

# NESNE YÖNELİMLİ PROGRAMLAMA 2(Object Oriented Programming 2/OOP)

**Öğr. Gör. Celil ÖZTÜRK**

Marmara Üniversitesi

Teknik Bilimler Meslek Yüksekokulu

# İçerik

✓ SOLID

✓ SINGLETON TASARIM KALIBI

✓ FACTORY TASARIM KALIBI

# SOLID

- **S** — Single-responsibility principle
- **O** — Open-closed principle
- **L** — Liskov substitution principle
- **I** — Interface segregation principle
- **D** — Dependency Inversion Principle

# Tasarım Kalıpları

“Tasarım kalıpları, uzmanların yeni sorunları çözmek için geçmişte çalıştıkları çözümlerin uygulamalarının iyi belgelenmiş halidir.”

Tasarım kalıplarının arkasındaki düşünce, yazılım geliştirilirken sıklıkla karşılaşılan problemler için sunulan genel çözümler için **standartlaşmış** bir yol geliştirmektir.

# Tasarım Kalıplarının Avantajları

- Kalıpların standartlaştırılması, tüm geliştiricilerin (profesyoneller, yeni başlayanlar veya uzmanların) kararlarını daha kolay vermesini sağlamaktadır.
- Tasarım kalıpları ortak bir kelime haznesi sağlar. Bu geliştiriciler arasındaki iletişimi daha da kolay hale getirir. Bir tasarımı detaylıca açıklamaktansa, planlarımızı açıklamak için kalıp adını kullanabiliriz.
- Kalıplar birbirleri ile ilişkilendirilebilir, böylece geliştiriciler projelerinde hangi kalıpların birlikte bulunması gerektiğini kolayca anlayabilir.

# Tasarım Kalıplarının Avantajları

- Tasarım Kalıpları nesneye yönelik programlama topluluğu aracılığıyla tecrübe paylaşımı için etkili bir yöntem sunmaktadır. Örneğin; C++, Smalltalk, C# ya da Java programlama dillerinde kazanılan bilgiler, Web projelerinde ortaya çıkan uzmanlık gibi öğrenilen bilgiler biriktirebilir ve bunlar diğer geliştiricilerle paylaşılabilir.

# Yaratımsal Kalıplar(**Creational Patterns**)

- Yaratımsal kalıplar, yazılım nesnelerinin nasıl yaratılacağı ile ilgilenen tasarım kalıplarıdır.
- Daha önceden belirlenen durumlara bağlı olarak, gerekli nesneleri yaratır.
- Uygulamada nesnelerin oluşturulmasından sorumlu yapılardır.
- Bu kalıplar nesneye yönelik programların en yaygın görevlerinden biri olan yazılım sistemindeki nesnelerin yaratılması hakkında yol göstermektedir.

# Tasarım Kalıpları

## **Creational Patterns(Yaratımsal Kalıplar)**

- Singleton Pattern
- Factory Pattern
- Abstract Factory Pattern
- Builder Pattern
- Prototype Pattern



# Singleton Tasarım Kalıbı

- Singleton(Tek Nesne...) bir sınıfın tek bir örneğini oluşturmak için kullanılır.
- Singleton tasarım kalıbında nesnenin uygulama kapanana kadar bir kez üretilmesini ve tek bir instance'ın olmasını kontrol altında tutar.
- Yaratılan nesne, **sınıf dışından da erişilebilir durumdadır(global)**.
- Yaratılan nesne **her yerden erişilebilir** olmalı fakat **sadece bir kez yaratılmalıdır**.
- Bu sınıfın bir anda sadece bir örneğinin olması istenildiği zamanlarda kullanılır.
- Singleton'a erişimde new ile nesne oluşturulamaz, sınıf ve instance metot kullanılır..

