mevcut threadin hemen durmasını sağlamaz; mevcut işini sürdürmesi için nazik bir istekte bulunma yoludur. Bu nedenle, kesme çağrısının etkisini hemen görmeyebiliriz.

Başlangıçta, bir threadin bir kesme durumu yoktur ve bu durum false'dur. Yani, interrupt() yöntemi çağrıldığında, bu durum true olarak ayarlanır. Bu, mevcut threadin çalışmasını devam ettirir ve hemen etkisi olmaz.

Eğer bir thread uyku veya bekleme durumunda ise (yani, thread wait() veya sleep() yöntemini çalıştırmışsa) ve kesilirse, thread yaptığı işi durdurur ve bir kesme istisnası fırlatır. Bu nedenle, kesinti istisnasını throws veya try/catch bloğu ile ele almalıyız.

1. **Threadler arası iletişimi ve bunun Java'da nasıl gerçekleştiğini açıklayın?**

Threadler genellikle farklı ve ilgisiz görevleri gerçekleştirmek üzere oluşturulur, ancak ilişkili görevler gerçekleştirmeleri gereken durumlar olabilir. Java'da threadler arasındaki iletişim, aşağıdaki üç yöntemin yardımıyla gerçekleştirilir:

wait():

notify():

notifyAll():

1. **Java’da wait(), notify() ve notifyAll() yöntemlerini açıklayın?**

wait(), notify() ve notifyAll() yöntemleri, object classında bulunur ve threadleri arasında iletişimi ve işbirliğini sağlamak için kullanılır. Bu yöntemler, senkronize bağlamda yani synchronized bloklarda veya synchronized metotlarda kullanılır.

wait(): wait() yöntemi, mevcut threadi uyutur ve kilidi serbest bırakır. Başka bir thread kilidi alıp notify() çağrısı yapana kadar mevcut thread uyumaya devam eder.

notify(): notify() yöntemi, aynı nesne üzerinde wait() çağrısı yapan bir threadi uyandırır. Ancak, hangi threadin uyandırılacağı belirsizdir. Sistem tarafından seçilen bir thread uyandırılır.

notifyAll(): notifyAll() yöntemi, aynı nesne üzerinde wait() çağrısı yapan tüm threadleri uyandırır. Tüm uyuyan threadler uyandırılır ve rekabetçi uyandırma sorunları engellenir.

Bu yöntemler, threadler arasında iletişimi sağlar ve senkronize bloklar arasındaki işbirliğini kolaylaştırır. InterruptedException gibi bazı istisnaları yönetmek için bu yöntemler try-catch blokları içinde veya throws ifadesi ile ele alınmalıdır.

1. **wait(), notify() ve notifyAll() yöntemlerinin neden thread sınıfında değil de Object sınıfında olduğunu açıklayınız?**

wait(), notify() ve notifyAll() yöntemleri, nesne seviyesindeki işbirliğini ve senkronizasyonu sağlamak için kullanılır. Bu yöntemler, bir nesnenin üzerinde çalışır ve bu nedenle Object sınıfında bulunurlar. Bunun nedeni, bir nesne üzerinde senkronizasyon yapmak ve bekletme, uyandırma ve tüm threadleri uyandırma gibi işlemleri nesne seviyesinde yönetmek için kullanılırlar.

Thread sınıfı, threadleri temsil etmek ve threadlerle ilgili yönetimi sağlamak için kullanılır. Ancak, wait(), notify() ve notifyAll() gibi senkronizasyon işlemleri, threadlerin üzerinde çalıştığı nesnelerle doğrudan ilgilidir ve threadlerin kontrolüne özgü değildir. Bu nedenle, bu yöntemler Object sınıfında tanımlanmıştır ve herhangi bir nesne üzerinde çağrılabilirler.

Özetle, wait(), notify() ve notifyAll() gibi senkronizasyon yöntemleri nesne seviyesinde işbirliği ve senkronizasyonu sağladığından, bunlar Object sınıfında bulunur ve herhangi bir nesne üzerinde kullanılabilir. Bu, threadlerin belirli nesnelerle ilişkili senkronizasyon ve işbirliği sağlama ihtiyacını yansıtır.

