**Factory Method Kalıbı(Factory Method Pattern)**

**Intent**: Define an interface for creating an object, but let subclasses decide which class to instantiate. **Factory Method** lets a class defer instantiation to subclasses.

• **Amaç**: Bir nesne yaratmak için bir arayüz tanımla, fakat hangi sınıfın nesnesinin oluşturulacağına alt sınıflar karar versin. **Factory Method** bir sınıfın nesne oluşturmasını alt sınıflarına ötelenmesine izin verir.

• Muhtemelen programcıların tabi olarak ihtiyacını ilk hissettikleri kalıptır.

**PROBLEM**

Nesne-merkezli programlardaki en temel iş, nesne yaratmaktır.

• Uygulamalarda farklı karmaşıklıkta pek çok sınıf vardır ve bu sınıfların nesneleri uygulamanın farklı yerlerinde oluşturulur.

• Bir nesneyi yaratmak, onu kullanmaktan daha karmaşık olabilir.

• Çünkü nesneleri yaratmak için pek çok bilgiye ihtiyaç duyulur.

• Bu yüzden tipik olarak kurucular (constructor), diğer metotlardan daha çok parametre alma eğilimindedirler.

**Bazı Noktalar.!**

**Factory Method**’da nesne farklı şekillerde oluşturulabilir:

• Sadece kurucu ya da varsayılan kurucu ve set metotları ile,

• Ya da nesne var olan bir **Prototype** nesneden kopyalanarak oluştulur,

• Veya nesne **Builder** ile inşa edilir.

• **Factory Method**’un her zaman nesne oluşturması gerekmez.

• Nesne havuzu (object pool) gibi yapılarda **Factory Method** havuzdan bir nesne de geri döndürebilir.

Aslolan **Factory Method**’un nesne yaratmayı soyutlamasıdır, sistemin dfiğer kısımlarınan ayırmasıdır, nesnenin nasıl yaratılacağı ayrı bir konudur ve bu nesneye bağlıdır.

**Factory Method** zaman zaman, var olan tek nesneyi, **Singleton**, döndüren metot olarak da kurgulanır.

• Bu durumda metot zorunlu olarak **static** olur.

• Nitekim, **Singleton**’ın **getInstance** metodu da bir **Factory Method** olarak düşünülebilir.

**SONUÇLAR**

* **Factory Method**, nesne yaratma işinin kontrolünü sisteme bırakarak, istemcinin sadece nesneleri istemekle yetinebileceğini göstermektedir.
  + Istemciler, kullandıkları sınıfların yaratılmalarına bağımlı değildir.
* Polymorphism sayesinde istemci yaratılan nesnelerinin gerçek tiplerini bilmeden onlarla haberleşebilmektedir.
* Bu yaklaşımın en temel problemi, her yeni sınıf için, örneğin **Engineer**, sınıfı için **EngineerFactory** gibi yeni bir **Factory** alt sınıfına hatta ayrı arayüzler olması durumunda yeni bir arayüze de ihtiyaç duyurmasıdır

**KULLANIMLAR**

Java APIlerinde **Factory Method**’un pek çok kullanımı vardır.

• **java.sql** paketinde **Connection** üzerinde **createStatement()** metotları farklı **Statement** nesneleri üretilir.

• Aynı paketteki **Statement** üzerindeki **executeQuery()** metodu ile de **ResultSet** nesnesi üretilir.

**java.sql** paketindeki aşağıdaki yapılardan **DriverManager** dışındakiler arayüzdürler.

• Peki nesneler nasıl oluşmaktadırlar?

Ya da sınıflar nerededirler?



Benzer şekilde:

• **javax.persistence** paketinde **EntityManagerFactory** üzerindeki **createEntityManager()** metotları **EntityManager** nesneleri üretir.

• Native Hibernate’deki **SessionFactory** sınıfı **Session** nesneleri üretir.

Bazen factory metodunun **static** olduğu da görülür:

• **java.sql.Driver getConnection()**

• **javax.persistence.Persistence createEntityManagerFactory()**

• **java.util.Calendar getInstance()**

• **java.util.ResourceBundle getBundle()**

• **java.text.NumberFormat getInstance()**

• Buradaki amaç nesne üreten metodunun override, içindeki sınıfın da genişletilmesini (**extend**) önlemektir.

**Factory Method**, **Abstract Factory** kalıbında sıklıkla kullanılır. • **Factory Method** nesneleri **Singleton** da yapılabilir.