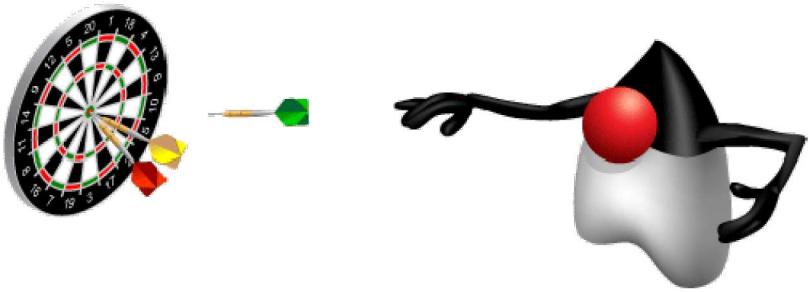
Lambda Built-in Functional Interfaces

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:

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

Predicate

```
1 package java.util.function;
2
3 public interface Predicate<T> {
4   public boolean test(T t);
5 }
6
```

Predicate: Example

```
16      Predicate<SalesTxn> massSales =
17          t -> t.getState().equals(State.MA);
18
19      System.out.println("\n== Sales - Stream");
20      tList.stream()
21          .filter(massSales)
22          .forEach(t -> t.printSummary());
23
24      System.out.println("\n== Sales - Method Call");
25      for(SalesTxn t:tList){
26          if (massSales.test(t)){
27              t.printSummary();
28          }
29      }
```

Consumer

```
1 package java.util.function;
2
3 public interface Consumer<T> {
4
5     public void accept(T t);
6
7 }
```

Consumer: Example

```
17      Consumer<SalesTxn> buyerConsumer = t ->
18          System.out.println("Id: " + t.getTxnId()
19              + " Buyer: " + t.getBuyer().getName());
20
21      System.out.println("== Buyers - Lambda");
22      tList.stream().forEach(buyerConsumer);
23
24      System.out.println("== First Buyer - Method");
25      buyerConsumer.accept(first);
```

Function

```
1 package java.util.function;
2
3 public interface Function<T,R> {
4
5     public R apply(T t);
6 }
7
```

Function: Example

```
17      Function<SalesTxn, String> buyerFunction =
18          t -> t.getBuyer().getName();
19
20      System.out.println("\n== First Buyer");
21      System.out.println(buyerFunction.apply(first));
22  }
```

Supplier

```
1 package java.util.function;
2
3 public interface Supplier<T> {
4
5     public T get();
6 }
7
```

Supplier: Example

```
15      List<SalesTxn> tList = SalesTxn.createTxnList();
16      Supplier<SalesTxn> txnSupplier =
17          () -> new SalesTxn.Builder()
18                .txnId(101)
19                .salesPerson("John Adams")
20                .buyer(Buyer.getBuyerMap().get("PriceCo"))
21                .product("Widget")
22                .paymentType("Cash")
23                .unitPrice(20)
//... Lines ommited
29                .build();
30
31      tList.add(txnSupplier.get());
32      System.out.println("\n== TList");
33      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

```
1 package java.util.function;
2
3 public interface ToDoubleFunction<T> {
4
5     public double applyAsDouble(T t);
6 }
7
```

Return a Primitive Type: Example

```
18      ToDoubleFunction<SalesTxn> discountFunction =
19          t -> t.getTransactionTotal()
20              * t.getDiscountRate();
21
22      System.out.println("\n== Discount");
23      System.out.println(
24          discountFunction.applyAsDouble(first));
```

Process a Primitive Type

```
1 package java.util.function;
2
3 public interface DoubleFunction<R> {
4
5     public R apply(double value);
6 }
7
```

Process Primitive Type: Example

```
 9      A06DoubleFunction test = new A06DoubleFunction();
10
11      DoubleFunction<String> calc =
12            t -> String.valueOf(t * 3);
13
14      String result = calc.apply(20);
15      System.out.println("New value is: " + result);
```

Binary Types

```
1 package java.util.function;
2
3 public interface BiPredicate<T, U> {
4
5     public boolean test(T t, U u);
6 }
7
```

Binary Type: Example

```
14      List<SalesTxn> tList = SalesTxn.createTxnList();
15      SalesTxn first = tList.get(0);
16      String testState = "CA";
17
18      BiPredicate<SalesTxn,String> stateBiPred =
19        (t, s) -> t.getState().getStr().equals(s);
20
21      System.out.println("\n== First is CA?");
22      System.out.println(
23        stateBiPred.test(first, testState));
```

Unary Operator

```
1 package java.util.function;
2
3 public interface UnaryOperator<T> extends
Function<T,T> {
4     @Override
5     public T apply(T t);
6 }
```

UnaryOperator: Example

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

```
17      UnaryOperator<String> unaryStr =
18          s -> s.toUpperCase();
19
20        System.out.println("== Upper Buyer");
21        System.out.println(
22          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 4-1: Create Consumer Lambda Expression
- Practice 4-2: Create a Function Lambda Expression
- Practice 4-3: Create a Supplier Lambda Expression
- Practice 4-4: Create a BiPredicate Lambda Expression