1. **IllegalMonitorStateException'ı ve ne zaman atılacağını açıklayın?**

IllegalMonitorStateException, wait(), notify() ve notifyAll() gibi senkronizasyon yöntemleri (synchronized methods veya synchronized blocks) dışında çağrıldığında ortaya çıkar. Bu durumda, senkronizasyon mekanizmaları doğru şekilde kullanılmamış demektir ve bu nedenle IllegalMonitorStateException istisnası alınır.

Bir threadğının wait(), notify() veya notifyAll() yöntemlerini çağırabilmesi için, bu işlemlerin bir senkronizasyon bağlamında (synchronized context) gerçekleştirilmesi gerekir. Bu, bir nesne üzerinde bir synchronized blok veya bir synchronized metot içinde gerçekleşmelidir. Aksi takdirde, IllegalMonitorStateException istisnası alınır.

Bu istisna, senkronizasyon mekanizmalarının doğru şekilde kullanılmasını teşvik etmek ve senkronizasyonun güvenli bir şekilde gerçekleştirilmesini sağlamak için Java'da bulunur. Threadler arasındaki senkronizasyonun güvenli ve tutarlı bir şekilde yapılabilmesi için bu kurala dikkat edilmelidir.

1. **wait(), notify(), notifyAll() yöntemleri çağrıldığında kilidi serbest bırakıyor mu yoksa bekletiyor mu?**

wait(), notify() ve notifyAll() yöntemleri her zaman senkronize bağlamda (synchronized context) çağrılır. Bu yöntemler bir nesne üzerinde senkronizasyon yapmak için kullanıldığından, bunlar senkronize bir blok veya metot içinde gerçekleşmelidir.

Bu yöntemler çağrıldığında, iş parçacığı mevcut nesne üzerindeki kiliti edinir. Daha sonra, wait(), notify() veya notifyAll() yöntemi çağrıldığında, iş parçacığı bu kilidi bırakır (unlock). wait() yöntemi çağrıldığında, iş parçacığı kilidi bırakır ve beklemeye alınır, diğer iş parçacıklarının bu kilidi alması ve çalışmaya devam etmesine izin verilir. notify() veya notifyAll() yöntemi çağrıldığında, iş parçacığı kilidi bırakır ve diğer iş parçacıklarının bu nesneye erişmesini ve çalışmasını sağlar.

Yani, bu yöntemler çağrıldığında, iş parçacığı önce kilidi alır ve sonra bu yöntemlerden biri çağrıldığında bu kilidi bırakır. Bu, senkronizasyon mekanizmasının düzgün çalışmasını sağlar ve iş parçacıklarının kritik bölümleri güvenli bir şekilde paylaşmasına olanak tanır.

1. **Aşağıdaki yöntemlerden kilidi serbest bırakma durumlarını açıklayınız.**

**yield(),join(),sleep(),wait(),notify(), notifyAll()**

yield(): Hayır. yield() yöntemi, sadece mevcut iş parçacığının çalışmasını duraklatır ve diğer iş parçacıklarına öncelik vermek için CPU'ya geri teslim eder, ancak kilidi serbest bırakmaz.

sleep(): Hayır. sleep() yöntemi, belirtilen süre boyunca iş parçacığının uyumasını sağlar ancak kilidi serbest bırakmaz.

join(): Hayır. join() yöntemi, çağrılan iş parçacığının tamamlanmasını beklerken mevcut iş parçacığının beklemesini sağlar, ancak kilidi serbest bırakmaz.

wait(): Evet. wait() yöntemi, iş parçacığının belirli bir koşulun gerçekleşmesini beklemesini sağlar ve kilidi serbest bırakır. İş parçacığı, notify() veya notifyAll() çağrılarıyla uyandırıldığında, tekrar çalışmaya başlamadan önce kilidi yeniden almaya çalışır.

notify(): Evet. notify() yöntemi, wait() yöntemiyle bekleyen bir iş parçacığını uyandırır ve kilidi serbest bırakır.

notifyAll(): Evet. notifyAll() yöntemi, wait() yöntemiyle bekleyen tüm iş parçacıklarını uyandırır ve kilidi serbest bırakır.