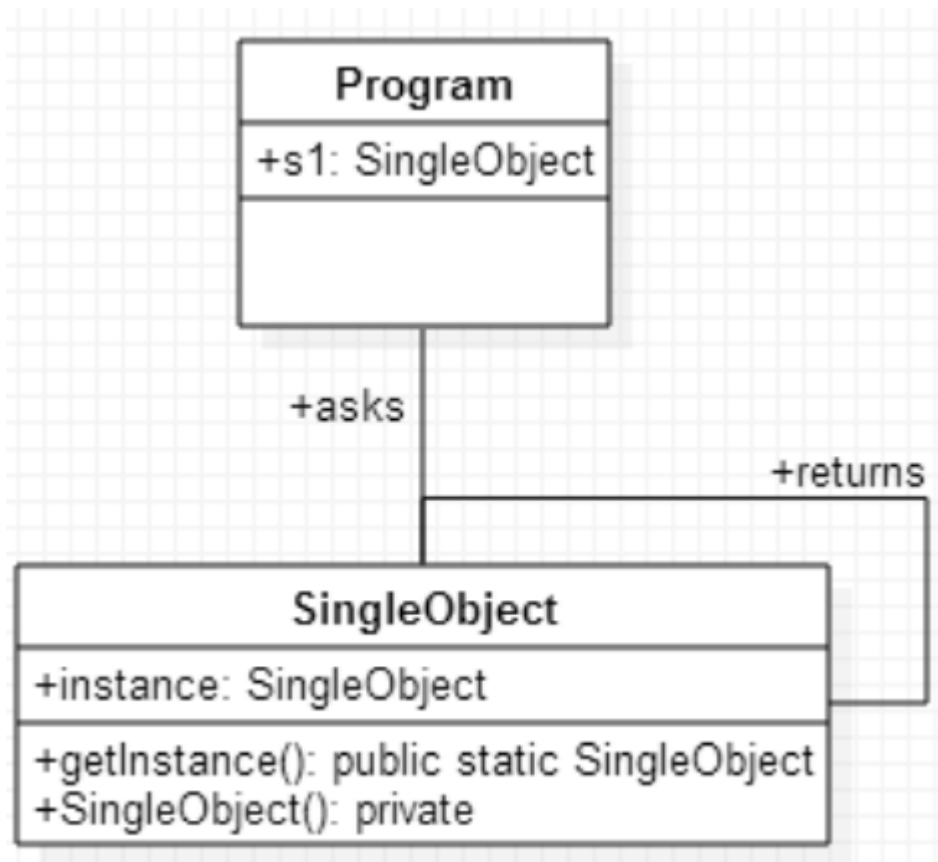
# Singleton Tasarım Kalıbı

- Ana uygulamada **global bir nesne yaratılmalı** ve sonrasında bu nesnenin referansının ihtiyaç olduğunda **geçirilmesinin sağlanmasıdır**.
- Diğer bir yöntem ise static değişken kullanmaktır, uygulama bir sınıfın içerisinde birkaç static nesneye sahip olmakta ve onlara direkt olarak ulaşmaktadır.
- Singleton yapısı sayesinde static bir örnek(instance) oluşturulur ve sonraki isteklerde aynı örnek gönderilir.

# Singleton Tasarım Kalıbı

- Singleton deseni uygulanacak olan sınıfın constructor metodu **private** olarak tanımlanır.(Nesne yaratmayı kontrol etmek amacıyla...)
- Yukarıdaki aşama bir soruna neden olur: Bir instance yaratılması imkansız olur, bundan dolayı erişimci metot bir **static** metot tarafından sağlanmaktadır.(getInstance())
- Yukarıdaki metot önceden yaratılmadıysa, **yeni tek bir instance yaratır** ve singleton'un referansını, bu nesneyi çağıran metot döndürür.
- Singleton'un referansı – gelecek istekler için - sınıfında tanımlanan static bir property'de saklanır.

