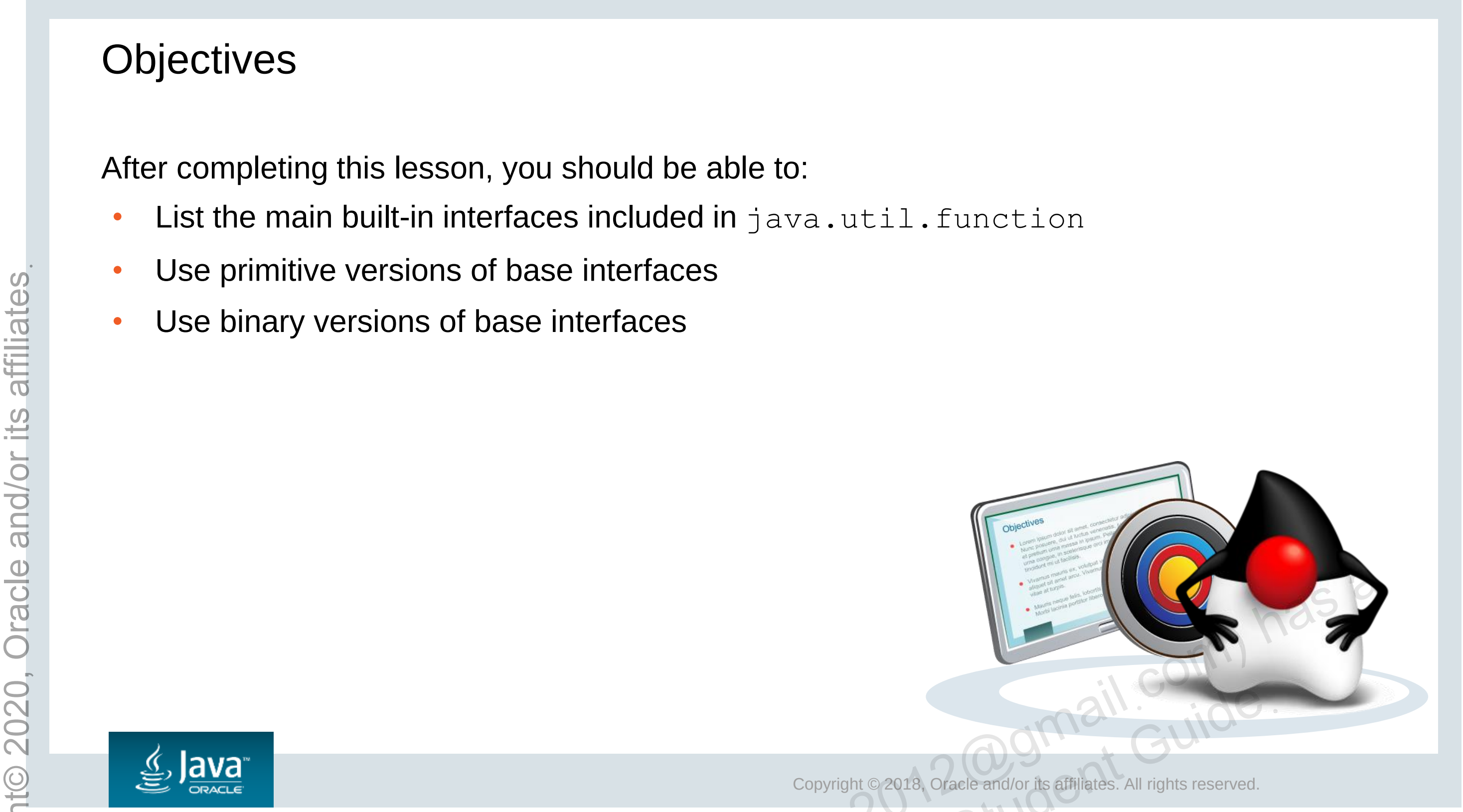
8

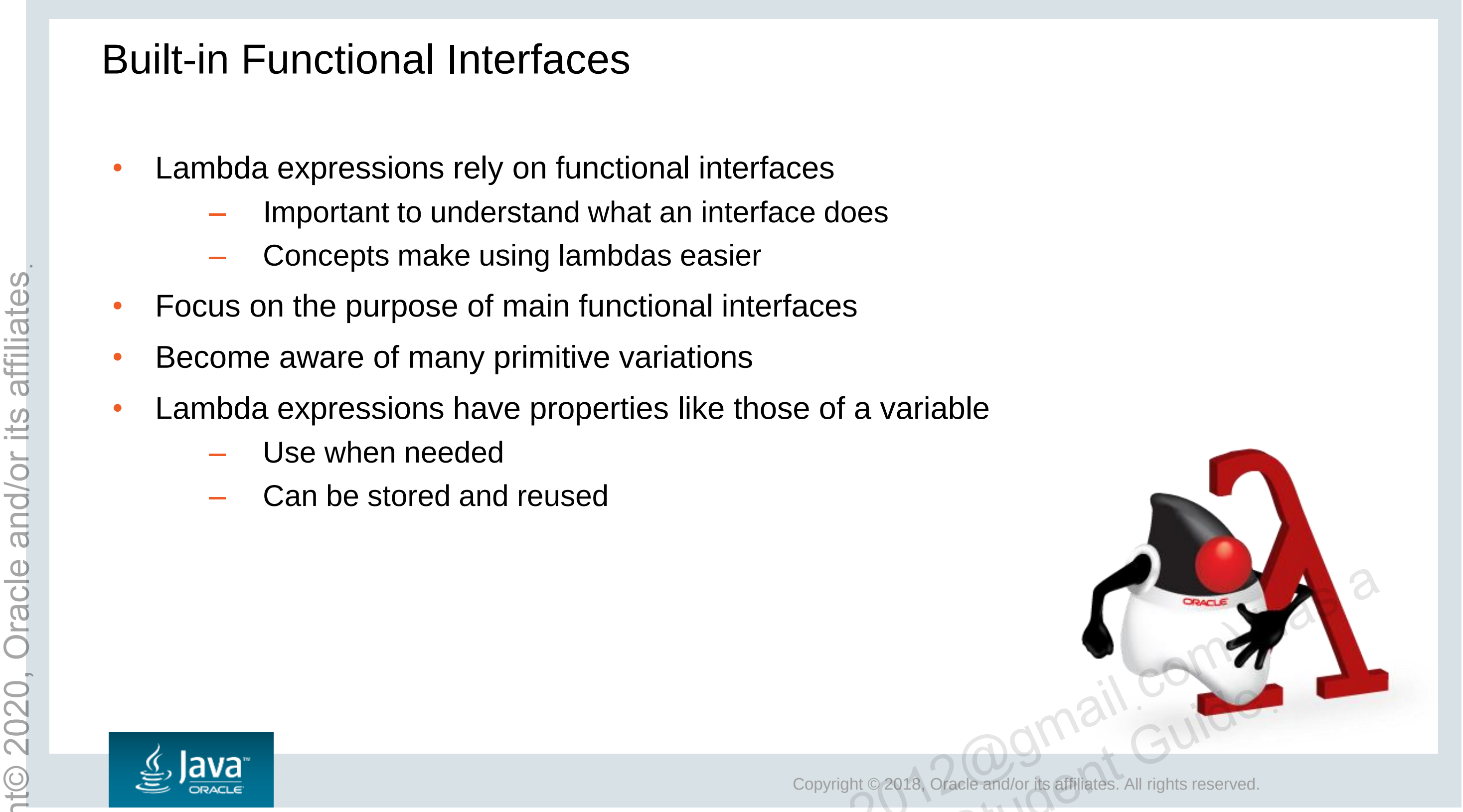
Lambda Built-in Functional Interfaces



Objectives

After completing this lesson, you should be able to:

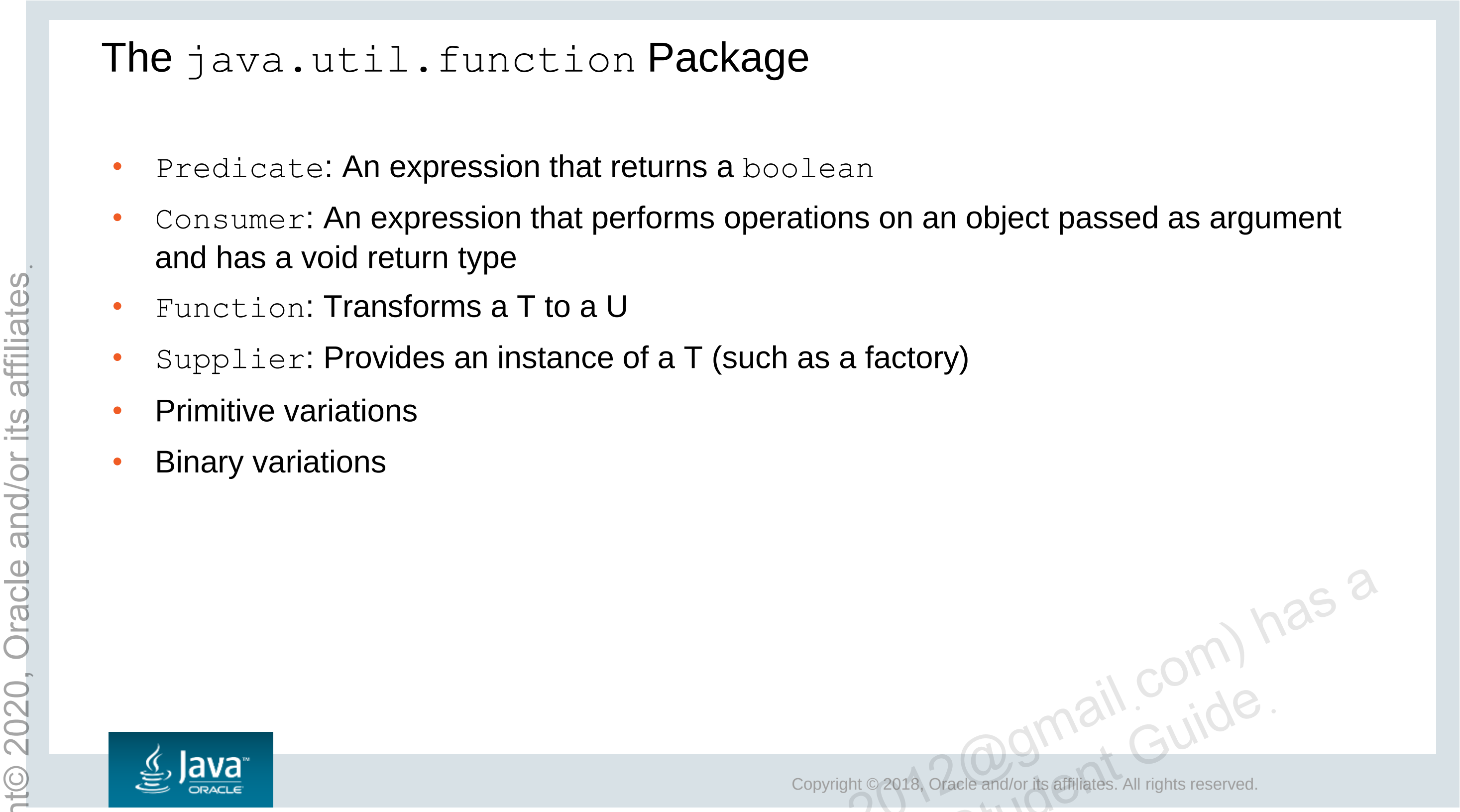
- List the main built-in interfaces included in `java.util.function`
- Use primitive versions of base interfaces
- Use binary versions of base interfaces



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

There are a lot of method signatures that refer to interfaces in `java.util.function`. Therefore, it is important to understand what these interfaces do and what variations on the basics exist. It makes writing lambda expressions a lot easier.



