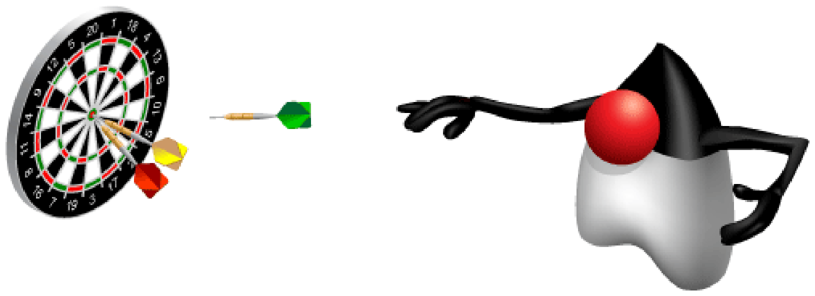
9

Lambda Built-in Functional Interfaces

Objectives

After completing this lesson, you should be able to:

- List the built-in interfaces included in `java.util.function`
- Use primitive versions of base interfaces
- Use binary versions of base interfaces



Built-in Functional Interfaces

- Lambda expressions rely on functional interfaces
  - Important to understand what an interface does
  - Concepts make using lambdas easier
- Focus on the purpose of main functional interfaces
- Become aware of many primitive variations
- Lambda expressions have properties like those of a variable
  - Use when needed
  - Can be stored and reused

The `java.util.function` Package

- `Predicate`: An expression that returns a `boolean`
- `Consumer`: An expression that performs operations on an object passed as argument and has a void return type
- `Function`: Transforms a T to a U
- `Supplier`: Provides an instance of a T (such as a factory)
- Primitive variations
- Binary variations

Example Assumptions

- The following two declarations are assumed for the examples that follow:

```
      List<SalesTxn> tList = SalesTxn.createTxnList();
      SalesTxn first = tList.get(0);
```

Predicate

```
package java.util.function;

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

```

Predicate: Example

```
    Predicate<SalesTxn> massSales =
        t -> t.getState().equals(State.MA);

    System.out.println("\n== Sales - Stream");
    tList.stream()
        .filter(massSales)
        .forEach(t -> t.printSummary());

    System.out.println("\n== Sales - Method Call");
    for(SalesTxn t:tList){
        if (massSales.test(t)){
            t.printSummary();
        }
    }
```

Consumer

```
package java.util.function;

public interface Consumer<T> {

    public void accept(T t);

}
```

Consumer: Example

```
Consumer<SalesTxn> buyerConsumer = t ->
    System.out.println("Id: " + t.getTxnId()
        + " Buyer: " + t.getBuyer().getName());

System.out.println("== Buyers - Lambda");
tList.stream().forEach(buyerConsumer);

System.out.println("== First Buyer - Method");
buyerConsumer.accept(first);
```

Function

```java
package java.util.function;

public interface Function<T,R> {

    public R apply(T t);
}

```

Function: Example

```
    Function<SalesTxn, String> buyerFunction =
        t -> t.getBuyer().getName();

    System.out.println("\n== First Buyer");
    System.out.println(buyerFunction.apply(first));
  }
```

Supplier

```
package java.util.function;

public interface Supplier<T> {

    public T get();
}

```

Supplier: Example

```
       List<SalesTxn> tList = SalesTxn.createTxnList();
       Supplier<SalesTxn> txnSupplier =
           () -> new SalesTxn.Builder()
               .txnId(101)
               .salesPerson("John Adams")
               .buyer(Buyer.getBuyerMap().get("PriceCo"))
               .product("Widget")
               .paymentType("Cash")
               .unitPrice(20)
//... Lines ommited
               .build();

       tList.add(txnSupplier.get());
       System.out.println("\n== TList");
       tList.stream().forEach(SalesTxn::printSummary);
```

Primitive Interface

- Primitive versions of all main interfaces
  - Will see these a lot in method calls
- Return a primitive
  - Example: `ToDoubleFunction`
- Consume a primitive
  - Example: `DoubleFunction`
- Why have these?
  - Avoids auto-boxing and unboxing

Return a Primitive Type

```
package java.util.function;

public interface ToDoubleFunction<T> {

    public double applyAsDouble(T t);
}

```

Return a Primitive Type: Example

```
    ToDoubleFunction<SalesTxn> discountFunction =
        t -> t.getTransactionTotal()
            * t.getDiscountRate();

    System.out.println("\n== Discount");
    System.out.println(
        discountFunction.applyAsDouble(first));
```

Process a Primitive Type

```
package java.util.function;

public interface DoubleFunction<R> {

    public R apply(double value);
}

```

Process Primitive Type: Example

```
    A06DoubleFunction test = new A06DoubleFunction();

    DoubleFunction<String> calc =
          t -> String.valueOf(t * 3);

    String result = calc.apply(20);
    System.out.println("New value is: " + result);
```

Binary Types

```
package java.util.function;

public interface BiPredicate<T, U> {

    public boolean test(T t, U u);
}

```

Binary Type: Example

```
List<SalesTxn> tList = SalesTxn.createTxnList();
SalesTxn first = tList.get(0);
String testState = "CA";

BiPredicate<SalesTxn,String> stateBiPred =
  (t, s) -> t.getState().getStr().equals(s);

System.out.println("\n== First is CA?");
System.out.println(
  stateBiPred.test(first, testState));
```

Unary Operator

```
package java.util.function;

public interface UnaryOperator<T> extends Function<T,T> {
      @Override
      public T apply(T t);
}
```

UnaryOperator: Example

- If you need to pass in something and return the same type, use the UnaryOperator interface.

```
    UnaryOperator<String> unaryStr =
        s -> s.toUpperCase();

      System.out.println("== Upper Buyer");
      System.out.println(
        unaryStr.apply(first.getBuyer().getName()));
```

Wildcard Generics Review

- Wildcards for generics are used extensively.
- `? super T`
  - This class and any of its super types
- `? extends T`
  - This class and any of its subtypes

Summary

After completing this lesson, you should be able to:

- List the built-in interfaces included in `java.util.function`
- Use primitive versions of base interfaces
- Use binary versions of base interfaces

Practice Overview

- Practice 9-1: Create Consumer Lambda Expression
- Practice 9-2: Create a Function Lambda Expression
- Practice 9-3: Create a Supplier Lambda Expression
- Practice 9-4: Create a BiPredicate Lambda Expression