Semih Kırdinli

Java Developer at Solvia

Linkedin : https://www.linkedin.com/in/semihkirdinli/
Github : https://github.com/semih

- Nested Classes
  - Static Nested Class
  - Member Inner Class (non-static)
  - Local Inner Class
  - Anonymous Inner Class
- Java 8 Lambda Expressions
  - Değişkene Atama Yöntemi
  - Fonksiyona Parametre Geçme Yöntemi
  - @FunctionalInterface Annotation
  - Predicate<T> Functional Interface
  - Aggregate Operations

Nested Classes

Java programlama dilinde bir sınıf başka bir sınıfın içinde tanımlanabilir.
İç içe sınıf türleri:

- Static nested class – dış sınıfın statik kapsam boğuyla ilişkilidir.
- Member inner class – dış sınıfın nesne örneğiyle ilişkilidir. (non-static)
- Local inner class – belirli bir metodun kapsam bloğuyla ilişkilidir.
- Anonymous inner class – bir interface ya da sınıfın içindeki anonim sınıflardır.

Static Nested Class

Static nested class'tan bir nesne örneği oluşturulmak istendiğinde, onu sarmalayan sınıfın adının da yazılması gerekir. **static** olarak tanımlandığı için sadece onu sarmalayan sınıfa ait nesne değişkenlerine erişebilir.

```
public class Outer {
    public static class StaticNested { }
}

   Outer.StaticNested inner = new Outer.StaticNested();
```

Member Inner Class (non-static)

Bu sınıflara **iç sınıflar** da denir. Member inner class'tan bir nesne örneği oluşturulmak istendiğinde, onu sarmalayan sınıfın nesnesinin adı yazılmalıdır. Bu yüzden önce sarmalayan sınıfın bir nesnesini oluşturmak gerekir. **static** olmayan iç sınıflar, onu sarmalayan sınıfın alanlarına **private** olsalar bile erişebilir.

```
public class Order {
    private Set<Item> items = new HashSet<>();
    public void addItem(Product product, int quantity) {
        items.add(new Item(product, quantity));
    }
    class Item {
        private Product product;
        private int quantity;
        private Item(Product product, int quantity) {
            this.product = product;
            this.quantity = quantity;
        }
    }
}
  Order order1 = new Order();
  order1.addItem(new Drink("Tea"),  quantity: 2);
  order1.addItem(new Food( drink: "Sandwich"),  quantity: 1);
```

Local Inner Class

Yerel sınıfların nesne örneği sadece onu sarmalayan sınıfın metodunun kapsam bloğunda oluşturulabilir. Bu yüzden sadece tanımlanmış oldukları blok veya metot içinden erişilebilir. Yerel sınıflar ise aynı iç sınıflar (member inner class) gibi onu sarmalayan sınıfın alan ve metotlarına erişebilir. Ek olarak, içinde tanımlandığı metodun parametreleri arasından sadece **final** olanlara erişebilir.

```
public class Order {
    private Map<Integer, Item> items = new HashMap<>();

    public void manageTax(final String saleLocation) {
        class OrderTaxManager {
            private BigDecimal findRate(Product product) { return new BigDecimal( val: 0.5); }
            BigDecimal calculateTax() { return new BigDecimal( val: 0.18); }
        }

        OrderTaxManager taxManager = new OrderTaxManager();
        BigDecimal taxTotal = taxManager.calculateTax();
    }
}
```

Anonymous Inner Class

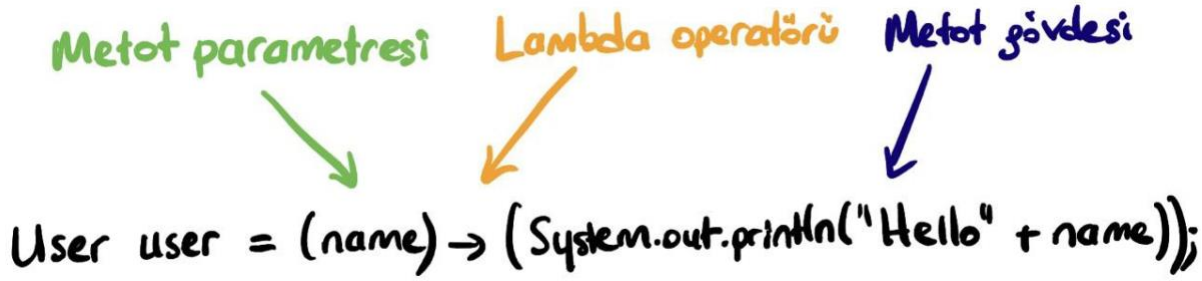
Anonim sınıflar bir sınıf ismine sahip olmayan sınıflardır. Bir başka sınıftan kalıtım alınarak tanımlandıkları gibi interface'in gerçeklemesi(implementation) olarak da tanımlanabilirler. Nesne örneği oluşturulduğu anda tanımlanırlar.

```
public class Order {
    public BigDecimal getDiscount() {
        return BigDecimal.ZERO;
    }
}
```

```
Order order = new Order() {
    @Override
    public BigDecimal getDiscount(){
        return BigDecimal.valueOf(0.1);
    }
};
```