Bu yöntemlerin kilidi serbest bırakıp bırakmadığını anlamak önemlidir çünkü senkronizasyon ve iş parçacıkları arası iletişimde doğru davranışı sağlamak için kullanılırlar.

1. **Thread local değişkenlerini açıklayınız.**

Thread local değişkenler, bir nesneye değil, belirli bir threade bağlı değişkenlerdir. Bir sınıfta ThreadLocal nesnesini private static olarak bildiririz. Yeni bir threadi nesneye getter veya setter kullanarak eriştiğinde, nesnenin bir kopyasına erişiriz. Bir thread, ThreadLocal örneğinin get veya set yöntemini çağırdığında, belirli bir nesneye yeni bir kopya ilişkilendirilir.

1. **Java da daemon threads nedir?**

Daemon threads arka planda çalışan iş parçacıklarıdır. Bunlar, diğer iş parçacıklarının yararına çalışan hizmet iş parçacıklarıdır. Örneğin, Garbage Collector (Çöp Toplayıcı) daemon iş parçacıklarının iyi bir örneğidir.

Varsayılan olarak, tüm iş parçacıkları daemon değildir. Bir iş parçacığının daemon doğası miras alınabilir. Eğer bir ana iş parçacığı daemon ise, alt iş parçacığı da iş parçacığının daemon doğasını miras alır.

1. **Daemon olmayan bir thread daemon olarak nasıl ayarlanılır?**

Varsayılan olarak, tüm threadler daemon değildir. Bir threadi daemon olarak ayarlamak için setDaemon() yöntemini kullanabiliriz. Burada dikkat etmemiz gereken önemli nokta, setDaemon() yöntemini yalnızca üzerinde start() yöntemi çağrılmadan önce çağırabiliriz. Eğer start() yöntemi çağrıldıktan sonra setDaemon() çağrılırsa, bir IllegalThreadStateException hatası alınacaktır.

1. **Java’da nested class ları açıklar mısın?**

Java'da başka bir sınıf içinde tanımlanan bir sınıf, iç içe sınıf olarak tanımlanır. Java'da iki tür iç içe sınıf bulunur:

Statik iç içe sınıf

Statik olmayan iç içe sınıf

Statik bir iç içe sınıfın tanımı önünde static anahtar kelimesi bulunur.

1. **Java'da inner classes ve non static olmayan nested classes nelerdir?**

Java'da, sınıf tanımında herhangi bir static anahtar kelimesi bulunmayan iç içe sınıflar, statik olmayan iç içe sınıflar olarak tanımlanır. Genellikle statik olmayan iç içe sınıflara iç içe sınıflar denir.

Java'da üç tür iç içe sınıf vardır:

Yerel iç içe sınıf (Local inner class)

Üye iç içe sınıf (Member inner class)

Anonim iç içe sınıf (Anonymous inner class)

1. **Java’da nested classes neden kullanırız?**

Java'da iç içe sınıfları kullanmanın birkaç nedeni vardır:

İlgili sınıfları gruplama:

Yeniden kullanılamayan sınıflar iç içe sınıf olarak tanımlanabilir. Örneğin, bir gönder düğmesine sahip bir formumuz var ve bu düğmeye tıklandığında bazı kodu çalıştırmamız gerekiyor. Bu kod sadece o sınıfla ilgilidir ve başka bir sınıf için yeniden kullanılamaz. Yeni bir sınıf oluşturmak yerine iç içe sınıf oluşturabiliriz.

Kapsüllemeyi artırma: (encapsulation)

İç içe sınıflar, dış sınıfın özel üyelerine erişebilir. Bu sayede dış dünya bu değişkenlere erişmek için get ve set metotları kullanırken, iç içe sınıf bu değişkenlere doğrudan erişebilir.

Kodun okunabilirliği ve bakımı:

Yeni bir sınıf oluşturmak yerine iç içe sınıf oluşturmak, kodun okunabilirliğini artırır ve bakımını kolaylaştırır.

