METOTLARA VE SINIFLARA GİRİŞ

Java'nın temelinde sınıf kavramı vardır. Java'nın üzerine inşa edildiği temel mantıksal yapı sınıflardır; çünkü sınıflar, nesnelerin biçimini ve doğasını tanımlar. Nesne yönelimli programlamanın temelini sınıflar oluşturur. Java'da kurulacak yapıların tamamı sınıfların içinde inşa edilmelidir.

Sınıf kavramının temelleri

Şu ana kadar yaptığımız bütün örneklerde sınıfları gördük. Fakat o örneklerde sınıflar yalnızca main() metodunu kapsamak amacıyla kullanılmıştı. Öte yandan, sınıf kavramı bununla sınırlı değildir.

İlk olarak, her bir sınıfın yeni bir veri türü tanımladığını belirterek işe başlamalıyız. Bir kere tanımlandıktan sonra, bu türde yeni nesneler oluşturmak için sınıfları kullanabiliriz. Bu yüzden diyebiliriz ki, sınıflar nesneler için bir **şablon** niteliğindedir.

Bir sınıf oluşturduğumuz zaman aslında yeni bir veri türünü taslak olarak tanımlamış oluruz. Sınıfın içinde tanımladığımız değişkenlere o sınıfın alanları (*fields*) denir. Kodlarımızı ise metotların içine yazarız. Birlikte, metotlar ve alanlar bir sınıfın üyelerini (*members*) oluşturur.

Sınıfın örneği alınarak yeni bir nesne oluşturulduğunda, sınıfın sahip olduğu alanların bir kopyası nesnede tutulur. Türleri aynı olsa da her bir nesnenin alanları birbirinden bağımsızdır.

Basit bir sınıf örneği

Basit bir sınıf tanımlayarak işe başlayalım.

```
class Box
{
      double width;
      double height;
      double depth;
}
```

Yukarıda, *Box* adında yeni bir sınıf tanımladık. Bu sınıfın 3 alanı vardır: genişlik, yükseklik ve derinlik. Her yeni sınıfın yeni bir tür tanımladığını daha önce belirtmiştik. Fakat unutmayın, sınıf tanımlamak yeni bir nesne oluşturmaz, sadece nesne şablonu belirtmiş olur.

Sınıfın örneğini alarak yeni bir nesne oluşturmak için **new** deyimini kullanırız:

```
Box myNewBox = new Box();
```

Burada *Box* türünde yeni bir nesne oluşturduk ve bu nesneyi yine *Box* türünde bir değişkene atadık. Bu kodu çalıştırdığımız zaman, *Box* şablonuna uygun olarak bir nesne oluşturulur ve hafızaya kaydedilir. Burada sınıf ile nesne kavramları arasındaki farkı iyi anlayalım: *Box* bir sınıf, *myNewBox* ise bu sınıfa göre oluşturulmuş bir nesnedir.

Her nesnenin alanlarının birbirinden bağımsız olduğunu söylemiştik. Şimdi bunu inceleyelim:

```
Box box1 = new Box();
Box box2 = new Box();

box1.width = 100.0;
box2.width = 150.0;
```

Burada *Box* türünde iki farklı nesne oluşturduk ve genişliklerine farklı değerler verdik. Buna göre *box1* nesnesinin genişliği 100 iken, *box2* nesnesininki 150'dir. İki ayrı nesnenin alanları bağımsız olduğu için, birinin alanını değiştirdiğimizde diğeri bundan etkilenmez.

Nesne oluşturmak

Nesne oluşturmak için **new** deyimini kullandığımızı söylemiştik. Şimdi bunu daha ayrıntılı inceleyelim. Nesne oluşturduğumuzda, sınıfın **yapılandırıcıları** (*constructor*) çalıştırılır. Yapılandırıcı, özel bir tür metottur. Nesne oluşturulduğu zaman çalıştırılırlar ve nesneye ilk değerini vermek için kullanılırlar. Biz bir yapılandırıcı yazmazsak, Java derleyicisi derleme aşamasında sınıfa varsayılan yapılandırıcıyı (*default constructor*) ekler.

Yapılandırıcıların yapısı şu şekildedir:

```
[sınıfın ismi] ( [parametreler] )
{
      [yapılandırıcının içeriği]
}
```

Yapılandırıcılar genellikle sınıf oluşturulduğunda ilk değerleri atamak için kullanılırlar. Örneğin aşağıdaki kodu inceleyelim:

```
Box box = new Box();
box.width = 150;
box.height = 100;
box.depth = 200;
```

Burada *Box* türünde yeni bir nesne oluşturuluyor ve ilk değerleri veriliyor. Bunu aşağıdaki şekilde de yapabilirdik:

```
class Box
{

      double width;
      double height;
      double depth;

      Box(double myWidth, double myHeight, double myDepth)
      {
            width = myWidth;
            height = myHeight;
            depth = myDepth;
      }

}
```

Yukarıdaki örnekte, *Box* sınıfı içinde bir yapılandırıcı tanımladık. Bu yapılandırıcı 3 parametre alıyor ve aldığı bu değerlere göre sınıfın alanlarının ilk değerlerini veriyor. Dolayısıyla şimdi, yukarıdaki kodu aşağıdaki gibi yeniden yazabiliriz:

```
Box box = new Box(150, 100, 200);
```

Gördüğünüz gibi, sınıfı oluştururken genişlik, yükseklik ve derinlik değerlerini tek bir satırda verdik. Bunu yapılandırıcı sayesinde yapabildik.

Metotlara giriş

Yazdığımız kodları çalıştırmak için kullandığımız yapılara **metot** denir. Metotların genel yapısı aşağıdaki gibidir:

```
[dönüş türü] [metodun ismi] ( [parametreler] )
{
      [metodun içeriği]
}
```

İlk olarak dönüş türünü inceleyelim. İki farklı tarzda metot vardır:

- **Değer döndüren metotlar:** Bu metotlar çalıştıktan sonra geriye bir değer döndürürler.
- **Değer döndürmeyen metotlar:** Bu metotlar yalnızca iş yapmak amacıyla kullanılırlar, çalıştıktan sonra geriye değer döndürmez.

Metodun döndüreceği verinin türünü *[dönüş türü]* kısmında belirtiriz. Eğer değer döndürmeyen bir metot yazıyorsak, *[dönüş türü]* kısmına **void** yazarız.

Daha sonra metodun ismini belirtiriz. Metot isimlendirme kuralları değişken isimlendirme kurallarıyla aynıdır.

Son olarak parantez içinde metodun aldığı parametreleri belirtiriz. Parametre, metoda verilecek değeri ifade eder. Bir metot bir veya daha fazla parametre alabileceği gibi, hiç parametre almayabilir. Parametresiz bir metot yazıyorsak parantez içini boş bırakırız.

Değer döndüren bir metot yazıyorsak, metodun çalışmasının sonlanacağı her durum için bir değer döndürmeliyiz. Değer döndürmeyi **return** deyimiyle yaparız.

Aşağıda örnek bir metot inceleyelim:

```
int add(int number1, int number2)
{
      int result = number1 + number2;
      return result;
}
```

Bu örnekte toplama işlemi yapan bir metot yazdık. Bu metot parametre olarak 2 sayı alır, bunları toplar ve sonucu döndürür. Yazılan bir metodu aşağıdaki gibi çağırabiliriz:

```
int result = add(5, 3);   // result değişkeninin değeri 8 olur
```

Şimdi, yukarıda oluşturduğumuz *Box* sınıfına bir metot ekleyelim. Bu metodu, kutunun hacmini hesaplamak için kullanacağız:

```
class Box
{

      double width;
      double height;
      double depth;

      Box(double myWidth, double myHeight, double myDepth)
      {
            width = myWidth;
            height = myHeight;
            depth = myDepth;
      }

      void volume()
      {
            System.out.println("Hacmim şudur: ");
            System.out.println(width * height * depth);
      }

}
```

Şimdi, oluşturduğumuz bu metodu kullanarak kutunun hacmini konsola yazdıralım:

```
Box box = new Box(150, 100, 200);
box.volume();    // Konsola 3000000 yazar
```

Bu metot bir değer döndürmediği için, sonucunu bir değişkene atamak istediğimizde hata alırız:

```
Box box = new Box(150, 100, 200);
double volume = box.volume();
// Yukarıdaki satır hataya neden olur; çünkü volume() metodu değer
// döndürmüyor
```

Şimdi, hacim hesaplayan metodumuzu değer döndüren bir metot haline getirelim:

```
class Box
{

	double width;
	double height;
	double depth;

	Box(double myWidth, double myHeight, double myDepth)
	{
		width = myWidth;
		height = myHeight;
		depth = myDepth;
	}

	double volume()
	{
		return width * height * depth;
	}

}
```

Artık metodun sonucunu bir değişkene atayabiliriz:

```
Box box = new Box(150, 100, 200);
double volume = box.volume();
System.out.println(volume);    // Konsola 3000000 yazar
```

Farklı nesnelerin alanlarının birbirinden bağımsız olması gibi, metotları da birbirinden bağımsız çalışır. Örneğin, aşağıdaki kodu inceleyelim:

```
Box box1 = new Box(2, 3, 4);
Box box2 = new Box(5, 6, 7);

System.out.println(box1.volume());    // Konsola 24 yazar
System.out.println(box2.volume());    // Konsola 210 yazar
```

Gördüğünüz gibi, aynı metot farklı nesneler için birbirinden bağımsız çalışır.