Java 8 Lambda Expressions

- Lambda ifadeleri kullanmadaki amaç koddaki satır sayısını azaltmak, daha sade ve anlaşılır kod yazmaktır.
- Lambda ifadeler bir functional interface'in inline implementasyonudur. (Functional interface'in uygulamak için sadece bir tane abstract metodu bulunur.)
- Lambda ifadelerin sonucu bir değişkene atanabilir veya bir fonksiyona parametre olarak geçilebilir.



Lambda Expression Yazım Şekli

Değişkene Atama Yöntemi

```
public interface User {
    void name(String name);
}
```

Java 8 öncesi kullanım:

```
public class Driver implements User {
    @Override
    public void name(String name) {
        System.out.println("Hello " + name);
    }
}
```

```
User user = new Driver();
user.name("Harold");
```

Lambda ile kullanım:

```
User user = (name) -> System.out.println("Hello " + name);
user.name("John");
```

Fonksiyona Parametre Geçme Yöntemi

```
public interface Anonymous {
    public int doOperation(int a, int b);
}
```

```
public static void main(String[] args) {
    calculate( a: 5, b: 8, (a,b)-> ( a + b ));
}

public static int calculate(int a, int b, Anonymous anonymous) {
    return anonymous.doOperation(a, b) * 2;
};
```

Lambda İfadelerin Kullanımı ile İlgili Örnekler

```
Collections.sort(products, (p1,p2) -> p1.getPrice().compareTo(p2.getPrice()));
List<String> list = new ArrayList<>();
Comparator<String> sortText = (s1, s2) -> s1.compareTo(s2);
list.removeIf((s)-> s.equals("remove me"));
list.sort((s1,s2) -> { return s1.compareTo(s2); });
Collections.sort(list, sortText);
```

@FunctionalInterface Annotation

Lambda ifadelerin aslında birer functional interface implementasyonu olduğunu belirtmiştik. Bu da o interface'e ait sadece bir tane abstract sınıfın olmasını gerektiriyordu. İkinci bir abstract metot yazıldığında artık functional interface'ten bahsedilemez ve artık lambda ifadelerinde kullanılamaz. Bu metodun bir tane abstract metodu olduğunu garanti etmek için @FunctionalInterface annotation'ı kullanılabilir.

```
@FunctionalInterface
public interface Foo { String method(); }
```

Predicate<T> Functional Interface

Predicate<T> interface'i generic interface'lerin bir örneğidir. Generic tipler (örneğin generic interface'ler) köşeli parantezler arasında (<>) bir veya daha fazla tip parametresi(T) alır. Bu şekilde tanım yaptığınızda parametreli bir tipiniz olur.

Bu interface, T tipinde parametre alan, boolean geri dönüş tipi olan, "test" isminde bir metoda sahiptir.

```
interface Predicate<T> {
   boolean test(T t);
}
```

Aggregate Operations

processElements Action	Aggregate Operation
Bir nesne kaynağı edinin	Stream<E> **stream**()
Bir Predicate nesnesiyle eşleşen nesneleri filtreleyin.	Stream<T> **filter**(Predicate<? super T> predicate)
Nesneleri, bir Function nesnesi tarafından belirtilen başka bir değerle eşleyin.	<R> Stream<R> **map**(Function<? super T,? extends R> mapper)
Bir Consumer nesnesi tarafından belirtilen bir eylemi gerçekleştirin.	void **forEach**(Consumer<? super T> action)