# Singleton Tasarım Kalıbı



# Singleton Tasarım Kalıbı

## SingleObject.java

```
public class SingleObject {
    private static SingleObject instance;

    //Kurucu metodu private olarak tanımlanır.
    private SingleObject()
    {}

    public static SingleObject getInstance()
    {
        //Lazy loading
        if (instance == null) {
            instance = new SingleObject();
            System.out.println("Yeni instance yaratıldı!");
        }
        else
            System.out.println("Önceden yaratılmış olan instance geri döndürüldü!");

        return instance;
    }
}
```

## Main.java

```
public class Program {
    public static void main(String[] args) {

        SingleObject s1 = SingleObject.getInstance();
        SingleObject s2 = SingleObject.getInstance();
        SingleObject s3 = SingleObject.getInstance();

    }
}
```

# Singleton Tasarım Kalıbı

## SingleSinif.java

```
public class SingletonSinif {  
    private static SingletonSinif _instance;  
  
    private SingletonSinif() {  
  
    }  
    public static SingletonSinif getInstance()  
    {  
        if(_instance==null)  
        {  
            _instance = new SingletonSinif();  
        }  
  
        return _instance;  
    }  
}
```

## Main.java

```
public class Singleton {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
  
        SingletonSinif singleton1= SingletonSinif.getInstance();  
        System.out.println(singleton1);  
  
        SingletonSinif singleton2= SingletonSinif.getInstance();  
        System.out.println(singleton2);  
    }  
}
```

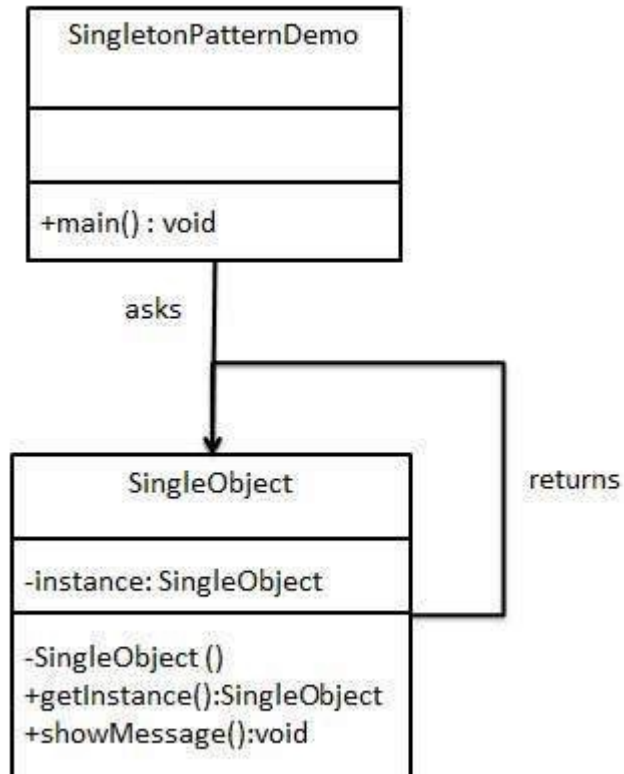
Ekran çıktısı



Output - Singleton (run) ×

```
run:  
singleton.SingletonSinif@1db9742  
singleton.SingletonSinif@1db9742  
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Singleton Tasarım Kalıbı



```
public class Singleton {

    //create an object of Singleton
    private static Singleton instance = new Singleton();

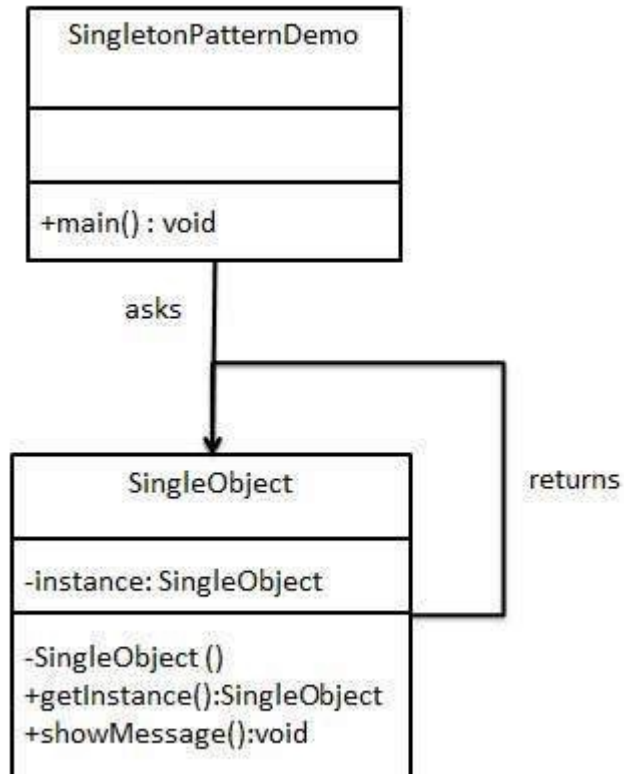
    //make the constructor private so that this class cannot be
    //instantiated
    private Singleton(){}