Değer döndürmeyen bir metodu sonlandırmak

Varsayılan olarak, bir metodun içeriğindeki bütün kodlar çalışır. Fakat bazı durumlarda metodun çalışmasını manuel olarak durdurmamız gerekebilir. Bu durumda **return** deyimi kullanılır. Aşağıdaki örneği inceleyelim:

```
void multiply(int number1, int number2)
{
	if (number1 == 0 || number2 == 0)
	{
		System.out.println(0);
		return;
	}

	int result = number1 * number2;
	System.out.println(result);
}
```

Yukarıda, çarpma işlemi yapan bir metot yazdık. Bu metot 2 parametre alır ve bu değerleri birbiriyle çarpar. Daha sonra sonucu konsola yazdırır. Gördüğünüz gibi, eğer çarpanlardan biri sıfırsa, konsola sıfır yazıyor ve metodu sonlandırıyoruz. Çarpma işlemini sadece çarpanların ikisi de sıfırdan farklıysa yapıyoruz. Metodumuz değer döndürmediği için, return deyiminden sonra bir değer yazmıyoruz.

Sınıf değişkeninin gizlenmesi (instance variable hiding)

Metot içinde tanımladığımız bir değişkenin veya metodun parametrelerinden birinin ismi, sınıfın içinde tanımlanmış değişkenlerden birinin ismiyle aynıysa sınıfın o alanına erişemez oluruz. Aşağıdaki örneği inceleyelim:

```
class MyClass
{

	int year = 2019;

	void myMethod()
	{
		int year = 2020;
		System.out.println(year);
	}

}
```

```
MyClass myObject = new MyClass();
myObject.myMethod();
```

Yukarıdaki kodu çalıştırdığımızda konsola 2020 yazar; çünkü hem metodun hem de sınıfın içinde *year* isminde değişkenler vardır. Metodun içindeki *year*

değişkeni, sınıfın alanını gizlemektedir. Bu nedenle konsolda 2019 değil, 2020 yazar.

this deyimi

Bazen bir sınıf için kod yazarken, kendi sınıfımıza referans vermemiz gerekir. Örneğin, bir önceki başlıkta incelediğimiz sınıf değişkeninin gizlenmesi durumunda sınıfın alanına erişmek isteyelim. Bu durumlarda **this** deyimini kullanırız:

```
class MyClass
{

	int year = 2019;

	void myMethod()
	{
		int year = 2020;
		System.out.println(this.year);
	}

}
```

```
MyClass myObject = new MyClass();
myObject.myMethod();
```

Yukarıdaki kodu çalıştırdığımızda konsola 2019 yazar; çünkü *year* değişkeninden önce **this** deyimini yazdığımız için metodun içindeki *year* değişkenine değil, sınıfın alanına erişmiş oluruz.

METOTLARA VE SINIFLARA DAHA YAKINDAN BAKIŞ

Yukarıda sınıfları ve metotları kısaca inceledik. Şimdi biraz daha ayrıntıya girelim.

Metotları aşırı yüklemek (overloading methods)

Normal olarak, bir sınıf içinde aynı isme sahip birden fazla metot olamaz. Bu durumun bazı eksileri vardır. Örneğin, toplama işlemi yapan iki metot yazalım. Bu metotlardan birisi 2 parametre alırken diğeri 3 parametre alsın:

```
class Math
{

	int add2(int x, int y)
	{
		return x + y;
	}

	int add3(int x, int y, int z)
	{
		return x + y + z;
	}

}
```

Burada *Math* isminde bir sınıf oluşturduk ve bu sınıfın içine toplama işlemi yapan 2 metot yazdık. Bu metotlara, isimleri aynı olamayacağı için, *add2()* ve *add3()* şeklinde isim vermek zorunda kaldık. Bu yöntemin iyi olmadığını ve kodu karmaşıklaştırdığını söylememize gerek yoktur.

Java'da, bu gibi durumların önüne geçmek için metotları aşırı yükleyebilme özelliği vardır. Aynı işi yapan fakat parametreleri farklı olan metotlara aynı ismi verebiliriz. Bu duruma, **metodu aşırı yüklemek** (*method overloading*) denir. Yukarıdaki kodu aşağıdaki gibi yazabiliriz:

```
class Math
{

	int add(int x, int y)
	{
		return x + y;
	}

	int add(int x, int y, int z)
	{
		return x + y + z;
	}

}
```

Gördüğünüz gibi, metodu aşırı yükleyebilmemiz sayesinde, aynı işi yapan fakat parametreleri farklı olan iki metoda farklı isimler vermek zorunda

kalmadık. Aşırı yüklenmiş metotlar çağrılırken parametre sayısına bakılır ve uygun metot çalıştırılır:

```
Math math = new Math();
math.add(1, 2);       // 2 parametre alan metot çalıştırılır
math.add(1, 2, 3);    // 3 parametre alan metot çalıştırılır
```

Metotları aşırı yükleyebilmek için mutlaka parametrelerinin farklı olması gerekir. **Parametreleri aynı olan birden fazla aşırı yüklenmiş metot olamaz.**

Son olarak şunu belirtelim: aşırı yüklenmiş metotların dönüş türü birbirinden farklı olabilir. Fakat sadece dönüş türünü değiştirmek metodu aşırı yüklemek için yeterli değildir.

Yapılandırıcıları aşırı yüklemek

Yapılandırıcıların (*constructors*) özel bir tür metot olduğunu söylemiştik. Aynı şekilde, metotları aşırı yükleyebildiğimiz gibi yapılandırıcıları da aşırı yükleyebiliriz. Bunu görebilmek için *Box* sınıfımıza geri dönelim:

```
class Box
{

	int width;
	int height;
	int depth;

	Box(int myWidth, int myHeight)
	{
		width = myWidth;
		height = myHeight;
		depth = 0;
	}

	Box(int myWidth, int myHeight, int myDepth)
	{
		width = myWidth;
		height = myHeight;
		depth = myDepth;
	}

}
```

Gördüğünüz gibi, *Box* sınıfının içine 2 farklı yapılandırıcı yazdık. Biri bütün alanlara değer verirken, diğeri yalnızca genişlik ve yüksekliği ayarlamak için kullanılıyor.

```
Box box1 = new Box(10, 15);
Box box2 = new Box(10, 15, 20);

System.out.println("Birinci kutu:");
System.out.println(box1.width);
System.out.println(box1.height);
System.out.println(box1.depth);

System.out.println("İkinci kutu:");
System.out.println(box2.width);
System.out.println(box2.height);
System.out.println(box2.depth);
```

Yukarıdaki kodu çalıştırdığımız zaman çıktısı şu şekilde olur:

```
Birinci kutu:
10
15
0
İkinci kutu:
10
15
20
```

Bir yapılandırıcıdan diğerini çağırmak

Birden fazla yapılandırıcısı olan sınıflarda bir yapılandırıcıda diğer yapılandırıcıyı çağırmak için, daha önce de gördüğümüz **this** deyimini kullanırız. Daha önce this deyimini nesne olarak kullanmıştık. Buradaki kullanımda ise this deyimi yapılandırıcı (metot) halindedir. Aşağıdaki örneği inceleyelim:

```
class Box
{

      int width;
      int height;
      int depth;

      Box(int myWidth, int myHeight)
      {
            this(myWidth, myHeight, 0);
            System.out.println("Diğer yapılandırıcı çağrılıyor...");
      }

      Box(int myWidth, int myHeight, int myDepth)
      {
            width = myWidth;
            height = myHeight;
            depth = myDepth;
      }

}
```

Yukarıda, ilk yapılandırıcıyı çalıştırdığımız zaman diğer yapılandırıcıya gideriz. Bu tarz kullanımlarda dikkat etmemiz gereken önemli bir nokta vardır: this deyimi kullanarak başka bir yapılandırıcıyı çağırıyorsak, bunu ilk satırda yapmalıyız, yani metoda ilk olarak bu satırla başlamalıyız. Örneğin, aşağıdaki kod hataya sebep olur:

```
class Box
{

	int width;
	int height;
	int depth;

	Box(int myWidth, int myHeight)
	{
		System.out.println("Diğer yapılandırıcı çağrılıyor...");
		this(myWidth, myHeight, 0);
		// Yukarıdaki kod hataya sebep olur; çünkü this çağrımından
		// önce başka bir kod yazılmış
	}

	Box(int myWidth, int myHeight, int myDepth)
	{
		width = myWidth;
		height = myHeight;
		depth = myDepth;
	}

}
```

Özyineleme (Recursion)

Bir metodun kendini çağırmasına **özyineleme** (*recursion*) denir. Bu tarz metotlara **özyineli** (*recursive*) metot denir. Özyineli algoritmalara verilebilecek en meşhur örnek faktöriyel hesabıdır. Bir sayının faktöriyeli aşağıdaki gibi hesaplanır:

$$n! = n * (n - 1)!$$

$$f(n) = n * f(n - 1)$$

Gördüğünüz gibi, sayıyı 1 azaltarak yine faktöriyel fonksiyonunu çağırıyoruz. Java'da da bu tarz metotlar yazabiliriz. Özyineli metot yazarken dikkat etmemiz gereken önemli bir nokta, metoda bir çıkış koşulu yazmaktır. Eğer metoda bir çıkış noktası yazmazsak, sonsuza kadar aynı metot çağrılmaya devam eder. Aşağıdaki örneği inceleyelim:

```
int factorial(int number)
{
      return number * factorial(number – 1);
}
```

Burada özyineli bir metot yazdığımızı görebilirsiniz; çünkü metodun içinde yine aynı metodu çağırıyoruz. Fakat bu metoda bir çıkış noktası yazmadığımız için metodumuz sonsuza kadar kendini çağırır ve bir süre sonra *StackOverflowException* hatası alırız. Aşağıda düzgün bir özyineli metot yazdık:

```
int factorial(int number)
{
      if (number == 1)
      {
            return 1;
      }

      return number * factorial(number – 1);
}
```

Yukarıda dikkat etmeniz gereken nokta, özyineli metoda bir çıkış noktası yazmış olmamızdır. Eğer parametre olarak aldığımız sayı 1 ise metodumuz 1 değerini döndürür; çünkü 1'in faktöriyeli 1'dir. Diğer durumlarda ise metodumuz kendini çağırmaya devam eder.

```
System.out.println(factorial(4));    // Konsola 24 yazar
System.out.println(factorial(5));    // Konsola 120 yazar
```

ERİŞİM DENETİMİ

Nesne yönelimli programlamanın 4 temel kavramından biri **kapsülleme**dir (*encapsulation*). Bu kavram basitçe, sınıfın içine yazdığımız alanları ve metotları gizleyebileceğimizi ifade eder. Bu gizleme farklı seviyelerde olabilir. Alanları ve metotları gizleyebilmek için **erişim belirteçl**erini (*access modifiers*) kullanırız.