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

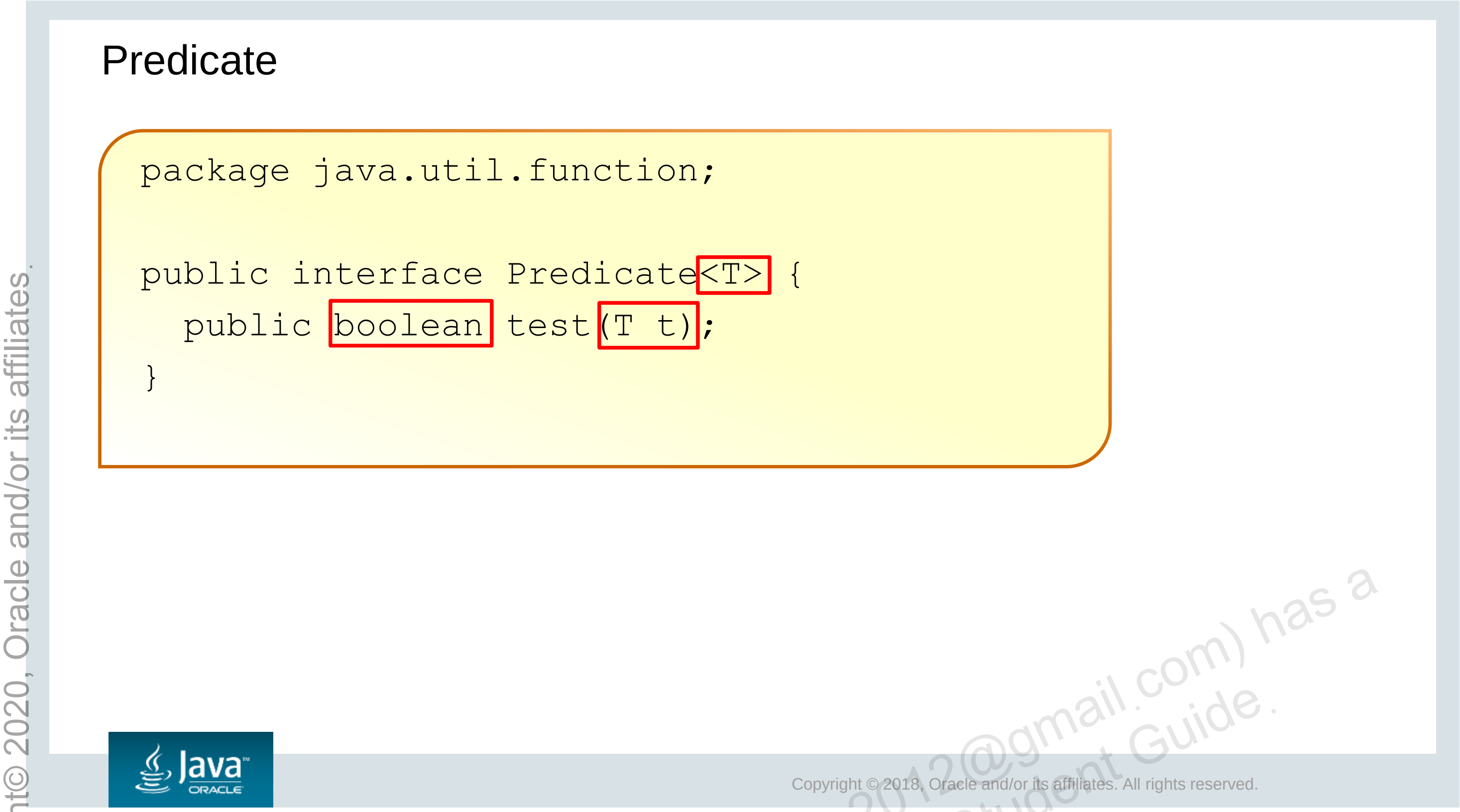
Predicate is not the only functional interface provided with Java. A number of standard interfaces are designed as a starter set for developers.

Example Assumptions

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

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

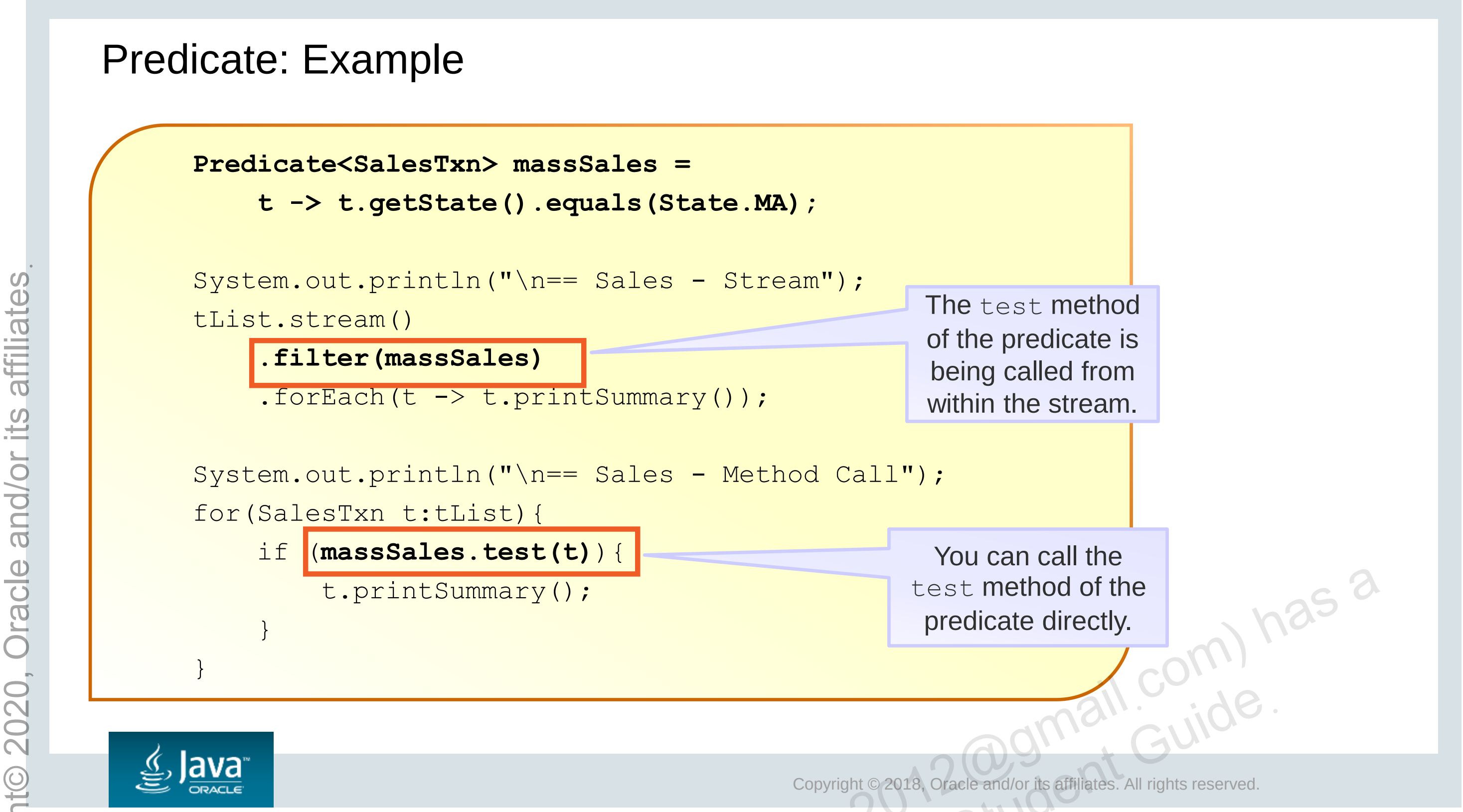
One or both of the declarations pictured are assumed in the examples that follow.



