**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.