Erişim belirteçleri (*access modifiers*)

Java'da 4 adet erişim belirteci vardır:

- **public:** Bu erişim belirteciyle belirtilen bir alana veya metoda her yerden erişilebilir. **public** deyimini kullanır. Gizlilik seviyesi en düşük olan erişim belirtecidir.
- **default:** Yalnızca aynı paket içinden erişilebilir. Bu belirteç *default* olarak isimlendirilmesine rağmen, herhangi bir deyim yazılmaz.
- **protected:** Yalnızca aynı paket içinden veya alt sınıflardan erişilebilir. **protected** deyimini kullanır.
- **private:** Yalnızca aynı sınıf içinden erişilebilir. **private** deyimini kullanır. Gizlilik seviyesi en yüksek olan erişim belirtecidir.

Aşağıdaki örnekleri inceleyelim:

```
package package1;

class Box
{

	public int width;
	public int height;
	public int depth;

}
```

```
package package2;

class Main
{

	public static void main(String[] args)
	{
		Box box = new Box();
		box.width = 100;
	}

}
```

Yukarıdaki örnekte *Box* ve *Main* sınıfları farklı paketler içindedir. Buna rağmen, *Box* sınıfının alanları **public** olarak belirtildiği için, farklı pakette olmasına rağmen *Main* sınıfından erişilebilir.

```
package package1;

class Box
{

	int width;
	int height;
	int depth;

}
```

```
package package2;

class Main
{

	public static void main(String[] args)
	{
		Box box = new Box();
		box.width = 100;   // Bu satır hataya sebep olur
	}

}
```

Yukarıdaki örnekte Box sınıfının alanlarına erişim **default** olarak belirlenmiştir, yalnızca **package1** paketinin içinden erişilebilir. Bu yüzden, *Main* sınıfı **package2** paketi içinde olduğu için *Box* sınıfının alanlarına erişmeye çalışıldığında hata fırlatılır.

```
package package1;

class Box
{

	private int width;
	private int height;
	private int depth;

}
```

```
package package1;

class Main
{

	public static void main(String[] args)
	{
		Box box = new Box();
		box.width = 100;   // Bu satır hataya sebep olur
	}

}
```

Yukarıdaki örnekte *Box* ve *Main* sınıfları aynı paket içinde olmalarına rağmen, *Main* sınıfından *Box* sınıfının alanlarına erişilemez. *Box* sınıfının alanları **private** olarak belirtildiği için yalnızca aynı sınıf içinden erişilebilir.

Statik metotlar

Sınıfın içine yazdığımız değişkenler ve metotlar, o sınıfın bir örneği oluşturulduğunda hafızaya yüklenir. Bir metodu çağırabilmeniz için, önce o metodun ait olduğu sınıfın bir örneğini almanız gerekir. Fakat bazı durumlarda sınıftan bağımsız metotlar yazmak isteyebilirsiniz. Bu metotları çağırmak için bütün sınıfın bir örneğini oluşturup hafızaya yüklemek gereksiz olabilir. Bu gibi durumlarda **statik metot** tanımlarız. Statik metotlar, sınıfın içine yazılmasına rağmen sınıftan bağımsız olan ve çalıştırılması için sınıfın örneğinin alınmasına gerek olmayan metotlardır. Her Java programının giriş noktası olan *main()* metodunun da statik bir metot olduğunu hatırlayacaksınız.

Statik metot tanımlarken metodun dönüş tipinden önce **static** deyimini kullanırız.

Statik metotlarla ilgili bazı kısıtlamalar bulunmaktadır:

- Yalnızca aynı sınıf içindeki statik metot ve değişkenlere doğrudan erişebilirler; çünkü statik olmayan alanların hafızada oluşabilmesi için sınıfın bir örneğinin alınması gerekir.
- Statik metotlar **this** ve **super** deyimlerini kullanamazlar; çünkü bu deyimleri kullanabilmek için sınıfın bir örneğinin alınmış olması gerekir.

Şimdi durumu daha iyi kavramak adına bir örnek inceleyelim. *Math* adında matematiksel işlemleri yaptığımız bir sınıfımız olsun. Bu sınıfın içine, iki parametre arasında değeri en küçük olan sayıyı bulan bir metot yazalım:

```
class Math
{

	public int min(int a, int b)
	{
		return a < b ? a : b;
	}

}
```

Yukarıda yazdığımız *min()* metodunu aşağıdaki gibi kullanabiliriz:

```
Math math = new Math();
System.out.println(math.min(3, 5));    // Konsola 3 yazar
```

Gördüğünüz gibi metodu çağırabilmek için sınıfın örneğini almak zorundayız. Fakat hemen fark edeceğiniz gibi, bu metodun Math sınıfıyla herhangi bir bağlantısı yoktur. Sınıfın herhangi bir metoduna veya değişkenine erişmediği gibi, çalışması için bu sınıfın hafızada var olmasına da gerek yoktur. Yine de metodu çağırabilmek için mecburen sınıfın bir örneğini alıyoruz. Şimdi aynı metodu statik olarak tanımlayalım ve aradaki farkı görelim:

```
class Math
{

	public static int min(int a, int b)
	{
		return a < b ? a : b;
	}

}
```

Statik olarak tanımladığımız bu metodu aşağıdaki gibi kullanabiliriz:

```
System.out.println(Math.min(3, 5));    // Konsola 3 yazar
```

Gördüğünüz gibi, statik metodu çağırmak için sınıfın bir örneğini almamıza gerek yoktur. Bu tarz metotları yazmak için statik deyimini sıklıkla kullanacağız. Fakat statik kısıtlamalarına dikkat etmemiz gerekir. Aşağıdaki örnekleri inceleyelim:

```
class Math
{

	private int result;

	public static int min(int a, int b)
	{
		result = a < b ? a : b;	// Bu satır hataya neden olur
		return result;
	}

}
```

Yukarıdaki örnekte *Math* sınıfının içinde *result* adında bir değişken tanımladık. Şimdi metodun içinde iki sayı arasından değeri küçük olanı buluyor ve bu değişkene atıyoruz. Fakat bu satır hataya neden olur; çünkü *result* değişkeni statik olarak belirtilmediği için ancak sınıfın bir örneği alınırsa hafızada bulunabilir. Metodumuz ise statik olduğundan *Math* sınıfının örneğini almaya gerek yoktur. Uzun lafın kısası, *min()* metodumuz çalıştığında hafızada *result* adında bir değişken yoktur.

Statik bir metottan, ancak ve ancak sınıfın diğer statik alanlarına erişilebilir. Yukarıdaki kodu değiştirerek *result* değişkenini de statik olarak belirlersek hata olmayacaktır:

```
class Math
{

	private static int result;

	public static int min(int a, int b)
	{
		result = a < b ? a : b;	// Bu satır hataya neden olmaz
		return result;
	}

}
```

Statik değişkenler

Sınıfın içerisine yazdığımız değişkenleri statik olarak belirleyebiliriz. Statik değişkenleri genelde sınıflarla ilgili sabit değerleri belirlemek için kullanırız.

Statik değişkenler **tek** (*singleton*) nesnelerdir. Bunun anlamı şudur: Java programı çalıştırıldığında statik değişkenler bir kez oluşturulurlar ve program boyunca buradan kullanılırlar. Statik değişkenler, erişim belirteçlerinin izin verdiği ölçüde programın diğer parçaları tarafından kullanılabilirler.

```
class Math
{

	public static final double PI = 3.14;

}
```

Yukarıdaki örnekte *Math* sınıfı içerisinde *PI* adında bir değişken oluşturduk. Matematikteki pi sayısını belirtmek için programımız boyunca bu değişkeni kullanabiliriz; çünkü **static** olarak belirttik. Değişkeni **final** olarak belirttiğimize de dikkat edin; bu sayede bu değişkenin değeri hiçbir şekilde değiştirilemeyecektir. Ayrıca değişkeni **public** olarak tanımladığımız için programımızın her parçasından bu değişkene ulaşılabilecektir. Aşağıdaki örnekte bu değişkene ulaşıp değerini konsola yazdırıyoruz:

```
class Main
{

	public static void main(String[] args)
	{
		System.out.println(Math.PI);	// Konsola 3.14 yazar
	}

}
```

Burada bir not olarak şunu belirtelim: Java'da statik değişkenleri isimlendirmek için genelde büyük harfle isimlendirme (upper case naming) yöntemi kullanılır. Bu yönteme göre değişkenin bütün harfleri büyük olarak yazılır. Eğer değişkenin ismi birden fazla kelimeden oluşuyorsa bu kelimeler alt çizgi (_) ile ayrılır.

Statik değişkenlere ilk değer atamak

Bazen statik değişkenlere ilk değerlerini vermemiz gerekebilir.

Hatırlayacaksanız, sınıfın içindeki değişkenlere ilk değerlerini vermek için yapılandıcıları (constructors) kullanıyorduk. Fakat yapılandırıcılar yalnızca sınıfın bir örneği alındığında çalışır. Statik değişkenler için bu durum söz konusu değildir. Statik değişkenlere ilk değerlerini vermek için **statik başlatma bloklarını** (*static initialization blocks*) kullanırız. Aşağıdaki örneği inceleyelim:

```
class Main
{

	static int[] EVEN_DIGITS = new int[5];

	static
	{
		EVEN_DIGITS[0] = 0;
		EVEN_DIGITS[1] = 2;
		EVEN_DIGITS[2] = 4;
		EVEN_DIGITS[3] = 6;
		EVEN_DIGITS[4] = 8;
	}

}
```

Bu örnekte, çift rakamları tutmak için bir dizi oluşturduk. Bu dizinin elemanlarını atamak için statik başlatma bloğunu nasıl kullandığımıza dikkatinizi çekerim.