Predicate

```
package java.util.function;

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

A `Predicate` takes a generic class and returns a `boolean`. It has a single method, namely `test`.



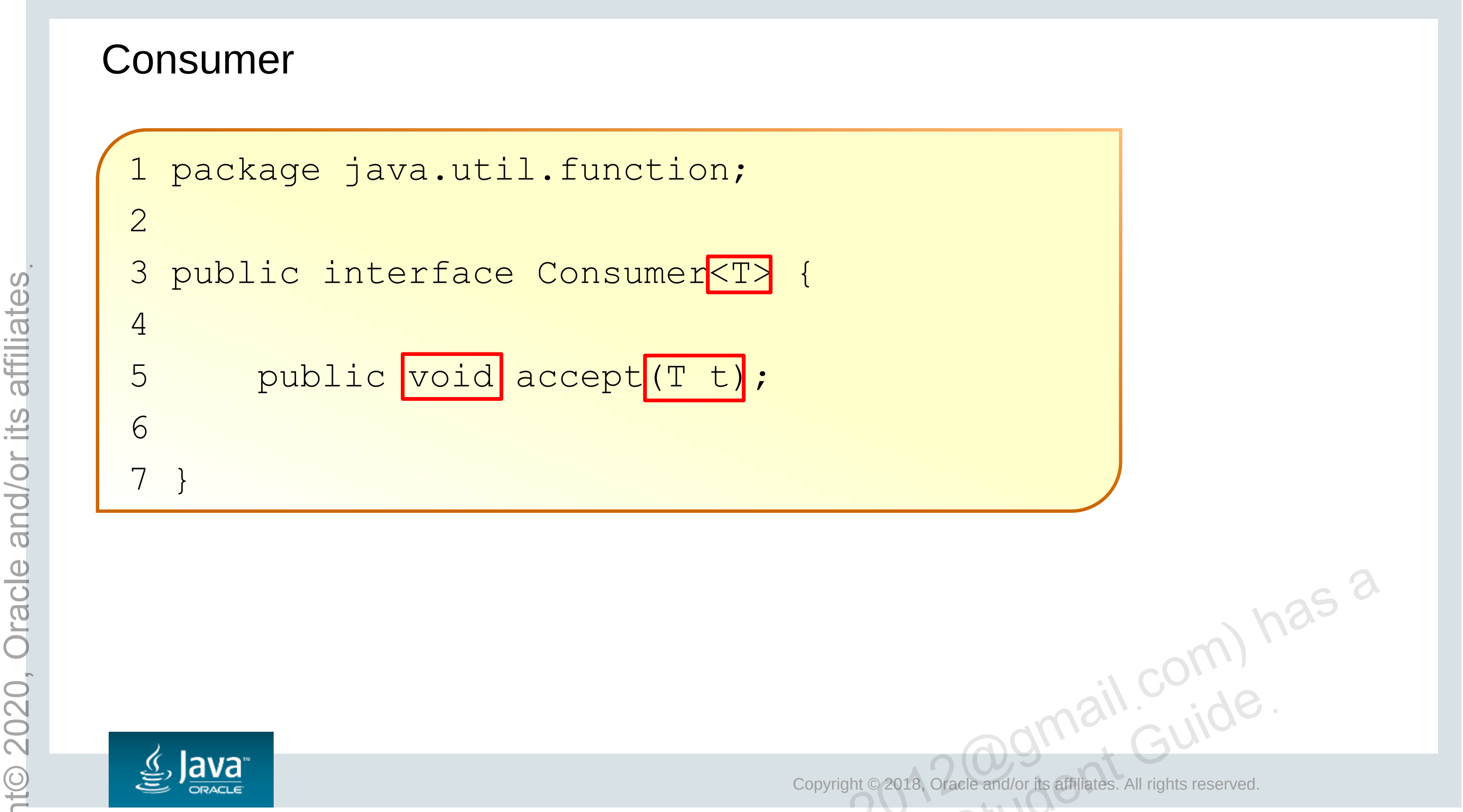
Predicate: Example

```
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();
    }
}
```

The `test` method of the predicate is being called from within the stream.

You can call the `test` method of the predicate directly.

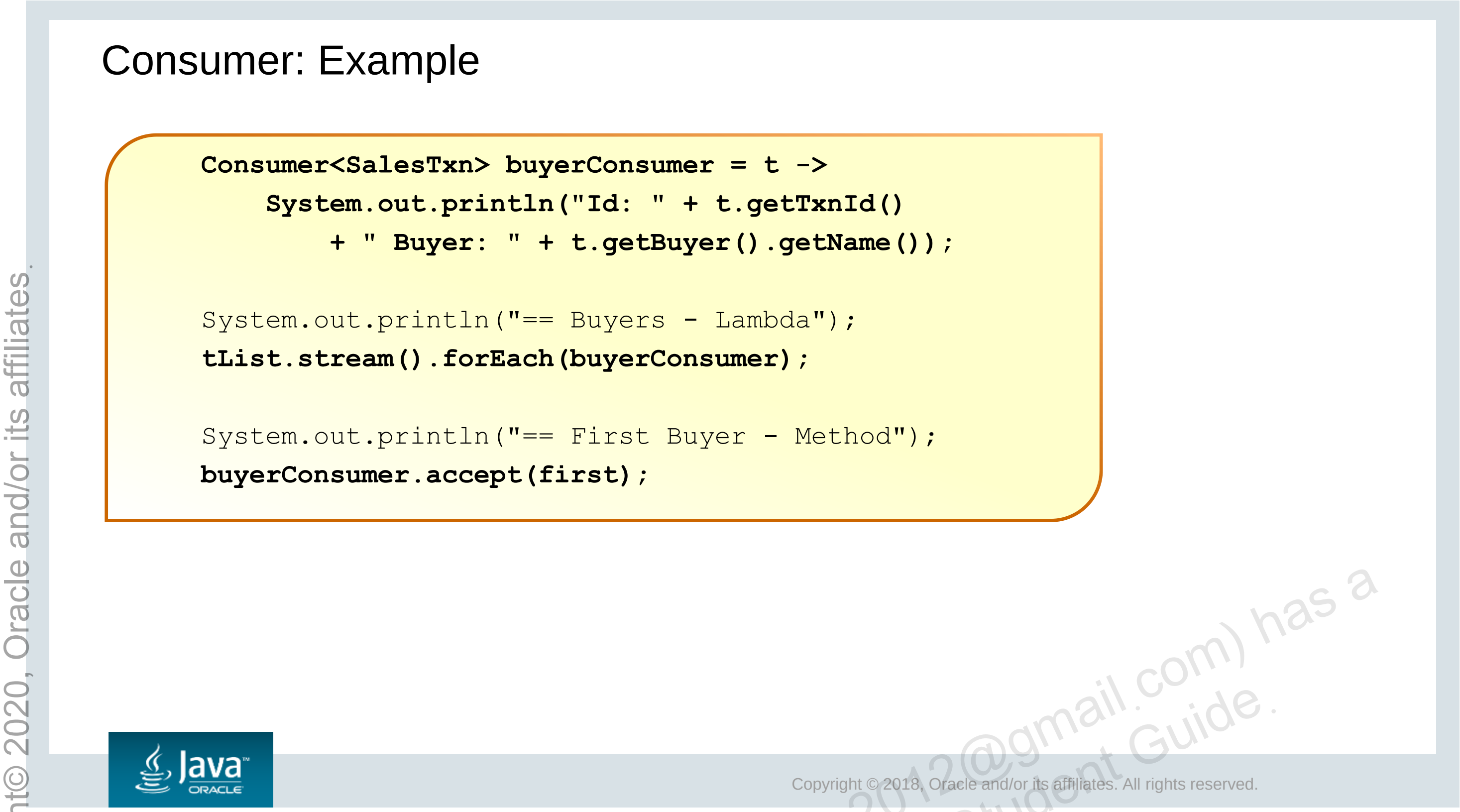
In this example, a `SalesTxn` is tested to see if it was executed in the state of MA. The `filter` method takes a predicate as a parameter. In the second example, notice that the predicate can call its `test` method with a `SalesTxn` object as a parameter. This is what the stream does internally.



Consumer