Uygulamanın detaylarını gizleme:

İç içe sınıflar, bir sınıfın uygulama detaylarını gizlememize yardımcı olur. Bu, dış dünyanın iç sınıfın özel yapısını görmesini engeller.

1. **Java'daki statik nested classları açıklayınız?**

Java'da bir iç içe sınıfı (nested class) tanımladığımızda, bu sınıfın statik olması durumunda bunu "static nested class" olarak adlandırırız. Statik iç içe sınıflar, iç içe sınıfların bir alt kümesidir ancak iç içe sınıflar gibi davranmazlar. Statik iç içe sınıflar, dışarıdaki sınıfın örneği olmadan doğrudan oluşturulabilirler. Statik iç içe sınıflar, dışarıdaki sınıfın örneğine erişemez ve dışarıdaki sınıfın örneğine ait olmayan değişkenlere ve yöntemlere erişemezler. Bu nedenle, statik iç içe sınıflar, iç sınıfın özelliklerine sahip değildir ve daha çok dış sınıfın içinde bir tür yardımcı sınıf olarak kullanılırlar.

1. **Java'da statik iç içe geçmiş sınıflar nasıl başlatılır?**

Statik iç içe sınıfları Java'da örneklemek için dış sınıfın bir örneğine ihtiyaç duymadan doğrudan erişebiliriz. Statik iç içe sınıfı örneklemek için aşağıdaki sözdizimini kullanabiliriz:

*DışSınıfAdı.StatikİçİçeSınıfAdı referans = new DışSınıfAdı.StatikİçİçeSınıfAdı();*

Bu ifadede, DışSınıfAdı, statik iç içe sınıfın içinde bulunduğu dış sınıfın adıdır. StatikİçİçeSınıfAdı, örneklemek istediğimiz statik iç içe sınıfın adıdır. Örneğin:

*public class OuterClass {*

*// Static nested class*

*public static class StaticNestedClass {*

*// Fields and methods*

*}*

*}*

*// Instantiate the static nested class*

*OuterClass.StaticNestedClass nestedInstance = new OuterClass.StaticNestedClass();*

1. **Java'da lokal metot inner classları ve local metotları açıklayınız.**

Yerel iç içe sınıflar, bir metot içinde tanımlanan iç içe sınıflardır. Yerel iç içe sınıflar, yalnızca sınıfın tanımlandığı metot içinde nesne oluşturabiliriz. Bir metot çağrıldığında yerel iç içe sınıflar var olur ve metot geri döndüğünde kapsam dışına çıkarlar.

Bu tür sınıflar, sadece belirli bir metotun mantıksal bölümünde kullanılacak işlevselliği içeren durumlar için kullanışlıdır. Özellikle, bir sınıfın yalnızca belirli bir metota özgü bir özellik veya işlevsellik içermesi gerektiğinde bu metota başvurulabilir. Yerel iç içe sınıflar, bu metota özgü olan işlevselliği gruplamak ve izole etmek için kullanılabilir.

Örneğin:

*public class OuterClass {*

*public void someMethod() {*

*// Method local inner class*

*class LocalInnerClass {*

*// Fields and methods*

*}*

*// Instantiate local inner class*

*LocalInnerClass innerInstance = new LocalInnerClass();*

*}*

*}*

Bu örnekte, someMethod() adında bir metodumuz var ve içinde LocalInnerClass adında bir yerel iç içe sınıf bulunmaktadır. LocalInnerClass, yalnızca someMethod() metodu içinde kullanılabilir ve bu metot çağrıldığında oluşturulur. Metot sona erdiğinde, LocalInnerClass'ın kapsamı da sona erer ve sınıfın örneği yok edilir.

1. **Java’da local inner class özelliklerini açıklayınız?**

Yöntem Yerel İç Sınıflar Döngüler (for, while) ve Bloklar (if gibi) İçinde Tanımlanabilir

Yerel iç sınıflar (local inner class), bir yöntem, döngü (örneğin for veya while döngüleri) veya blok (örneğin if veya switch blokları) içinde tanımlanabilir. Bu tür sınıflar sadece tanımlandıkları kapsam içinde geçerlidir ve bu kapsamın dışından erişilemezler. Bu, yerel iç sınıfların daha kapsayıcı bir kontrol sağlar ve kodun daha düzenli ve okunabilir olmasına yardımcı olur.