    //Get the only object available
    public static Singleton getInstance(){
        return instance;
    }

    public void showMessage(){
        System.out.println("Hello World!");
    }

}
```

# Singleton Tasarım Kalıbı



```
public class App {
    public static void main(String[] args) {

        //illegal construct
        //Compile Time Error: The constructor Singleton() is not visible
        //Singleton object = new Singleton();

        //Get the only object available
        Singleton object = Singleton.getInstance();

        //show the message
        object.showMessage();

    }
}
```



# Singleton Artıları ve Eksileri

- Singleton kendi kendinin instance'ını yaratabilen tek sınıftır.
- Sağlanan static metodu kullanmadan yeni bir singleton yaratılamaz böylece yaratılan nesne programın çalışma süresince o sınıfın tek nesnesi olur.
- Singleton'a ihtiyaç duyan nesnelerin tümüne aynı singleton'un referansını geçirilmesi gerekmez; çünkü o sınıfa her erişildiğinde aynı singleton nesnesinin referansını geri döndürür.
- Fakat, Singleton tasarım kalıbı implementasyona bağlı olarak, threading sorunları çıkarabilir. Bir multi threading uygulamada singleton'un başlatılma şekline dikkat edilmelidir.

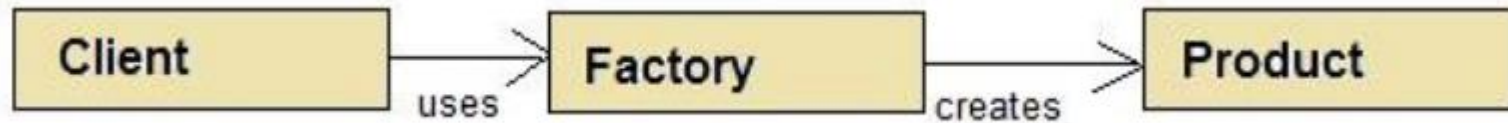
# Factory(Fabrika) Tasarım Kalıbı(Factory Method)

- Yapısal olarak birbirine benzeyen sınıflara aynı arayüzü uygular. Sonrasında bu sınıfların yaratılma sorumluluğunu bir metoda veya sınıfa verir.
- Temel amaç, oluşturmak istediğimiz sınıfın kendisinden bir örnek(instance) istemek yerine(yeni bir new işlemi), ortak bir instance üzerinden istenen nesnenin üretilmesini sağlamaktır.