```java
package java.util.function;

public interface Consumer<T> {

    public void accept(T t);

}
```

A `Consumer` takes a generic and returns nothing. It has a single method `accept`.

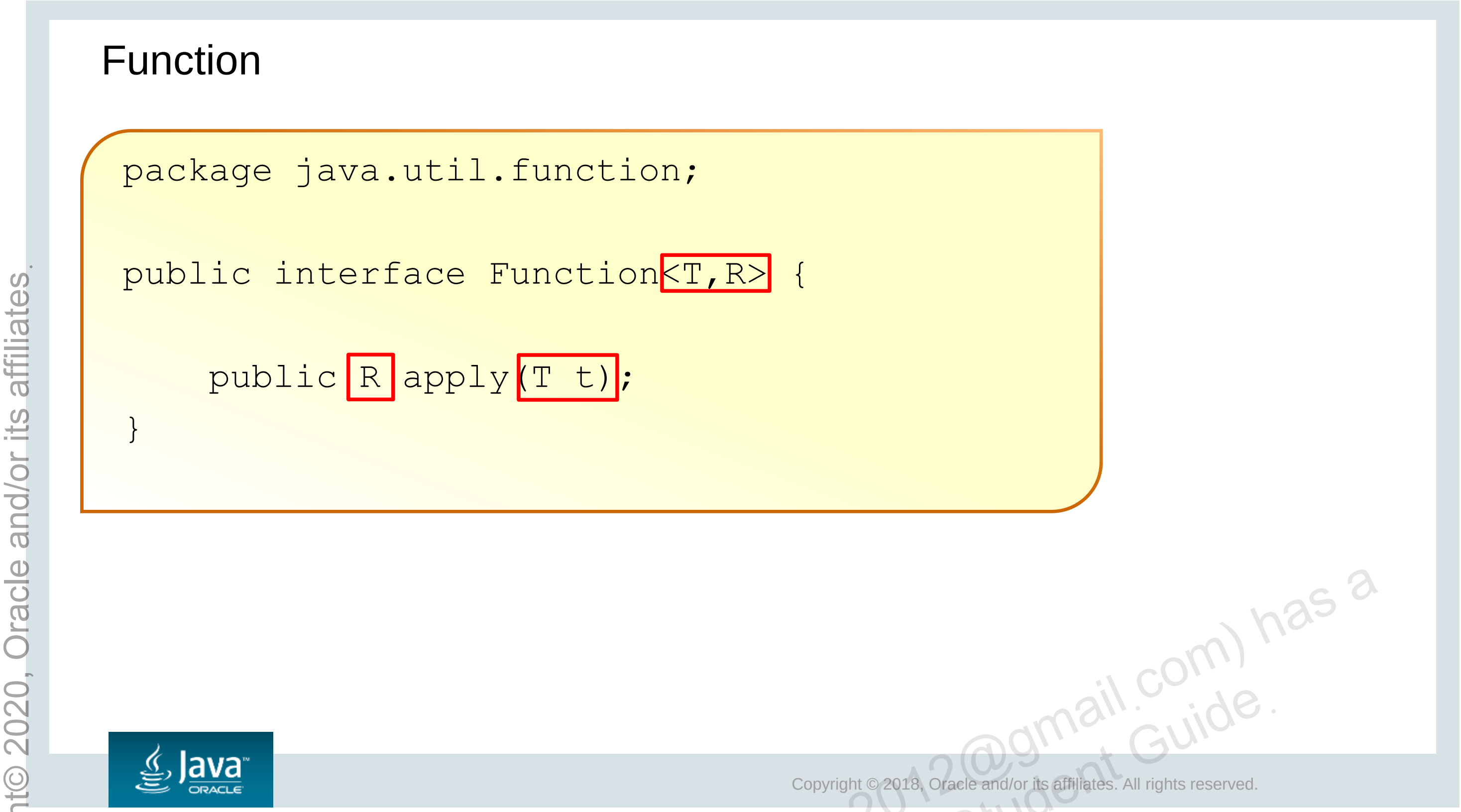


Consumer: Example