Örneğin:

*public void myMethod() {*

*int number = 10;*

*if (number > 5) {*

*class LocalInner {*

*void display() {*

*System.out.println("Number is greater than 5");*

*}*

*}*

*LocalInner local = new LocalInner();*

*local.display();*

*}*

*}*

Yukarıdaki örnekte, LocalInner sınıfı if bloğu içinde tanımlanmıştır. Bu sınıf sadece bu if bloğu içinde kullanılabilir ve bu bloğun dışından erişilemez. Bu, kodun belirli bir koşula bağlı olarak yerel iç sınıf tanımlama ve kullanma esnekliği sağlar.

1. **Java’da anonim iç sınıfları açıklayınız?**

Bir sınıf adı olmadan tanımlanan iç sınıfa anonim iç sınıf denir. Anonim iç sınıf, new anahtar kelimesi kullanılarak tanımlanır ve başlatılır. Java'da anonim iç sınıfların ana amacı, bir arayüzü (interface) uygulamaktır. Anonim sınıfları, bir sınıfın yalnızca bir örneğine ihtiyaç duyduğumuzda kullanırız. Anonim iç sınıflar, kapsayan (enclosing) sınıfın tüm üyelerini ve final yerel değişkenleri kullanabilir.

Anonim iç sınıfları derlediğimizde, derleyici iki dosya oluşturur:

EnclosingName.class

EnclosingName$1.class

Bu dosyalar, kapsayan sınıf ve anonim iç sınıfı temsil eder. Anonim iç sınıflar, kodun okunabilirliğini artırır ve küçük sınıfların hızlı bir şekilde tanımlanmasını sağlar. Örneğin:

*public class OuterClass {*

*public void myMethod() {*

*final int number = 10;*

*MyInterface myInterface = new MyInterface() {*

*@Override*

*public void display() {*

*System.out.println("Number is: " + number);*

*}*

*};*

*myInterface.display();*

*}*

*}*

*interface MyInterface {*

*void display();*

*}*

Yukarıdaki örnekte, MyInterface arayüzünü uygulayan anonim bir iç sınıf tanımlanmıştır. Bu anonim sınıf, number değişkenine ve OuterClass içindeki diğer üyelere erişebilir.

1. **Java’da anonim iç sınıf kullanım kısıtlamaları nelerdir?**

1. Anonim iç sınıfın bir adı olmadığı için herhangi bir yapıcı (constructor) yöntemi olamaz.

2. Anonim iç sınıf, statik yöntemler, alanlar veya sınıflar tanımlayamaz.

3. Bir arayüzü (interface) anonim olarak tanımlayamayız.

4. Anonim iç sınıf yalnızca bir kez örneklenebilir.

Bu kısıtlamalar, anonim iç sınıfların kullanımını belirli durumlarla sınırlar ve genellikle küçük, geçici sınıflar oluşturmak için kullanılırlar. Anonim iç sınıflar, adlandırılmamış ve tekrar kullanılmayan kısa süreli görevler için idealdir.

1. **Aşağıdaki kod Java'da geçerli mi? Java'da arayüzü başlatabilir miyiz?**

*Runnable r = new Runnable() {*

*@Override*

*public void run() {*

*}*

*};*

Java'da bir arayüzü doğrudan örneklemek mümkün değildir. Ancak, anonim iç sınıflar kullanarak arayüzleri dolaylı olarak uygulayabilir ve bir arayüzün bir örneğini oluşturabiliriz. Verdiğiniz kod örneğinde;

Runnable bir arayüzdür. Bu kod parçası, sanki Runnable arayüzünü örnekliyor gibi görünebilir, ancak gerçekte olan şey şudur: Anonim bir iç sınıf oluşturuyoruz ve bu iç sınıf Runnable arayüzünü uyguluyor.

Yani, burada yapılan işlem aslında Runnable arayüzünün isimsiz (anonim) bir uygulamasını yaratmaktır. Bu anonim sınıf, Runnable arayüzünün run() metodunu uygulamak zorundadır çünkü Runnable arayüzü bu metodu tanımlar.

