Farklı Parametrik Yapılara Sahip Aynı İsimli Metotlar

Farklı sınıflarda aynı isimli metotların bulunması tamamen normaldir. Sınıflar farklı olduğundan herhangi bir karışıklık olmaz. Örneğin:

```
package csd;

class App {
	public static void main(String[] args)
	{
		Sample.foo();
		Mample.foo();
	}
}

class Sample {
	public static void foo()
	{
		System.out.println("Sample.foo");
	}
}

class Mample {
	public static void foo()
	{
		System.out.println("Mample.foo");
	}
}
```

Görüldüğü gibi Sample ve Mample sınıflarının foo metotları birbirlerinden farklıdır. Çağırma noktasında da sınıf ismi kullanıldığından karışıklık olmaz.

Aynı sınıf içerisinde de aynı isimli birden fazla metot bulunabilir. Buna nesne yönelimli programlama tekniğinde "method overloading" denilmektedir. Eğer sınıf içerisinde birden fazla aynı isimli metot bulunacaksa bunların parametrik yapılarının farklı olması gerekir. Parametrik yapıların farklı olması demek parametrelerin türce ya da sayıca farklı olması demektir. Parametre değişken isimlerinin farklı olmalarının bir önemi yoktur. Ayrıca metotların erişim belirleyicilerinin farklı olması, metodun static olup olmaması ya da metotların geri dönüş değerlerinin türlerinin farklı olması bu anlamda bir farklılığa yol açmaz. Örneğin:

```
class Sample {
	public static void foo(int a, double b)
	{
		//...
	}

	public void foo(int a, double b) //error:
	{
		//...
	}
	static void foo(int a, double b) //error
	{
		//...
	}

	public static int foo(int a, double b) //error
	{
		return 0;
	}
	public static void foo(int x, double y) //error
	{

	}
```

```
}
```
Burada tüm foo metotlarının parametrik yapıları aynı olduğundan aynı sınıf içerisinde bulunamazlar.

Örneğin:

```
package csd;

class Sample {
	public static void foo(int a, double b) //geçerli
	{
		//..
	}

	public static int foo(double b, int a) // geçerli
	{
		return 0;
	}

	public void foo(float a, char b)  //geçerli
	{
		//...
	}

	public static float foo() //geçerli
	{
		return 3.4F;
	}
}
```

Burada foo metotlarının parametrik yapıları farklı olduğundan tüm foo metotları geçerlidir.

Bir metodun ismi ve sırasıyla parametre türlerinden oluşan dizilime metodun imzası (signature) denilmektedir. Aynı sınıfta aynı imzaya sahip birden fazla metot bulunamaz. Örneğin:

```
package csd;

class Sample {
	public static void foo(int a, double b) //imza= foo, int, double
	{
		//..
	}

	public static int foo(double b, int a) //imza=foo, double, int
	{
		return 0;
	}
}
```

Burada foo metotlarının her ikisinin de imzaları farklıdır. Dolayısıyla bu metotların bildirimleri geçerlidir.

Aynı isimli bir metot çağrıldığında derleyici tarafından hangisinin çağrılacağına karar verme sürecine overload resolution denilmektedir. Overload resolution işleminin özet kuralı şöyledir: Derleyici çağrılma ifadesindeki argümanların türlerini inceler. Bununla tam uyuşan (exact match) bir metot varsa onun çağrıldığını kabul eder. Örneğin:

```
package csd;

class App {

	public static void main(String[] args)
	{
		char ch = 'A';
```

```
		float b = 3.24F;

		Sample.foo(b, ch);
	}
}

class Sample {
	public static void foo(int a, double b) //#1
	{
		System.out.println("foo(int, double)");
	}

	public static void foo(double b, int a)  //#2
	{
		System.out.println("foo(double, int)");
	}

	public static void foo(float a, char b)  //#3
	{
		System.out.println("foo(float, char)");
	}

	public static void foo()  //#4
	{
		System.out.println("foo()");
	}
}
```

Burada tam uyum olduğundan #3 metot çağrılacaktır.