İç içe sınıflar (Nested classes)

Şu ana kadar sınıfların içinde değişkenlerin ve metotların bulunabileceğini biliyorduk. Fakat sınıfların gücü bununla sınırlı değildir. Sınıfların içinde başka sınıflar da tanımlayabiliriz. Bu şekilde **iç içe sınıflar** (*nested classes*) oluşturabiliriz.

İç içe tanımlanan sınıfları tabir ederken, diğerini kapsayan sınıfa **dıştaki sınıf** (*outer class*), içeride bulunan sınıfa ise **içteki sınıf** (*inner class*) deriz. Bir sınıf oluşturulduğunda bütün alanlarının ve metotlarının hafızaya yüklendiğini söylemiştik. Bu durum içteki sınıflar için de geçerlidir. İç içe sınıflarda, içteki sınıfı kullanabilmemiz için dıştaki sınıfın bir örneğinin alınması gerekir.

İçteki sınıf, dıştaki sınıfın bütün alanlarına ve metotlarına erişebilir. Dıştaki sınıf tek olmasına rağmen, içteki sınıfın birden fazla örneği alınabilir; bu gibi durumlarda içteki sınıftan oluşturulan nesnelerin hepsi aynı dıştaki sınıfa erişir.

```
class Outer
{

    public void run()
    {
        System.out.println("Dıştaki sınıfın metodu çalıştı.");

        Inner inner = new Inner();
        inner.run();
    }

    class Inner
    {

        public void run()
        {
            System.out.println("İçteki sınıfın metodu çalıştı.");
        }

    }

}
```

Yukarıdaki örnekte iç içe 2 sınıf oluşturduk. Her ikisine de *run()* adında metotlar yazdık.

```
Outer outer = new Outer();
outer.run();
```

Yukarıdaki kodu çalıştırdığımızda çıktısı aşağıdaki gibi olur:

```
Dıştaki sınıfın metodu çalıştı.
İçteki sınıfın metodu çalıştı.
```

Şimdi de aşağıdaki örneği inceleyelim:

```
class Outer
{

    private int number = 10;

    public void run()
    {
        System.out.println(number);

        Inner inner = new Inner();
        inner.run();

        System.out.println(number);
    }

    class Inner
    {

        public void run()
        {
            number++;
        }

    }

}
```

Yukarıdaki örnekte dıştaki sınıfın içinde *number* isminde bir değişken oluşturduk. İçteki sınıfta ise bu değişkene erişip değerini 1 artırdık. Dıştaki sınıfın *run()* metodunu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
10
11
```

Buradan da görebileceğiniz gibi, içteki sınıflar dıştaki sınıfların elemanlarına erişebilir.

İç içe sınıflarla ilgili dikkat etmemiz gereken bazı noktalar vardır. İlk olarak, dıştaki sınıf olmadan içteki sınıfın bir örneğini alamayız. Ayrıca, içteki sınıfın içine statik bir değişken veya metot yazamayız. Bu kısıtlamaların önüne geçmek için içteki sınıfı statik olarak tanımlamalıyız.

İçteki sınıfı statik yapmak

Hatırlarsınız: bir metodu kullanabilmek için tanımlandığı sınıfın bir örneğini almamız gerekiyordu. Bunun önüne geçmek için metodu statik olarak belirtiyorduk. Aynı durum içteki sınıflar için de geçerlidir: bu sınıfları kullanabilmek için dıştaki sınıfın örneğini almamız gerekir. Bunun önüne geçmek için içteki sınıfı statik olarak tanımlamalıyız:

```
class Outer
{

    static class Inner
    {

        public void run()
        {
            System.out.println("İçteki sınıfın metodu çalıştı");
        }

    }

}
```

Şimdi aşağıdaki kodu çalıştırabiliriz:

```
Outer.Inner inner = new Outer.Inner();
inner.run();
```

Gördüğünüz gibi, artık içteki sınıfın bir örneğini alabilmek için dıştaki sınıfa ihtiyacımız yoktur. Fakat, statik metotların sınıfın statik olmayan alanlarına erişemeyeceğini hatırlayın. Aynı durum burada da geçerlidir: içteki statik sınıflar dıştaki sınıfın statik olmayan alanlarına erişemez:

```
class Outer
{

    private int number = 10;

    public void run()
    {
        System.out.println(number);

        Inner inner = new Inner();
        inner.run();

        System.out.println(number);
    }

    static class Inner
    {

        public void run()
        {
            number++;    // Bu satır hataya neden olur
        }

    }

}
```

Yukarıdaki kod hataya neden olur; çünkü statik olarak belirlenmiş içteki *Inner* sınıfı dıştaki sınıfın statik olmayan *number* değişkenine erişmeye çalışıyor.

Değişken sayıdaki metot argümanları (Varargs: variable-length arguments)

Metotların nasıl parametre aldığını daha önce görmüştük. Örneğin aşağıda tanımladığımız toplama metodu 2 parametre almaktadır:

```
public int add(int x, int y)
{
      return x + y;
}
```

Bu metot 2 sayıyı toplar ve sonucunu döndürür. Peki, ya 3 sayıyı toplamak istersek? Java'da metotları aşırı yükleyebildiğimizi görmüştük. Şimdi bu metodu aşağıdaki gibi aşırı yükleyelim:

```
public int add(int x, int y)
{
	return x + y;
}

public int add(int x, int y, int z)
{
	return x + y + z;
}
```

Artık aynı metodun 2 ve 3 parametre alan farklı varyasyonları var. Peki, ya 4 sayıyı toplamak istiyorsak? Aynı şekilde metodu aşırı yüklemeye devam edebiliriz:

```
public int add(int x, int y)
{
	return x + y;
}

public int add(int x, int y, int z)
{
	return x + y + z;
}

public int add(int x, int y, int z, int t)
{
	return x + y + z + t;
}
```

Fakat bu yöntemin pek hoş olmadığı gözünüzden kaçmamıştır. 5 veya 6 sayıyı toplamak istediğimizde metodu tekrar aşırı yüklememiz gerekir. Bu durum kod fazlalığına neden olur. Buradaki sorun, metodun alacağı parametre sayısının belirsiz (değişken) olmasıdır. Toplama işleminde birden fazla toplanan vardır: fakat sayısı değişken olabilir. Java, parametre sayısını önceden bilemeyeceğimiz bu gibi durumlar için metodun **değişken sayıda argüman** (*varargs*) almasına izin verir.

Değişken sayıda parametre tanımlamak için değişken türünden sonra 3 nokta koyarız. Artık, metodu çağırırken istediğimiz sayıda parametre verebiliriz: bunların her biri için metodu aşırı yüklememiz gerekmez. Java bu parametreleri bize bir dizi (*array*) halinde sunar.

```
public int add(int... numbers)
{
    int sum = 0;
    for (int number : numbers)
    {
        sum += number;
    }
    return sum;
}
```

Yukarıda *add()* metodumuzu yeniden düzenledik ve toplama işlemine giren eleman sayısının değişken olmasını sağladık. Artık metoda parametre olarak farklı sayıda toplanan verebiliriz:

```
System.out.println(add(2, 3));              // Konsola 5 yazar
System.out.println(add(1, 5, 7));           // Konsola 13 yazar
System.out.println(add(9));                 // Konsola 9 yazar
System.out.println(add(10, 15, 20, 25));    // Konsola 70 yazar
```

Değişken sayıdaki argümanlar bir dizi halinde sunulur. Dizinin elemanı olmayabilir. Yani, hiç argüman vermesek de değişken sayıda argüman alan metot yine çalışır.

```
System.out.println(add());    // Konsola 0 yazar
```

Yukarıda hiç argüman vermememize rağmen metodun yine çalıştığını görürüz. Değişken sayıda argüman alan metot yazarken bunu göz önünde bulundurmak gerekir.

Değişken sayıda parametre alan metotları aşırı yükleyebilirsiniz. Aşağıdaki örneği inceleyelim:

```
public int add(int... numbers)
{
    int sum = 0;
    for (int number : numbers)
    {
        sum += number;
    }
    return sum;
}

public double add(double... numbers)
{
    double sum = 0.0;
    for (double number : numbers)
    {
        sum += number;
    }
    return sum;
}
```

```
System.out.println(add(5, 4, 3));          // Konsola 12 yazar
System.out.println(add(9.0, 8.5, 8.0));    // Konsola 25.5 yazar
```

Varargs ve belirsizlik (Ambiguity)

Değişken sayıda parametre (varargs) alan metotları aşırı yüklediğinizde ortaya belirsizlik çıkabilir. Yukarıdaki örnekte 2 farklı *add()* metodu oluşturmuştuk. Bu durumda aşağıdaki kodu çalıştırdığımızı düşünelim:

```
add();
```

Bu satır belirsizliğe neden olur; çünkü metoda parametre vermedik. Bu durumda Java ortamı aşırı yüklenmiş 2 metottan hangisini çalıştıracağına karar veremez; çünkü bu kullanım her ikisine de uygundur.

Başka bir belirsizlik örneği inceleyelim:

```
public int add(int... numbers)
{
    // metodun içeriği
}

public int add(int number, int... numbers)
{
    // metodun içeriği
}
```

İki farklı *add()* metodu tanımladık. Şimdi aşağıdaki kodu çalıştıralım:

```
add(1);
```

Bu kod belirsizliğe sebep olur; çünkü metodun her 2 tanımına da uygundur. Dolayısıyla Java ortamı hangisini çalıştıracağına karar veremez.

KALITIM – MİRAS ALIP VERME (INHERITANCE)

Kalıtım, nesne yönelimli programlamanın 4 temel kavramından biridir. Kod tekrarını önlemek için dile eklenen bu özellik bize birçok fayda sağlar. Şimdi kalıtıma neden ihtiyaç duyduğumuzu anlamak için bir örnek yapalım.

2 boyutlu koordinat düzlemindeki noktaları temsil etmek için **Point2D** adında bir sınıf oluşturalım:

```
public class Point2D
{

    private int x;
    private int y;

    public Point2D()
    {
    }

    public int getX()
    {
        return x;
    }

    public void setX(int x)
    {
        this.x = x;
    }

    public int getY()
    {
        return y;
    }

    public void setY(int y)
    {
        this.y = y;
    }

    public double distance()
    {
        return Math.sqrt(x * x + y * y);
    }

}
```

Sınıfımızın 2 alanı vardır. Bu alanları noktanın x ve y düzlemindeki değerlerini belirtmek için kullanacağız. Bu değişkenleri *private* olarak belirledik. Diğer yandan bu değerleri değiştirebilmek amacıyla *getter* ve *setter* metotlar tanımladık. Son olarak bir *distance()* metodu tanımladık. Bu metodu, noktanın orijine, yani (0, 0) noktasına olan uzaklığını hesaplamak için kullanacağız.

Şimdi bu sınıfı bir örnekte kullanalım:

```
Point2D point2D = new Point2D();           // Yeni bir nokta oluşturduk
point2D.setX(3);                           // x değerini 3 olarak atadık
point2D.setY(4);                           // y değerini 4 olarak atadık
System.out.println(point2D.distance());    // Konsola 5.0 yazar
```

Şimdi de 3 boyutlu koordinat düzlemindeki noktaları temsil etmek için **Point3D** sınıfını oluşturalım:

```
public class Point3D
{

    private int x;
    private int y;
    private int z;

    public Point3D()
    {
    }

    public int getX()
    {
        return x;
    }

    public void setX(int x)
    {
        this.x = x;
    }

    public int getY()
    {
        return y;
    }

    public void setY(int y)
    {
        this.y = y;
    }

    public int getZ()
    {
        return z;
    }

    public void setZ(int z)
    {
        this.z = z;
    }

    public double distance()
    {
        return Math.sqrt(x * x + y * y + z * z);
    }

}
```

Gördüğünüz gibi, iki sınıf temelde birbirine benziyor. Point3D sınıfının, Point2D sınıfının bütün alanlarına sahip olduğunu görüyoruz. Buna ek olarak bir **z değişkeni** ve bu değişkenin getter/setter metotları yazılmıştır. Uzaklık hesaplayan *distance()* metodunun aynı şekilde var olduğunu; yalnızca işleyişinin değiştiğini görüyoruz (çünkü uzayda yeni bir nokta ekledik). Point3D sınıfının Point2D sınıfından farklı olan kodlarını kalın olarak gösterdik.

Her iki sınıfı incelediğinizde bazı kodların gereksiz olduğunu görürsünüz. Her iki sınıfta da x ve y değişkenleri tanımlanmıştır. Ayrıca bu değişkenler için getter/setter metotları da yazılmıştır. Bunların her iki sınıfta da aynı olduğunu, aralarında bir kod farklılığı olmadığını görüyoruz. Uzaklık hesaplayan *distance()* metodu ise şeklen aynı, işleyiş olarak farklıdır.

Programcılığın altın kuralı şudur: kod tekrarı kötüdür ve bundan kaçınmalıyız. Point3D sınıfındaki x ve y değişkenleri ve bunların getter/setter metotları kod tekrarından ibarettir. Peki, bundan kaçınmamızın bir yolu var mıdır? Point3D sınıfının, zaten var olan değişken ve metotları Point2D sınıfından almasını sağlayabilir miyiz?

Nesne yönelimli programlama dillerinde var olan kalıtım özelliği sayesinde bu tarz kod tekrarlarından kaçınabiliriz. Kalıtım, bir sınıfın başka bir sınıfın alanlarını miras almasını sağlar. Bu sayede miras alan sınıf, miras veren sınıfın değişken ve metotlarını tekrar yazmak zorunda kalmaz.

Kalıtım sayesinde sınıflar arasında bir alt/üst ilişkisi, yani hiyerarşik bir yapı oluşur. Miras veren sınıfa **üst sınıf** (*super-class*), miras alan sınıfa ise **alt sınıf** (*sub-class*) denir. Bu alt/üst ilişkisi aynı zamanda, nesne yönelimli programlamanın bir diğer kavramı olan **çokbiçimliliği** (*polymorphism*) sağlar. Alt sınıflar, üst sınıflarının bütün alanlarına sahip olduğu için; her bir alt sınıfı bir üst sınıfı cinsinden ifade edebiliriz. Bunu daha iyi anlayabilmek için, canlıların sınıflandırılmasını örnek olarak verebiliriz.

Aslan türünün sınıflandırılması yukarıdan aşağıya (alemden türe) doğru şu şekildedir:

- Hayvanlar (animalia)
- Kordalılar (chordata)

- Memeliler (mammalia)
- Etçiller (carnivora)
- Kedimsiler (feliformia)
- Kedigiller (felidae)
- Büyük kediler (pantherinae)
- Panthera
- Panthera leo

Bu sınıflandırmadan şunu çıkarabiliriz: etçiller, memelilerin bir alt sınıfıdır. Etçiller, memelilerin her özelliğine sahiptir. Dolayısıyla her etçil aynı zamanda bir memelidir. Fakat bunun tam tersi geçerli değildir: her memeli aynı zamanda bir etçil olmayabilir; çünkü memelilerin etçiller dışında başka alt sınıfları da olabilir.

Java sınıfları arasındaki kalıtım da benzer şekilde işler. Her alt sınıf aynı zamanda bir üst sınıf cinsinden ifade edilebilir; çünkü üst sınıfın bütün özelliklerine sahiptir. Bu sayede çokbiçimlilik sağlanmış olur.

Java kalıtımını insanlar arasındaki baba oğul ilişkisine de benzetebiliriz. Her insanın bir babası vardır. Aynı şekilde her Java sınıfının bir üst sınıfı vardır. Diğer yandan, herkesin çocuğu olmayabilir. Aynı şekilde, her Java sınıfının bir alt sınıfı olmayabilir.

İnsanların biyolojik olarak birden fazla babası olamaz. Aynı şekilde, Java sınıflarının da birden fazla üst sınıfı olamaz. Bu duruma **tekli kalıtım** (*single inheritance*) denir. Bir sınıfın birden fazla sınıftan kalıtım alması durumuna **çoklu kalıtım** (*multiple inheritance*) denir. Çoklu kalıtım destekleyen başka programlama dilleri vardır; fakat Java çoklu kalıtımı desteklemez.

Her insanın bir babası vardır demiştik. Geçmişe doğru gidildiğinde herkesin bir soy ağacı çıkar. Fakat bunun bir başlangıç noktasının olması gerekir. Herkesin babasının olması geçmişe doğru sonsuza kadar sürdürülemeyeceği için, bir noktada babası olmayan bir insanın bulunması gerekir. Çoğu mitolojiye göre bu kişi, insanların atası olan Adem'dir. Adem'in babası yoktur, diğer yandan bütün insanlar Adem'den türemiştir. Dolayısıyla bütün insanlar Adem'in çocuklarıdır.

Java'da da buna benzer bir durum söz konusudur. Her sınıfın bir üst sınıfı vardır; fakat bunun bir istisnası vardır. İnsanlarda söz konusu olduğu gibi, Java sınıflarında da bir başlangıç noktasının olması gerekir.

Java'da diğer bütün sınıfların atası olan sınıf **Object** sınıfıdır. Diğer bir deyişle, Java'daki her sınıfı *Object* türünden ifade edebiliriz. Java'nın tasarımcıları, nesne yönelimli bir dil olduğu mesajını vermek için, sınıf hiyerarşisinde en üstte yer alan bu sınıfa *object* (nesne) ismini vermişlerdir. *Object* sınıfının bir üst sınıfı yoktur; fakat doğrudan veya dolaylı olarak, Java'da yazılan her sınıf *Object* sınıfından kalıtım alır. Dolayısıyla bütün sınıflar *Object* sınıfının metotlarına sahiptir. Bu metotları daha sonra inceleyeceğiz.

Kalıtımla ilgili temel kavramları yukarıda kısaca anlattık. Şimdi örneğimize geri dönelim ve Point2D ile Point3D sınıfları arasında bir miras ilişkisi oluşturalım. Daha önce de anlattığımız gibi, Point3D sınıfı Point2D sınıfının bütün özelliklerine sahip olmakla birlikte, Point2D sınıfında olmayan bazı yeni özellikler getirmiş veya Point2D sınıfındaki bazı özellikleri (*distance()* metodunun işleyişi gibi) değiştirmiştir. Buna bilgisayar bilimlerinde **genişletme** (*extension*) denir. Yani, Point3D sınıfı Point2D sınıfını genişletmektedir. Buradan da anlaşılacağı üzere, Point2D üst sınıf, Point3D alt sınıf olacaktır; çünkü alt sınıflar üst sınıfları genişletir.

Java'da sınıflar arasındaki kalıtım **extends** deyimiyle kurulur. Bu deyim alt sınıfları tanımlarken kullanılır ve yapısı şu şekildedir:

```
class [alt sınıfın ismi] extends [üst sınıfın ismi]
```

Şimdi kalıtım kurallarına göre Point3D sınıfını tekrar yazalım:

```
public class Point3D extends Point2D
{

    private int z;

    public Point3D()
    {
    }

    public int getZ()
    {
        return z;
    }

    public void setZ(int z)
    {
        this.z = z;
    }

}
```

Gördüğünüz gibi, Point3D sınıfını yazarken Point2D sınıfını genişlettik ve zaten Point2D sınıfında mevcut olan alanları tekrar yazmadık. Bu sayede kod tekrarından kurtulduğumuz gibi, iki sınıf arasında alt/üst ilişkisi kurmuş olduk.

Şimdi aşağıdaki kodu çalıştırdığımızda Point3D sınıfının Point2D sınıfının metotlarına sahip olduğunu görebiliriz:

```
Point3D point3D = new Point3D();
point3D.setX(3);
point3D.setY(4);
point3D.setZ(5);

System.out.println(point3D.getX());    // Konsola 3 yazar
System.out.println(point3D.getY());    // Konsola 4 yazar
System.out.println(point3D.getZ());    // Konsola 5 yazar
```

Alt sınıfların, kalıtım aldıkları üst sınıfın bütün alanlarına erişebildiğini söylemiştik. Bunun tek bir istisnası vardır. Bazı durumlarda, üst sınıfın bazı üyelerini alt sınıflardan gizlemek isteyebiliriz. Bunun için, gizlemek istediğimiz üyeyi **private** belirteciyle tanımlamamız yeterli olacaktır. Yukarıdaki örnekte, Point2D sınıfının x ve y değişkenlerini private olarak belirttiğimiz için Point3D sınıfından bu değişkenlere erişemeyiz:

```
public class Point3D extends Point2D
{

    public Point3D()
    {
        x = 0;    // Bu satır hataya neden olur
    }

    // sınıfın diğer üyeleri

}
```

Bu örnekte Point3D sınıfının yapılandırıcısı içinde x değişkenine değer vermeye çalışıyoruz. Fakat x değişkeni Point2D sınıfı içinde *private* olarak tanımlandığı için burada erişemeyiz. Dolayısıyla bu satır hataya neden olur.

Sınıf üyelerini dış dünyadan açmadan alt sınıflardan erişilebilir hale getirmek için **protected** erişim belirtecini kullanırız. Şimdi Point2D sınıfını yeniden düzenleyelim, x ve y değişkenlerini *protected* olarak belirtelim:

```
public class Point2D
{

    protected int x;
    protected int y;

    // sınıfın diğer üyeleri

}
```

Artık yukarıdaki kodu tekrar çalıştırırsak hataya neden olmaz; çünkü x değişkenini protected olarak belirttiğimiz için alt sınıflardan erişebiliriz.

```
public class Point3D extends Point2D
{

    public Point3D()
    {
        x = 0;    // Bu satır hataya neden olmaz
        y = 0;    // Bu satır hataya neden olmaz
    }

    // sınıfın diğer üyeleri

}
```

Kalıtım ve tür dönüşümü

Tür dönüşümünü daha önce ilkel veri türlerini anlatırken görmüştük. Benzer bir durum kalıtımda da geçerlidir. Birbirinden miras alan sınıflar arasında tür dönüşümü yapmak mümkündür.

Alt sınıflar üst sınıfın bütün üyelerine sahip olur. Dolayısıyla her alt sınıf bir üst sınıfı cinsinden ifade edilebilir. Bu nedenle, alt sınıftan üst sınıfa doğru olan tür dönüşümüne dolaylı tür dönüşümü denir. Diğer yandan, bir üst sınıfın birden fazla alt sınıfı olabilir, bu yüzden üst sınıftan alt sınıfa doğru yapılan tür dönüşümüne doğrudan tür dönüşümü denir; çünkü dönüşüm yapılacak türü belirtmek gerekir.

Aşağıdaki örneği inceleyelim:

```
class A
{
}

class B extends A
{
}
```

```
B b = new B();

A implicit = b;                // Dolaylı tür dönüşümü yapılıyor
B explicit = (B) implicit;     // Doğrudan tür dönüşümü yapılıyor
```

super deyimi

Daha önce bir sınıfın kendisine referans vermek için **this** deyimini kullanmamız gerektiğini görmüştük. Şimdi benzer bir durumdan bahsedelim. Bazı durumlarda alt sınıf içinde üst sınıfa referans vermek isteyebiliriz. Bu gibi durumlarda **super** deyimini kullanırız. Bu deyim üst sınıfı ifade etmek için kullanılır.

Üst sınıfın yapılandırıcısını çağırmak

Bir alt sınıfın örneği alındığı zaman önce üst sınıfının yapılandırıcısı çağrılır. Eğer üst sınıfın yapılandırıcısı parametre alıyorsa, alt sınıfı oluştururken üst sınıfın yapılandırıcısını çağırmak gerekir. Bu çağrım super deyimiyle yapılır. Örneğin, *Point2D* sınıfımızı değiştirelim ve parametreli bir yapılandırıcıya sahip olacak şekilde kodlayalım:

```
public class Point2D
{

    private int x;
    private int y;

    public Point2D(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    // sınıfın diğer üyeleri

}
```

Gördüğünüz gibi, *Point2D* sınıfına parametreli bir yapılandırıcı yazdık. Bu sınıfta başka bir yapılandırıcı oluşturmadık. Şimdi alt sınıfın yapılandırıcısında bunu çağırmamız gerekir:

```
public class Point3D extends Point2D
{

    private int z;

    public Point3D(int x, int y, int z)
    {
        super(x, y);
        // Üst sınıfın 2 parametreli yapılandırıcısı çağrılıyor
        thiz.z = z;
    }

    // sınıfın diğer üyeleri

}
```

Gördüğünüz gibi, artık *Point3D* sınıfını oluştururken bir üst sınıfın yapılandırıcısını da çağırmak zorundayız. Bunu super deyimiyle yaptık. Burada super deyiminin bir yapılandırıcıya referans ettiğine dikkatinizi çekerim.

Üst sınıfın üyelerini çağırmak

Bazen alt sınıflarda, üst sınıfındaki bir değişkenle aynı isme sahip başka bir değişken olabilir. Bu durumda üst sınıftaki değişken gizlenmiş olur. Bu tarz durumlarda üst sınıftaki değişkene erişmek için super deyimini kullanırız. Örneğin, *Point3D* sınıfının içinde de *x* adında bir değişken olduğunu düşünelim:

```
public class Point3D extends Point2D
{

    private String x;

    public Point3D()
    {
        x = “Hello, World!”;
        // Bu sınıfın x değişkenine erişiliyor
    }

    // sınıfın diğer üyeleri

}
```

```
public class Point3D extends Point2D
{

    private String x;

    public Point3D()
    {
        super.x = 2020;
        // Üst sınıfın x değişkenine erişiliyor
    }

    // sınıfın diğer üyeleri

}
```

Burada super deyiminin bir sınıfa (nesneye) referans ettiğine dikkatinizi çekerim. Ayrıca not olarak şunu belirtelim: bu şekilde üst sınıfın bir üyesine erişmek için üyenin alt sınıflardan erişilebilir olması gerekir. Yani, üst sınıfta private olarak tanımlanmış bir değişkene alt sınıflardan super deyimiyle erişemeyiz.

Yapılandırıcılar hangi sırayla çalıştırılır?

Alt sınıfın bir örneği oluşturulduğunda, üst sınıfların yapılandırıcılarının da çağrıldığını yukarıda belirtmiştik. Peki bu çağrım hangi sırayla olur? Bunun için aşağıdaki örneği inceleyelim:

```
class A
{

    public A()
    {
        System.out.println("A'nın yapılandırıcısı çalıştı");
    }

}

class B extends A
{

    public B()
    {
        System.out.println("B'nin yapılandırıcısı çalıştı");
    }

}

class C extends B
{

    public C()
    {
        System.out.println("C'nin yapılandırıcısı çalıştı");
    }

}
```

Bu örnekte 3 sınıf oluşturduk. Bu sınıfın hiyerarşik yapısı A > B > C şeklindedir. En üstte A sınıfı bulunurken, en altta C sınıfı vardır. C sınıfının bir örneğini aldığınızda çıktısı şu şekilde olur:

```
A'nın yapılandırıcısı çalıştı
B'nin yapılandırıcısı çalıştı
C'nin yapılandırıcısı çalıştı
```

Gördüğünüz gibi, kalıtım alan sınıflarda ilk olarak üst sınıfın yapılandırıcısı çalışır.

Metot ezme (Method overriding)

Nesne yönelimli programlarda kalıtım kavramının en önemli ayaklarından biri metot ezmedir. Metot ezmeyi anlamak için yukarıdaki *Point2D* ve *Point3D* sınıflarımızı tekrar hatırlayalım:

```
public class Point2D
{

    private int x;
    private int y;

    public Point2D()
    {
    }

    public int getX()
    {
        return x;
    }

    public void setX(int x)
    {
        this.x = x;
    }

    public int getY()
    {
        return y;
    }

    public void setY(int y)
    {
        this.y = y;
    }

    public double distance()
    {
        return Math.sqrt(x * x + y * y);
    }

}
```

```
public class Point3D extends Point2D
{

    private int z;

    public Point3D()
    {
    }

    public int getZ()
    {
        return z;
    }

    public void setZ(int z)
    {
        this.z = z;
    }

}
```

Yukarıdaki örnekte, kalıtım kavramını kullanarak uzaydaki 2 boyutlu ve 3 boyutlu noktaları temsil etmek için güzel bir mimari kurmuştuk. Bu mimariye göre, *Point3D* sınıfı, *Point2D* sınıfından miras alıyor ve bu sınıfın üyelerini tekrar yazmak zorunda kalmıyordu. Miras alan sınıf, miras veren sınıfın private olarak belirtilmemiş bütün üyelerine sahip olur. *Point3D* sınıfının *Point2D* sınıfından miras aldığı üyelerden biri de **distance()** metodudur. Hatırlayacağınız gibi, bu metot, noktanın (0, 0) noktasına olan uzaklığını hesaplıyordu. *Point2D* sınıfı içinde tanımlandığı için bu hesaplamada yalnızca x ve y değerlerini hesaba katıyor. Diğer yandan, *Point3D* sınıfında bu metot yanlış çalışır; çünkü z değerini hesaba katmamaktadır:

```
Point3D point = new Point3D();
point.setX(3);
point.setY(4);
point.setZ(6);