```
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);
```

Note how the `Consumer` is defined and nothing is returned. The example takes a sales transaction and prints a couple values.

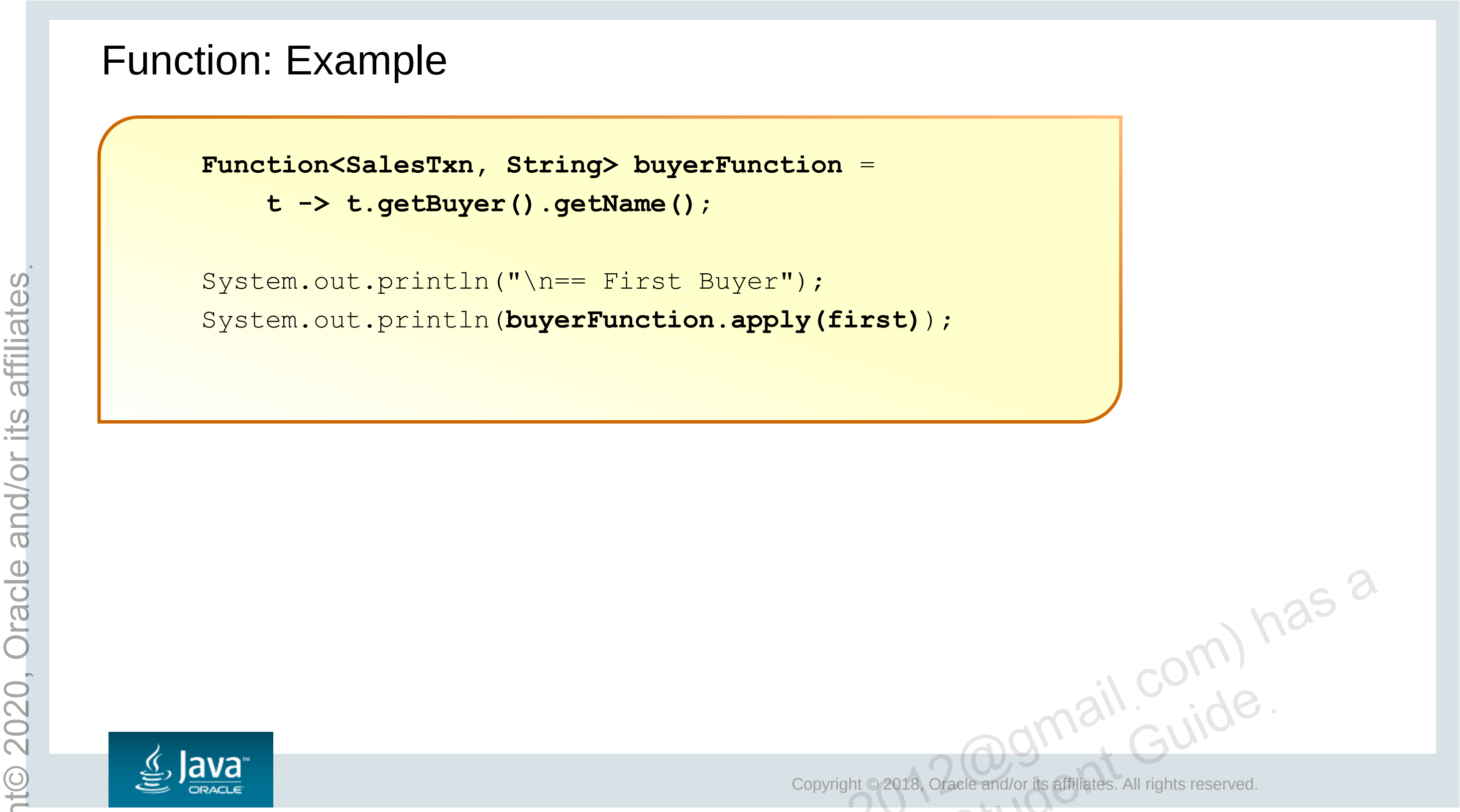
Two examples are provided in the slide. The first shows that the default parameter for `forEach` is `Consumer`. The second shows that once a lambda expression is stored, it can be executed on the specified type by using the `accept` method.



Function

```
package java.util.function;