# Factory Tasarım Kalıbı(Fabrika)

- Bir fabrikada benzer ürünlerin üretilmesini düşünün. Bizim bazı ürünlerin üretiminde bilgi sahibi olmamamız ancak onları kullanmamız gibi bir durum söz konusudur.
- Nesnelerin nasıl yaratılacağını kalıtım yoluyla alt sınıflara bırakıp, nesne yaratımı için tek ara yüz kullanarak ara yüzle nesne yaratım işlevlerini temelde birbirinden ayırmaya yarayan **yaratımsal tasarım kalıbıdır**.
- Bazı nesneler doğrudan kullanıcısı tarafından yaratılabilmektedir. Bunun anlamı, sınıfın başlangıç fonksiyonunu (new operatörü) kullanarak nesnenin yaratılmasıdır.
- Bazı nesnelerin (Product) kullanıcı(Client) olarak erişebileceğimiz başlangıç fonksiyonları bulunmamaktadır.

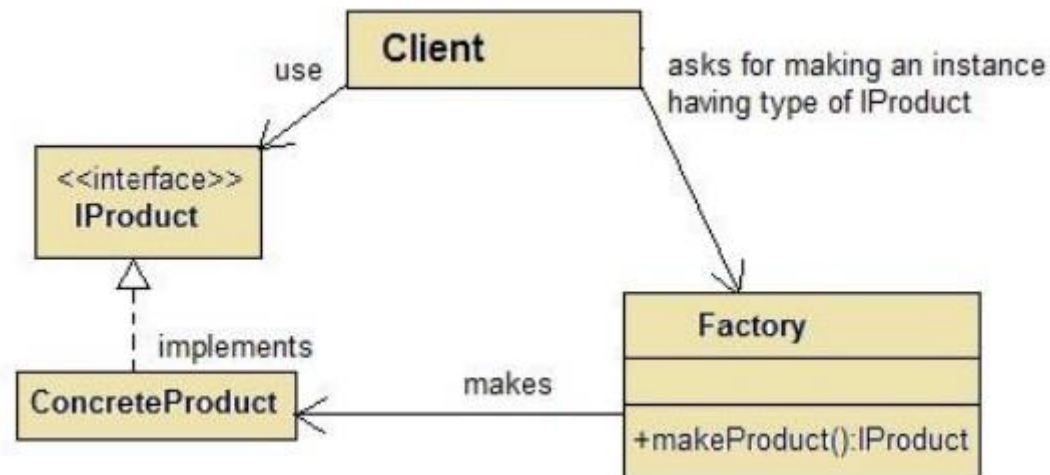
# Factory Tasarım Kalıbı(Fabrika)



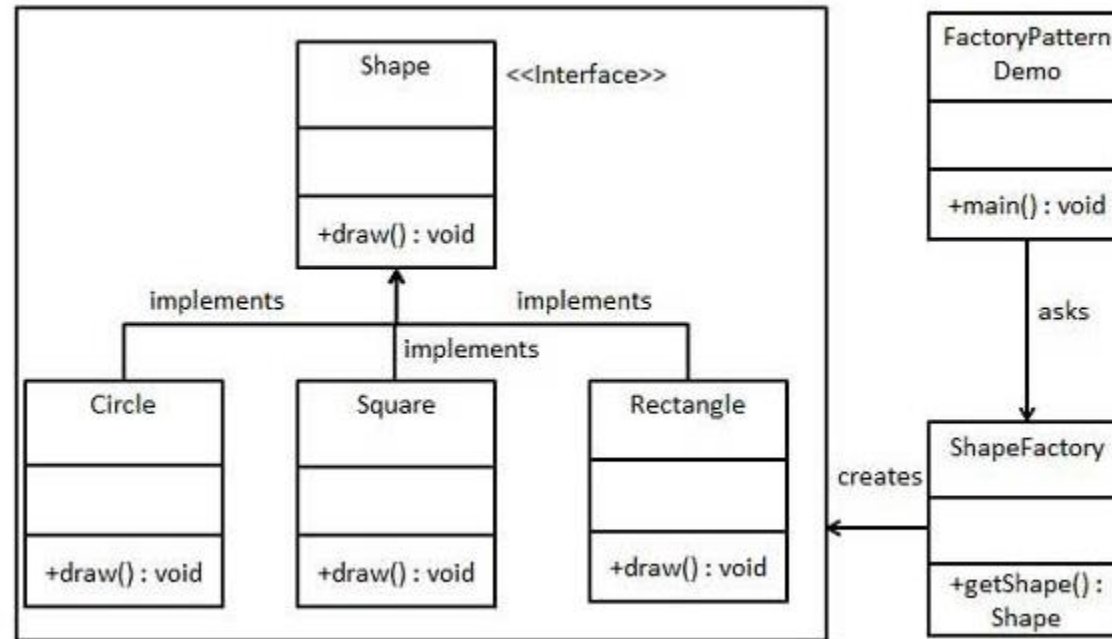
- Client olan nesne Factory nesnesini kullanarak ihtiyacı olan Product nesnesini elde eder.
- İstenen tipte yeni nesne oluşturma sürecinin **Factory** sınıfına aktarılması ile birlikte **nesne üretme ve initialize etme süreci client'tan soyutlanmış olur.**
- Bu sayede client; uygulama içerisinde tamamen kendi rolüne odaklanmış olur, çünkü yeni nesnenin nasıl oluşturulacağına dair detaylardan soyutlanmış olur, bunları bilmek zorunda değildir.

# Factory Tasarım Kalıbı(Fabrika)

- İstenen tipte nesne oluşturma sürecini Client'ın bu konuda detay bilgi sahibi olmadan gerçekleştirilmesini sağlar.
- Yeni oluşturulan nesneye bir interface ile referans edilerek ulaşılmasını sağlar.



# Factory Tasarım Kalıbı(Fabrika)



# Factory Tasarım Kalıbı(Fabrika)