System.out.println(point.distance());    // Konsola 5 yazar
```

Yukarıdaki kodu çalıştırdığınızda, *point* nesnesinin uzaklığı gerçekte 7.81 olmasına rağmen konsolda 5 yazar; *Point3D* sınıfı *distance()* metodunu *Point2D* sınıfından miras almıştır ve bu metot hesaplama yaparken z değerini dikkate almaz.

Point2D sınıfının alt sınıflarında da *distance()* metodunun düzgün çalışması için bu metodu her bir alt sınıfta yeniden düzenlemek gerekir. Buna **metot ezme** (*method overriding*) denir. Metot ezme şu şekilde yapılır:

- Üst sınıftaki metot, tanımı değiştirilmeden alt sınıfa kopyalanır.
- Metodun içeriği alt sınıfta değiştirilir.
- Metodun üzerine *@Override* işareti konulur.

Şimdi Point3D sınıfımızı bu yönteme göre tekrar yazalım:

```
public class Point3D extends Point2D
{

    private int z;

    public Point3D()
    {
    }

    public int getZ()
    {
        return z;
    }

    public void setZ(int z)
    {
        this.z = z;
    }

    @Override
    public double distance()
    {
        return Math.sqrt(x * x + y * y + z * z);
    }

}
```

Gördüğünüz gibi, metodun tanımını değiştirmeden alt sınıfta tekrar yazdık ve yalnızca içeriğini düzenledik. Buna metot ezme denir, çünkü artık üst sınıftaki alt sınıftaki metot tarafından ezilmiştir. Artık *Point3D* türünden bir nesnenin *distance()* metodu çalıştırıldığında, üst sınıftaki metot görmezden gelinir. Yukarıdaki kodu tekrar çalıştırırsak artık doğru sonuç verecektir:

```
Point3D point = new Point3D();
point.setX(3);
point.setY(4);
point.setZ(6);

System.out.println(point.distance());
// Konsola 7.810249675906654 yazar
```

Metot ezme, metodun hem üst sınıflarda hem alt sınıflarda farklı şekilde çalışmasını sağlar. Artık 2 boyutlu noktaların uzaklığı hesaplanırken x ve y değerleri göz önüne alınırken, 3 boyutlu noktalarda buna bir de z değeri eklenir.

Metot ezme, Java'nın çalışma zamanında çokbiçimliliği desteklemesine imkân tanır. Bu sayede, üst sınıflarda bir metot tanımlayabilir ve her bir alt sınıfın bu metodu kendine göre özelleştirmesine izin verebilirsiniz. Böylece, bir yandan kalıtım kullanarak kod tekrarından kurtulduğunuz gibi, bir yandan gerektiğinde kodun modifiye edilmesine olanak tanıyabilirsiniz.

Ezilmiş mettota üst sınıfın metodunu çağırmak

Bir metodu ezdiğiniz zaman, üst sınıftaki metot tamamen görünmez olur. Fakat bazı durumlarda, ezilmiş metodun kodunu üst sınıfta ezdiği metoda göre yazmak isteyebilirsiniz. Bu durumda super deyimini kullanırız. Aşağıdaki örneği inceleyelim:

```
public class Point3D extends Point2D
{

    // Sınıfın diğer üyeleri

    @Override
    public double distance()
    {
        System.out.println("Bu metodu üst sınıfta çağırsaydık sonuç" +
            " şu olacaktı: " + super.distance());
        return Math.sqrt(x * x + y * y + z * z);
    }

}
```

```
Point3D point = new Point3D();
point.setX(3);
point.setY(4);
point.setZ(6);

System.out.println(point.distance());
```

Şimdi bu kodu çalıştırdığımız zaman çıktısı şu şekilde olur:

```
Bu metodu üst sınıfta çağırsaydık sonuç şu olacaktı: 5
7.810249675906654
```

SOYUT SINIFLAR (ABSTRACT CLASSES)

Bazen, hiyerarşik olarak çok çeşitli alt sınıfların olacağı mimarilerde, ortak bir üst yapı tanımlamak isteriz. Bu üst yapıda, bütün alt sınıfların miras alacağı ortak metotları belirleriz; fakat metodun çalışma prensibini alt sınıfların belirlemesini isteriz. Şu ana kadar bildiğimiz yöntemlerle bunu yapmak mümkün değildi; çünkü bir metot tanımladığınız zaman içeriğini de yazmak zorundasınız. Bu tarz üst yapıları belirlemek için **soyut sınıfları** (*abstract classes*) kullanırız. Bu sınıflara soyut deriz; çünkü tek başlarına bir anlam ifade etmezler, yalnızca alt sınıfları oluşturulduğunda anlam kazanırlar. Soyut sınıfları **abstract** deyimiyle oluştururuz:

Soyut sınıflar, kavramsal olarak da soyuttur, fiziksel olarak var olamazlar. Yani, soyut sınıfların bir örneği oluşturulamaz.

Soyut metotlar (Abstract methods)

Soyut sınıfları soyut metotlar tanımlayabilmek için kullanırız. Yalnızca tanımı belirlenmiş olan; fakat içeriği olmayan metotlara **soyut metot** (*abstract method*) denir. Soyut metotları **abstract** deyimiyle oluştururuz.

Soyut metot kullanarak, bir metot yapısı belirlemiş oluruz; fakat içeriğini alt sınıfların belirlemesini isteriz. Bir soyut sınıfın alt sınıfı oluşturulduğunda, soyut metotların her birini doldurması gerekir. Soyut metotlar yalnızca soyut sınıfların içerisinde kullanılabilir.

Aşağıdaki örneği inceleyelim:

```
public abstract class Animal
{

    public abstract void breathe();

}
```

Bu örnekte hayvanlar için soyut bir üst sınıf tanımladık. Her hayvan solunum yaptığı için bir solunum metodu ekledik; fakat bu metodu soyut olarak tanımladık, yani her bir alt sınıf bu metodu kendine göre özelleştirmek zorundadır.

Öncelikle hemen şunu belirtelim ki, soyut sınıfların bir örneği oluşturulamaz:

```
Animal animal = new Animal();    // Bu satır hataya sebep olur
```

Soyut sınıflar ve metotlar fiziksel olarak var olmadığı için, yukarıdaki kod hataya sebep olur.

Şimdi *Animal* sınıfının çeşitli alt sınıflarını oluşturalım:

```
class Human extends Animal
{

    @Override
    public void breathe()
    {
        System.out.println("Akciğerli solunum yapıyorum.");
    }

}

class Worm extends Animal
{

    @Override
    public void breathe()
    {
        System.out.println("Deri solunumu yapıyorum.");
    }

}

class Insect extends Animal
{

    @Override
    public void breathe()
    {
        System.out.println("Trake solunumu yapıyorum.");
    }

}

class Fish extends Animal
{

    @Override
    public void breathe()
    {
        System.out.println("Solungaç solunumu yapıyorum.");
    }

}
```

Gördüğünüz gibi, değişik hayvan alt sınıfları oluşturduk ve solunum metodunu kendilerine göre özelleştirdik. Solunum metodunun **abstract** olarak belirtilmediğine dikkat edin; çünkü üst sınıfta **abstract** olmasına rağmen, alt sınıflarda içeriğini doldurduğumuz için **abstract** olarak belirtilmez.

Ayrıca, oluşturduğumuz alt sınıflarında **abstract** olmadığına dikkat edin. Soyut sınıfların başka soyut alt sınıfları olabilir; fakat alt sınıfı soyut değilse, üst sınıftaki soyut metotların hepsini doldurmak zorundadır. Yukarıdaki sınıfların hiçbiri soyut olmadığı için *breathe()* metodu hepsinde olmak zorundadır.

Şimdi şu kodu inceleyelim:

```
Animal human = new Human();
Animal worm = new Worm();
Animal insect = new Insect();
Animal fish = new Fish();

human.breathe();
worm.breathe();
insect.breathe();
fish.breathe();
```

Bu kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
Akciğerli solunum yapıyorum.
Deri solunumu yapıyorum.
Trake solunumu yapıyorum.
Solungaç solunumu yapıyorum.
```

KALITIMI ÖNLEMEK

Nesne yönelimli programlama dillerinde kalıtım bize birçok fayda sağlar. Fakat bazen kalıtımı önlemek isteyebilirsiniz. Bu gibi durumlarda **final** deyimi kullanılır.

Metot ezmeyi önlemek

Alt sınıflarda ezilmesini istemediğiniz metotları **final** olarak belirtebilirsiniz:

```
public class Super
{

    public final void myMethod()
    {
        System.out.println("Bu ezilemez bir metottur.");
    }

}

class Sub extends Super
{

    @Override
    public void myMethod()
    {
        System.out.println("Ezilemeyen metodu ezmeye çalışıyorum..");
    }

}
```

Yukarıdaki kod henüz derleme aşamasındayken hata alır ve derlenmez; çünkü üst sınıftaki *myMethod()* metodu **final** olarak belirtilmesine rağmen, alt sınıfta ezmeye çalışılıyor.

Kalıtımı tamamen önlemek

Bazen yazdığınız bir sınıfın kalıtım hiyerarşisinde en altta olmasını, daha alt seviyede bir sınıf olmamasını istersiniz. Kısacası, o sınıfın bir alt sınıfının oluşturulmasını önlemek istersiniz. Bu durumda sınıfı **final** olarak belirtmeniz yeterlidir:

```
public final class Super
{
}

class Sub extends Super
{
}
```

Yukarıdaki kod henüz derleme aşamasında hata alır ve derlenmez; çünkü *Super* sınıfı final olarak belirtilmesine rağmen alt sınıfı oluşturmaya çalışılıyor.

Object sınıfı

Daha önce de belirttiğimiz gibi, Java'nın kalıtım hiyerarşisinde en üstte *Object* sınıfı bulunur. Bütün sınıflar doğrudan veya dolaylı olarak *Object* sınıfının alt sınıfıdır. Dolayısıyla bütün sınıflar *Object* sınıfının metotlarına sahiptir. Bu nedenle bu sınıfın metotlarını iyi bilmek gerekir.