public interface Function<T,R> {

    public R apply(T t);
}
```

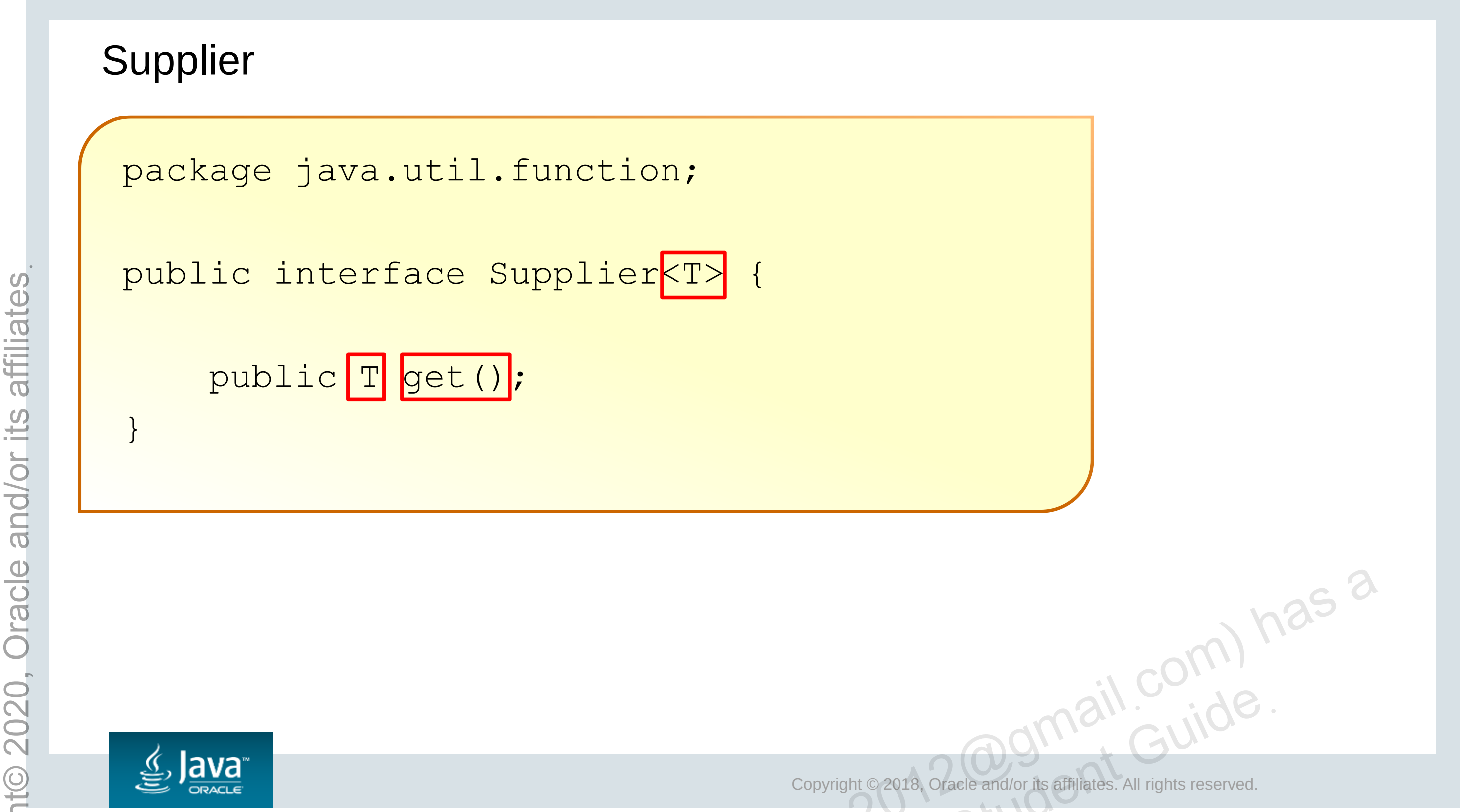
A `Function` takes one generic type and returns another. Notice that the input type comes first in the list and then the return type. So the `apply` method takes a T and returns an R.



Function: Example

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

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

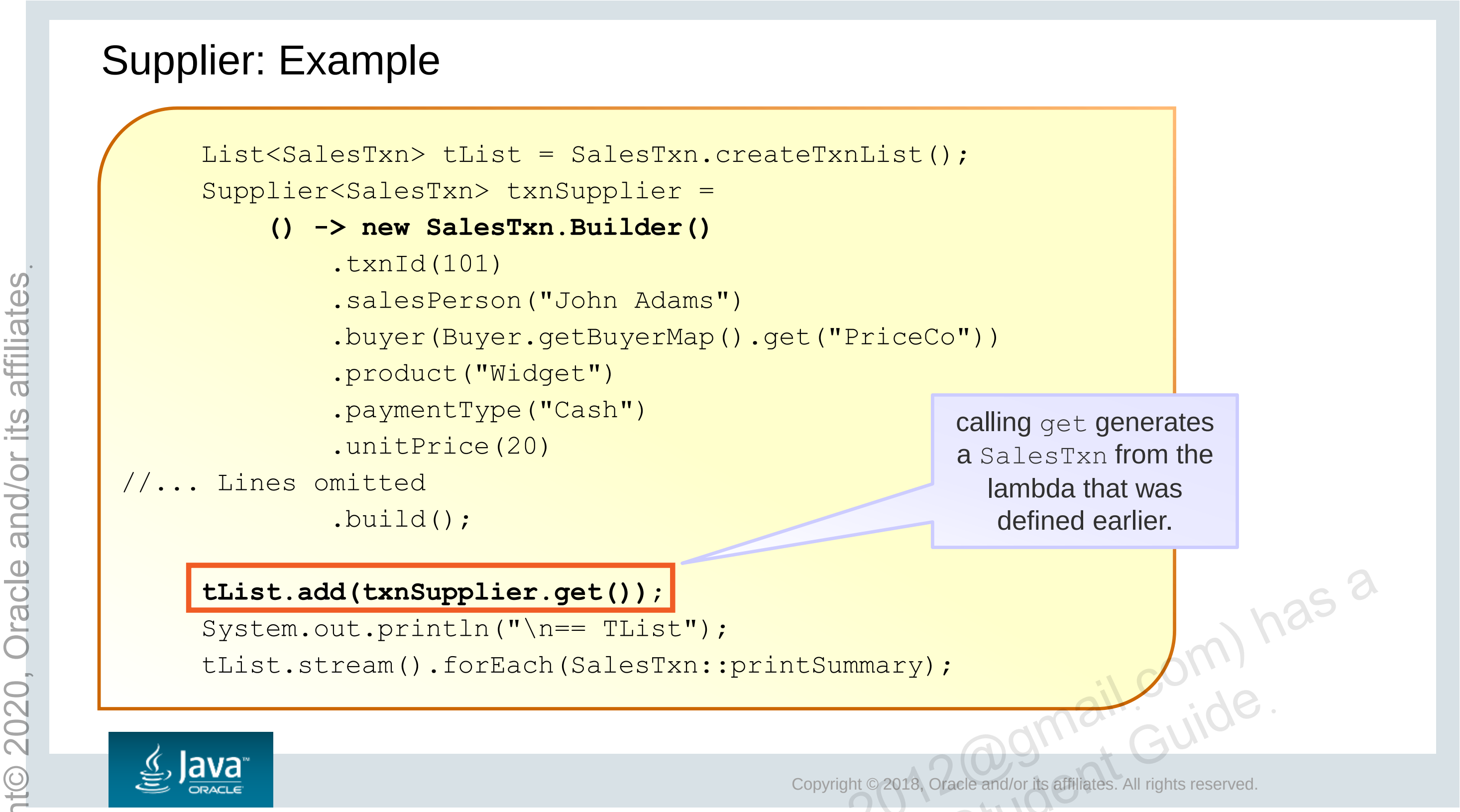
The example takes a `SalesTxn` and returns a `String`. The `Function` interface is used frequently in the update `Collection` APIs.



Supplier

```
package java.util.function;