Eğer çağrılma ifadesindeki argüman türleriyle tam uyumlu bir metot yoksa, bazı ayrıntılı kurallar devre girmektedir. Bu durumda overload resolution işlemi sırasıyla 3 aşamada gerçekleşir:

1. Aday metotlar (candidate methods) belirlenir.

2. Uygun Metotlar (applicable methods) seçilir.

3. En uygun metot (the most applicable method) belirlenir.

Son aşamada en uygun metot bulunamazsa error oluşur. Örneğin:

```
package csd;

class App {

	public static void main(String[] args)
	{
		int a = 20;
		short b = 10;

		Sample.foo(a, b); //#1 çağrılır
	}
}

class Sample {
	public static void foo(int a, int b) // #1
	{
		System.out.println("foo(int, int)");
	}

	public static void foo(double b, int a) // #2
	{
		System.out.println("foo(double, int)");
	}
```

```
	public static void foo(int a, long b) // #3
	{
		System.out.println("foo(int, long)");
	}

	public static void foo(short a, float b) // #4
	{
		System.out.println("foo(short, float)");
	}

	public static void foo() // #5
	{
		System.out.println("foo()");
	}

	public static void foo(int a) // #6
	{
		System.out.println("foo(int)");
	}

	public static void bar() // #7
	{
		System.out.println("bar()");
	}
}
```

Aday metotlar çağrılma ifadesindeki isimle aynı isme sahip olan metotlardır. Yukarıdaki örnekte 1, 2, 3, 4, 5 ve 6 numaralı metotlar aday metotlardır. Uygun metotlar çağrılma ifadesindeki argüman sayısı kadar parametreye sahip ve çağrılma ifadesindeki argümanlardan karşılık geldikleri parametrelere otomatik tür dönüşümünün tanımlı olduğu metotlardır. Yukarıdaki örnekte 1, 2 ve 3 numaralı metotlar uygun metotlardır. Nihayet uygun metotlar arasından en uygun metot seçilir. En uygun metot öyle bir metottur ki, argüman parametre dönüştürmeleri diğerleriyle yarışa sokulduğunda ya daha iyi bir dönüştürme sunar ya da daha kötü olmayan dönüştürme sunar.

Otomatik dönüştürmeler arasında kalite farkı vardır. T1 türünden T2 türüne ve T1 türünden T3 türüne otomatik dönüştürmeler mümkün olsun. Bu dönüştürmelerden biri diğerinden daha iyidir (daha kalitelidir). Otomatik dönüştürmede kaliteyi belirlemek için iki kural vardır (else-if gibi değerlendirilmelidir):

1. T1 -> T2 ve T1 -> T3 otomatik dönüştürmelerinde T2 ile T3 ün hangisi T1 ile aynı ise o dönüştürme daha kalitelidir. Örneğin:

int -> long
int -> int (bu dönüştürme daha kalitelidir)

2. T1 -> T2 ve T1 -> T3 otomatik dönüştürmelerinde T2'den T3'e otomatik dönüştürme var fakat T3'den T2'ye yoksa T1 -> T2 dönüştürmesi daha iyidir. Örneğin:

int -> long (bu dönüştürme daha iyidir)
int -> double
Örneğin:

int -> double
int -> float (bu dönüştürme daha iyidir)

NOT: En uygun metot öyle bir metottur ki, tüm argüman parametre dönüştürmesi diğer uygun metotlarla yarışa sokulduğunda ya onlardan daha iyi bir dönüştürme sunar ya da daha kötü olmayan bir dönüştürme sunar.

Yukarıdaki örnekte, 1 numaralı metot seçilir.

Örneğin:

```
package csd;

class App {

	public static void main(String[] args)
	{
		short a = 10;
		short b = 20;

		Sample.foo(a, b); // #2 çağrılır
	}
}

class Sample {
	public static void foo(double a, long b) // #1
	{
		System.out.println("foo(double, long)");
	}

	public static void foo(long b, int a) // #2
	{
		System.out.println("foo(long, int)");
	}

	public static void foo() // #3
	{
		System.out.println("foo()");
	}

	public static void foo(int a) // #4
	{
		System.out.println("foo(int)");
	}

	public static void bar() // #5
	{
		System.out.println("bar()");
	}
}
```

Burada en uygun metot 2 numaralı metottur.

Örneğin:

```
package csd;

class App {
	public static void main(String[] args)
	{
		short a = 10;
		short b = 20;

		Sample.foo(a, b); //error: İki anlamlılık (#1 ve #2 arasında seçim yapılamıyor)
	}
}
```

```
class Sample {
	public static void foo(int a, long b) // #1
	{
		System.out.println("foo(int, long)");
	}

	public static void foo(long b, int a) // #2
	{
		System.out.println("foo(long, int)");
	}

	public static void foo() // #3
	{
		System.out.println("foo()");
	}

	public static void foo(int a) // #4
	{
		System.out.println("foo(int)");
	}

	public static void bar() // #5
	{
		System.out.println("bar()");
	}
}
```

Burada çağrılabilecek iki tane metot bulunmaktadır. Dolayısıyla en uygun metot bulunamamaktadır. Bu durumda iki anlamlılık (ambiguity) oluşmaktadır. En uygun metot bulunamazsa error oluşur.

Java'da standart kütüphane içerisindeki sınıflarda pek çok metot overload edilmektedir.