```
public interface Shape {  
    void draw();  
}
```

```
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Square::draw() method.");  
    }  
}
```

```
public class Rectangle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Rectangle::draw() method.");  
    }  
}
```

```
public class Circle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Circle::draw() method.");  
    }  
}
```

# Factory Tasarım Kalıbı(Fabrika)

```
public class ShapeFactory {  
  
    public Shape getShape(String shapeType){  
        if(shapeType == null){  
            return null;  
        }  
        if(shapeType.equalsIgnoreCase("CIRCLE")){  
            return new Circle();  
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
            return new Rectangle();  
        } else if(shapeType.equalsIgnoreCase("SQUARE")){  
            return new Square();  
        }  
        return null;  
    }  
}
```


Sınıfların nesnesini oluşturmak için kullanılan  
**\*\*Factory Sınıfı\*\***



# Factory Tasarım Kalıbı(Fabrika)

```
public class FactoryPatternDemo {  
    public static void main(String[] args) {  
        ShapeFactory shapeFactory = new ShapeFactory();  
  
        Shape shape1 = shapeFactory.getShape("CIRCLE");  
        shape1.draw();  
  
        Shape shape2 = shapeFactory.getShape("RECTANGLE");  
        shape2.draw();  
  
        Shape shape3 = shapeFactory.getShape("SQUARE");  
        shape3.draw();  
    }  
}
```

**\*\*Client tarafından nesnenin oluşturulması için,  
Factory sınıfının kullanımı**



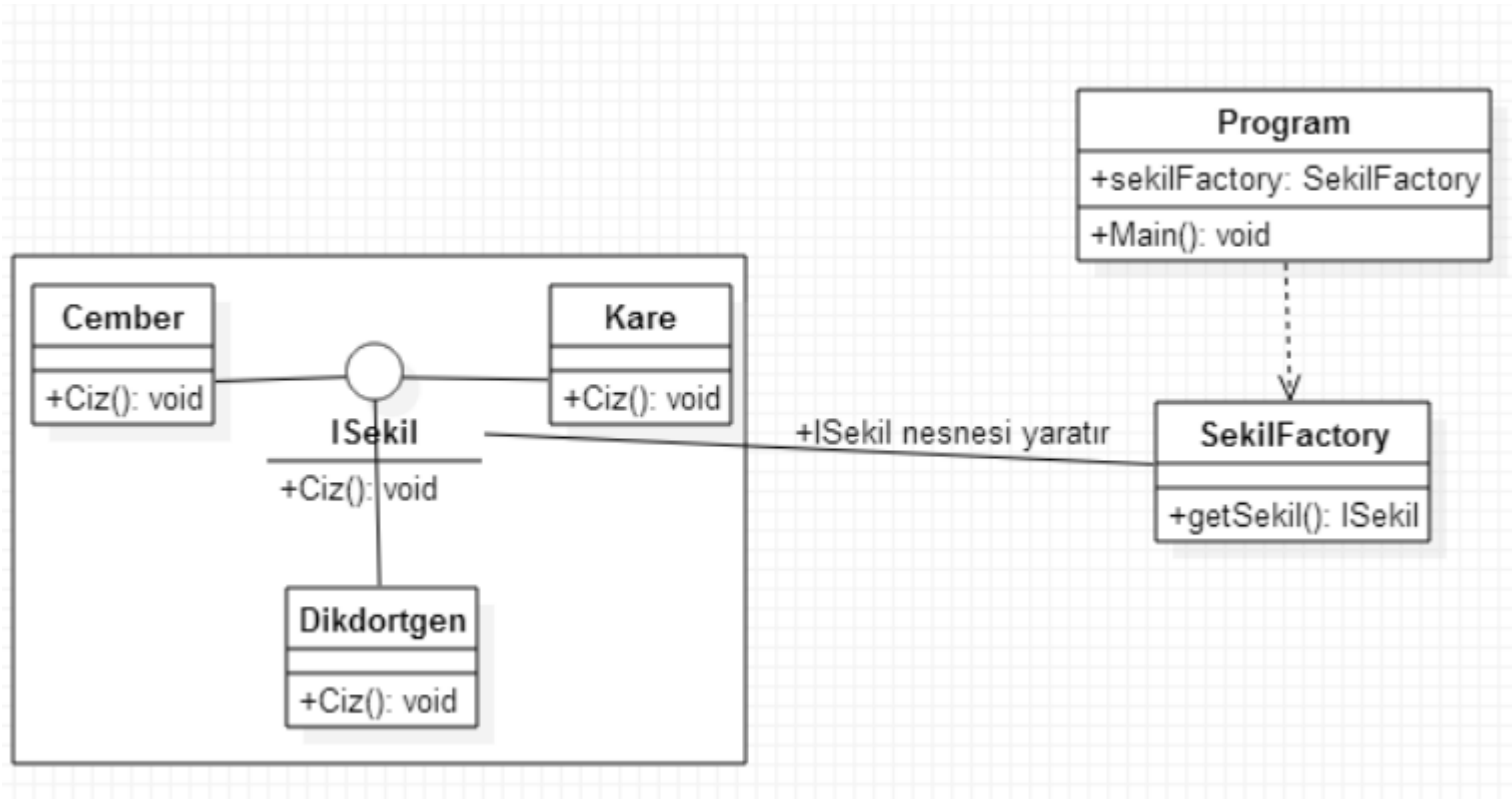
```
Inside Circle::draw() method.  
Inside Rectangle::draw() method.  
Inside Square::draw() method.
```

- Tek arayüze bağlı 3 adet sınıf oluşturuldu.
- Bu sınıflardan nesne oluşturma işlemi bir factory sınıfına görev olarak verildi.
- Bu factory sınıfı da bir istemci tarafından çalıştırıldı.

# Factory Tasarım Kalıbı(Fabrika)

- ISekil arayüzünü ve bu arayüzü implemente eden somut sınıfları (Kare, Cember, Dikdortgen) yaratınız.