Aşağıdaki tabloda *Object* sınıfının metotlarını görebilirsiniz:

Metot tanımı	**Ne için kullanılır?**
Object clone()	Nesnenin birebir aynısını oluşturur.
boolean equals(**Object** other)	Nesneyi parametre olarak aldığı diğer nesneyle karşılaştırır, aynı nesne olup olmadıklarını denetler.
void finalize()	Nesne artık kullanılmadığında hafızadan silineceği zaman çalıştırılır. (JDK 9 ile kullanımdan kaldırılmıştır)
final Class<?> getClass()	Nesnenin türünü belirtir.
int hashCode()	Nesnenin hash kodunu hesaplar.
final void notify()	Nesneyi bekleyen iş parçacıklarından birini devam ettirir.
final void notifyAll()	Nesneyi bekleyen iş parçacıklarının hepsini devam ettirir.
String toString()	Nesneyi tanımlayan bir String döndürür.
final void wait() **final void** wait(**long** milliseconds) **final void** wait(**long** milliseconds, **int** nanoseconds)	Bir iş parçacığının çalışmasını duraklatır ve bu nesneyi beklemesini sağlar.

equals()

İki nesnenin birbirine eşit olup olmadığını anlamak için bu metodu çağırırız:

```
String a = “deneme”;
String b = “deneme 2”;
String c = “deneme”;

System.out.println(a.eqauls(b));     // Konsola false yazar
System.out.println(a.eqauls(c));     // Konsola true yazar
```

equals() metodu iki nesneyi karşılaştırırken varsayılan olarak hafızada aynı nesne olup olmadıklarına bakar. Bu yöntemi alt sınıflarda metodu ezerek değiştirebiliriz. Örneğin, *Point2D* sınıfı için aşağıdaki örneği inceleyelim:

```
public class Point2D
{

    private int x;
    private int y;

    // Sınıfın diğer üyeleri

    @Override
    public boolean equals(Object other)
    {
        if (this == other)
            return true;

        if (other == null || getClass() != other.getClass())
            return false;

        Point2D otherPoint = (Point2D) other;
        return x == otherPoint.getX() && y == otherPoint.getY();
    }

}
```

Yukarıdaki örnekte *equals()* metodunu *Point2D* sınıfında ezdik ve farklı bir biçimde çalışmasını sağladık. Burada önce iki nesnenin aynı olup olmadığı kontrol ediliyor: eğer iki nesne aynıysa daha derin bir incelemeye gerek yoktur. Daha sonra diğer nesnenin türünü kontrol ediyoruz; eğer parametre olarak aldığımız nesne farklı bir türdeyse **false** döndürüyoruz. Eğer diğer nesne de *Point2D* türündeyse doğrudan tür dönüşümü yaparak bu nesneyi elde ediyoruz. Son olarak her iki nesnenin x ve y değerlerini karşılaştırıyoruz. Buna göre, iki *Point2D* nesnesi karşılaştırıldığında, yalnızca x ve y değerleri eşitse true döner.

hashCode()

Hash kodu, her nesneye özel olması gereken bir sayıyı ifade eder. Bu sayıyı *hashCode()* metoduyla elde ederiz. Java'nın varsayılan olarak bir hash kodu hesaplama yöntemi vardır; fakat metodu ezerek bu yöntemi değiştirebilirsiniz.

Eğer *equals()* metodunu ezdiyseniz bu metodu da ezmeli ve *equals()* metoduyla ilintili çalışır hale getirmelisiniz.

toString()

Java'da en çok kullanılan metotlardan biridir. Her nesnenin metin olarak bir karşılığı vardır. Bu metodu kullanarak bu karşılığı elde edebiliriz. Java varsayılan olarak nesnenin sınıfının ismini ve hash kodunu döndürür; fakat bu metodu ezerek bu yöntemi değiştirebiliriz.

Örneğin, *Point2D* sınıfının *toString()* metodunu çağırırsak şu şekilde bir karşılık döner:

```
Point2D point = new Point2D();
point.setX(3);
point.setY(4);

String str = point.toString();
System.out.println(str);     // Konsola Point2D@422 yazar
```

Gördüğünüz gibi konsolda sınıfın ismi, @ işareti ve ardından 422 yazmıştır. 422 nesnenin hash kodunu belirtir. Hemen fark edeceğiniz gibi, bu metin anlamsız bir metindir. Şimdi bu metodu ezelim ve anlamlı bir metin döndürmesini sağlayalım:

```
public class Point2D
{

    private int x;
    private int y;

    // Sınıfın diğer üyeleri

    @Override
    public String toString()
    {
        return "(" + x + ", " + y + ")";
    }

}
```

Şimdi yukarıdaki kodu tekrar çalıştırdığımızda çıktısı şu şekilde olur:

```
(3, 4)
```

ARAYÜZLER (INTERFACES)

Arayüzler soyut sınıflara benzer. İçeriklerini tanımlamadan bir sınıfın sahip olabileceği metotları belirler. Yine de soyut sınıflarla arayüzler arasında farklar vardır. İlk olarak, soyut sınıflar kalıtım hiyerarşisinde bir yer teşkil eder; arayüzler ise etmez. Bir sınıf yalnızca tek bir sınıftan kalıtım alabilir; diğer yandan birden fazla arayüz uygulayabilir. Bu bakımdan diyebiliriz ki, arayüzler Java'nın **çoklu kalıtım** (*multiple inheritance*) destekleme yöntemidir.

Arayüzler **interface** deyimiyle tanımlanır. Şimdi örnek bir arayüz tanımlayalım:

```
interface Nullable
{

      boolean isNull();

      Object getValue();

}
```

Yukarıda gördüğünüz gibi, arayüzlerde de soyut metotlar tanımlanır. Yine de bu metotlarda **abstract** deyimini kullanmadığımıza dikkat edin. Bu arayüzü uygulayan sınıfların belirtilen iki metoda da sahip olmaları gerekir. Şimdi bu arayüzü uygulayan bir sınıf yazalım. Arayüzler **implements** deyimiyle sınıfa uygulanır:

```
public class NullableObject implements Nullable
{

    private Object value;

    public void setValue(Object value)
    {
        this.value = value;
    }

    @Override
    public Object getValue()
    {
        return value;
    }

    @Override
    public boolean isNull()
    {
        return value == null;
    }

}
```

Bu örnekten de görebileceğiniz üzere, *Nullable* arayüzünü uyguladığımız için bu arayüzde belirtilen bütün metotlara sahip olmamız gerekir. Arayüzler de sınıflar gibi birer tür belirleyici olduğu için, aşağıdaki gibi bir atama hata vermez:

```
Nullable nullable = new NullableObject();
// NullableObject türünde bir nesneyi Nullable türünde bir değişkene
atayabiliriz
```

Bir arayüzü uygulayan sınıf **abstract** olarak belirtilmediği sürece, o arayüzde tanımlanmış bütün metotları doldurmak zorundadır.

Arayüzler birbirini genişletebilir. Bu, sınıflarda olduğu gibi **extends** deyimiyle sağlanır:

```
public interface X
{

    void myMethod1();

}

interface Y extends X
{

    void myMethod2();

}
```

Yukarıdaki örnekte *myMethod1()* ve *myMethod2()* adında iki farklı metot tanımlanmıştır. Bu metotlardan ilki *X* arayüzünde iken ikincisi *Y* arayüzündedir. Fakat *Y* arayüzü *X* arayüzünden miras aldığı için her iki metoda da sahip olur.

Varsayılan metotlar (default methods)

JDK 8'den önceki sürümlerde arayüzlerde yalnızca metot tanımı yapılabiliyordu. Arayüzlerde metot içeriği yazılmıyordu. JDK 8 ile arayüzlerde içeriği olan metot tanımlama özelliği gelmiştir. Bu tarz metotlara **varsayılan metot** (*default method*) denir. Varsayılan metotlar **default** deyimiyle belirtilir.

Varsayılan metotları anlamak için aşağıdaki örneği inceleyelim:

```
public interface Person
{

    int getBirthYear();

    int getAge();

}
```

Bu arayüzde iki metot tanımlanmıştır: *getBirthYear()* metodu kişinin doğduğu yılı elde etmek için kullanılırken, *getAge()* metodu kişinin yaşını hesaplamak için kullanılır. Burada şunu düşünelim: herkesin doğum yılı birbirinden farklıdır, dolayısıyla *getBirthDate()* metodu için ortak bir kod yazmak söz konusu değildir. Diğer yandan, yaş hesaplama metodu kişiden kişiye değişmez. Bulunduğumuz yıldan kişinin doğum yılını çıkarırsak kişinin yaşını öğrenmiş

oluruz. Bu kodu bu arayüzü uygulayan her sınıf için ayrı ayrı yazmaya gerek yoktur. Fakat JDK 8'den önce durum böyleydi. JDK 8 ile varsayılan metot kullanarak *getAge()* metodunu aşağıdaki gibi genelleştirebiliriz:

```
import java.util.Calendar;

public interface Person
{

    int getBirthYear();

    default int getAge()
    {
        return Calendar.getInstance().get(Calendar.YEAR) –
            getBirthYear();
    }

}
```

Gördüğünüz gibi, *getAge()* metodu için varsayılan bir kod yazdık. Bunu **default** deyimini kullanarak yaptık. Bu arayüzü uygulayan sınıflar artık *getAge()* metodunu doldurmak zorunda değildir; çünkü varsayılan bir uygulamasını biz zaten kodladık. Yine de bu sınıfların bu metodu kodlayamayacağı anlamına gelmez. Varsayılan metotlar bu arayüzü uygulayan sınıflar tarafından değiştirilebilir.

Kullanım alanları kısıtlı olduğu için anlatmayacağız; fakat arayüzlere JDK 8'den itibaren **statik** metot, JDK 9'dan itibaren **private** metotlar yazabilirsiniz.