Bu şekilde, Runnable arayüzüne uygun bir nesne yaratmış oluruz ve bu nesne, arayüzde tanımlanan metotları kendi içinde barındırır ve tanımlar. Bu nedenle, anonim iç sınıflar kullanarak arayüzleri etkili bir şekilde kullanabiliriz.

Özetle, yukarıdaki kod parçası doğrudur ve çalışır. Burada, Runnable arayüzünü doğrudan örneklemek yerine, bu arayüzü uygulayan anonim bir iç sınıf yaratıyoruz.

1. **Üye iç sınıfları(member inner classes) açıklar mısın?**

Bir dış sınıf içinde tanımlanan statik olmayan sınıflara üye iç sınıf denir. Üye iç sınıf, bir sınıfın üye düzeyinde tanımlanır. Bir üye iç sınıf, dış sınıfın üyelerine, özel üyeler de dahil olmak üzere erişebilir.

Üye iç sınıfların özellikleri:

Bir üye iç sınıf soyut (abstract) veya son (final) olarak bildirilebilir.

Bir üye iç sınıf başka bir sınıfı genişletebilir (extend) veya bir arayüzü (interface) uygulayabilir (implement).

Bir iç sınıf statik alanlar (fields) veya yöntemler (methods) bildiremez.

Bir üye iç sınıf genel (public), özel (private), korumalı (protected) veya varsayılan (default) erişim ile bildirilebilir.

1. **Üye iç sınıfı nasıl örneklendiririz?**

Bir üye iç sınıfı örneklendirmek için dış sınıf referansı kullanmak gerekir. Üye iç sınıfı, dış sınıfın bir örneği olmadan doğrudan örneklendiremeyiz.

Örnek kullanım:

*OuterClassName.InnerClassName inner = new OuterClassReference.new InnerClassName();*

Burada, OuterClassName dış sınıfın adıdır ve InnerClassName iç sınıfın adıdır. OuterClassReference ise dış sınıfın bir örneğidir. Bu şekilde, iç sınıfı oluşturmak için dış sınıfın bir örneğini kullanmış oluruz.

1. **Java'da kapsülleme (encapsulation) nasıl yapılır?**

Java'da kapsülleme (encapsulation) yapmak için şu adımlar izlenebilir:

Instance Değişkenlerini Private Yapma:

Instance değişkenlerini private erişim belirleyicisi ile tanımlayarak sınıf dışından doğrudan erişim engellenir. Bu sayede verilerin güvenliği sağlanmış olur ve dışarıdan yapılacak hatalı erişimlerin önüne geçilmiş olur.

Getter ve Setter Metotlarını Tanımlama:

Instance değişkenlerine erişimi sağlamak için getter ve setter metotlarını tanımlanır. Bu metotlar sayesinde değişkenlere kontrollü bir şekilde erişim ve değişiklik yapılabilir. Getter metotları, değişkenin değerini döndürmek için kullanılırken, setter metotları değişkenin değerini değiştirmek için kullanılır.

Örnek bir Java kodu:

*public class Person {*

*// Instance değişkenleri private yapıyoruz*

*private String name;*

*private int age;*

*// name için getter metodu*

*public String getName() {*

*return name;*

*}*

*// name için setter metodu*

*public void setName(String name) {*

*this.name = name;*

*}*

*// age için getter metodu*

*public int getAge() {*

*return age;*

*}*

*// age için setter metodu*

*public void setAge(int age) {*

*if (age > 0) { // Yaşa pozitif bir değer atanmasını sağlıyoruz*

*this.age = age;*

*}*

*}*

*}*

Bu örnekte, name ve age isimli iki instance değişkeni private olarak tanımlanmıştır. Bu değişkenlere erişmek ve değerlerini değiştirmek için getName, setName, getAge ve setAge metotları tanımlanmıştır. Böylece, Person sınıfının dışından bu değişkenlere doğrudan erişim engellenmiş ve sadece belirli kurallar dahilinde erişim sağlanmıştır. Bu, kapsüllemenin temel prensiplerinden biridir.