- Sonrasında factory sınıfı olan SekilFactory sınıfını yaratınız.
- Program sınıfında SekilFactory sınıfından bir Sekil nesnesi elde edebilecek şekilde SekilFactory sınıfını tanımlayınız.
- Program sınıfında gereken nesnenin tipini (Cember, Kare, Dikdortgen) bilgi olarak geçebilecektir.

# Factory Tasarım Kalıbı(Fabrika)



# Factory Tasarım Kalıbı(Fabrika)

\*\*\*ISekil arayüzünü **implemente** eden somut sınıfları yaratırız.

ISekil.java

```
public interface ISekil
{
    void Ciz();
}
```

Kare.java

```
public class Kare implements ISekil
{
    @Override
    public void Ciz()
    {
        System.out.println("Kare::Ciz() metodu çalıştı!");
    }
}
```

# Factory Tasarım Kalıbı(Fabrika)

**ISekil** arayüzünü **implemente** eden somut sınıfları yaratırız.

Cember.java

```
public class Cember implements ISekil
{
    @Override
    public void Ciz()
    {
        System.out.println("Cember::Ciz() metodu çalıştı!");
    }
}
```

Dikdortgen.java

```
public class Dikdortgen implements ISekil
{
    @Override
    public void Ciz()
    {
        System.out.println("Dikdortgen::Ciz() metodu çalıştı!");
    }
}
```

# Factory Tasarım Kalıbı(Fabrika)

- Geçirilen bilgiye göre ISekil arayüzünü implemente eden sınıfların nesnelerini yaratan **SekilFactory** sınıfını ve **getSekil(ESekilTur sekilTur)** metotunu oluşturunuz.

ESekilTur.java

```
public enum ESekilTur
{
    cember,
    dikdortgen,
    kare;
}
```

SekilFactory.java

```
public class SekilFactory
{
    public ISekil getSekil(ESekilTur sekilTipi)
    {
        switch (sekilTipi) {
            case cember:
                return new Cember();
            case kare:
                return new Kare();
            case dikdortgen:
                return new Dikdortgen();
            default:
                return null;
        }
    }
}
```

# Factory Tasarım Kalıbı(Fabrika)

- Şeklin tipi gibi bir bilgiyi geçirerek somut sınıflardan nesne elde etmek için **SekilFactory** sınıfını kullanırız.

```
public class Program {  
  
    public static void main(String[] args) {  
        SekilFactory sekilFactory = new SekilFactory();  
  
        ISekil sekil1 = sekilFactory.getSekil(ESekilTur.cember);  
        sekil1.Ciz();  
  
        ISekil sekil2 = sekilFactory.getSekil(ESekilTur.kare);  
        sekil2.Ciz();  
  
        ISekil sekil3 = sekilFactory.getSekil(ESekilTur.dikdortgen);  
        sekil3.Ciz();  
    }  
}
```

# Factory Tasarım Kalıbı(Fabrika) / Örnek 2

```
public interface Computer
{
    void name(); void
    since(int year);
}
```

```
public class Asus implements Computer {

    @Override
    public void name() {
        System.out.println("Bilgisayarın Markası Asus");
    }

    @Override
    public void since(int year) {
        System.out.println(year + " senesinde alınmış.");
    }
}
```



# Factory Tasarım Kalıbı(Fabrika) / Örnek 2

```
public class Mac implements Computer
{

    @Override
    public void name()
    {
        System.out.println("Bilgisayarın Markası Mac");
    }

    @Override
    public void since(int year)
    {
        System.out.println(year + " senesinde alınmış.");
    }

}
```

# Factory Tasarım Kalıbı(Fabrika) / Örnek 2

```
public class ComputerFactory
{
    public static Computer createComputer(Class aClass)
    throws IllegalAccessException, InstantiationException
    {
        return (Computer) aClass.newInstance();
    }
}
```

# Factory Tasarım Kalıbı(Fabrika) / Örnek 2

```
public class Main {  
  
    public static void main(String[] args) {  
  
        try {  
            Asus asus = (Asus) ComputerFactory.createComputer(Asus.class);  
            asus.since(1234);  
            asus.name();  
  
            Mac mac = (Mac) ComputerFactory.createComputer(Mac.class);  
            mac.name();  
  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
  
    }  
}
```

# Factory Tasarım Kalıbı(Fabrika)

**\*\***Factory tasarım kalıbında, istemciye normal oluşturma mantığına maruz bırakmadan nesne oluşturulmasına olanak sağlanır ve yeni yaratılan nesneye ortak bir arayüz kullanarak erişilebilir.

# Kaynaklar

- Java ve Java Teknolojileri, *Tevfik KIZILÖREN* – Kodlab Yayınları
- Yazılım Mühendisliği CBU-Dr. Öğr. Üyesi Deniz Kılınç Yazılım Mimarisi ve Tasarımı Ders Notları
- [Yazılım Kalitesi ve Kötü Tasarım Belirtileri | by Ramazan Ümit Bülbül | Medium](#)
- <https://medium.com/gokhanyavas/creational-patterns-yarat%C4%B1msal-desenler-d4ccd26da0a>
- <http://cagataykiziltan.net/tr/tasarim-kaliplari-design-patterns/1-creational-tasarim-kaliplari/singleton-pattern/>
- <https://github.com/gokhanyavas/Design-Pattern/blob/master/Singleton/src/App.java>
- [https://bidb.itu.edu.tr/sevir-defteri/blog/2013/09/08/fabrika-tasar%C4%B1m-kal%C4%B1b%C4%B1-\(factory-design-pattern\)](https://bidb.itu.edu.tr/sevir-defteri/blog/2013/09/08/fabrika-tasar%C4%B1m-kal%C4%B1b%C4%B1-(factory-design-pattern))
- <https://medium.com/bili%C5%9Fim-hareketi/factory-fabrika-pattern-c14baca707be>
- <https://yasinmemic.medium.com/factory-design-pattern-4c12afa1c760>