public interface Supplier<T> {

    public T get();
}
```

The `Supplier` returns a generic type and takes no parameters.

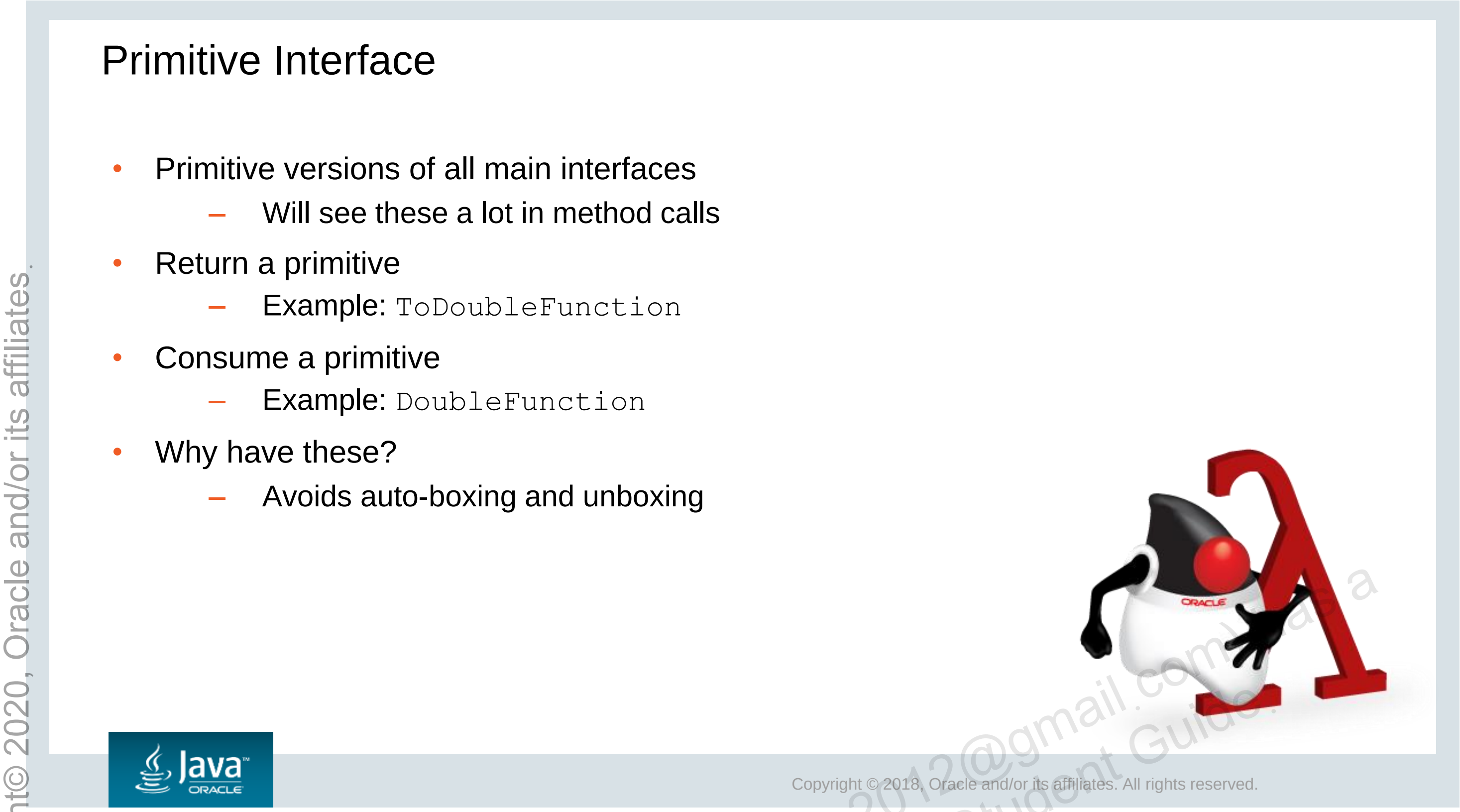


Supplier: Example

```
     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 omitted
             .build();

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

calling `get` generates a `SalesTxn` from the lambda that was defined earlier.

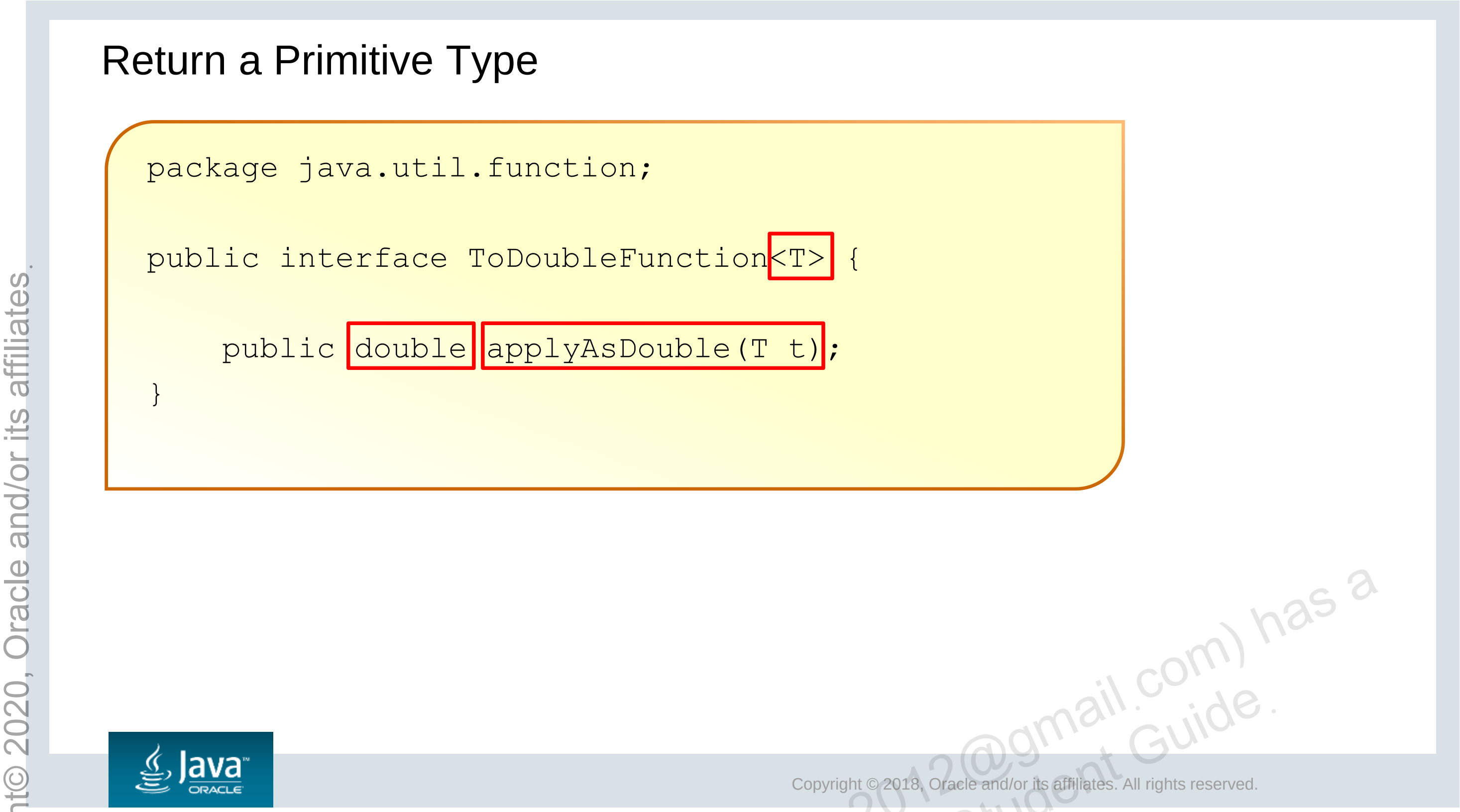
In the example, the `Supplier` creates a new `SalesTxn`.



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

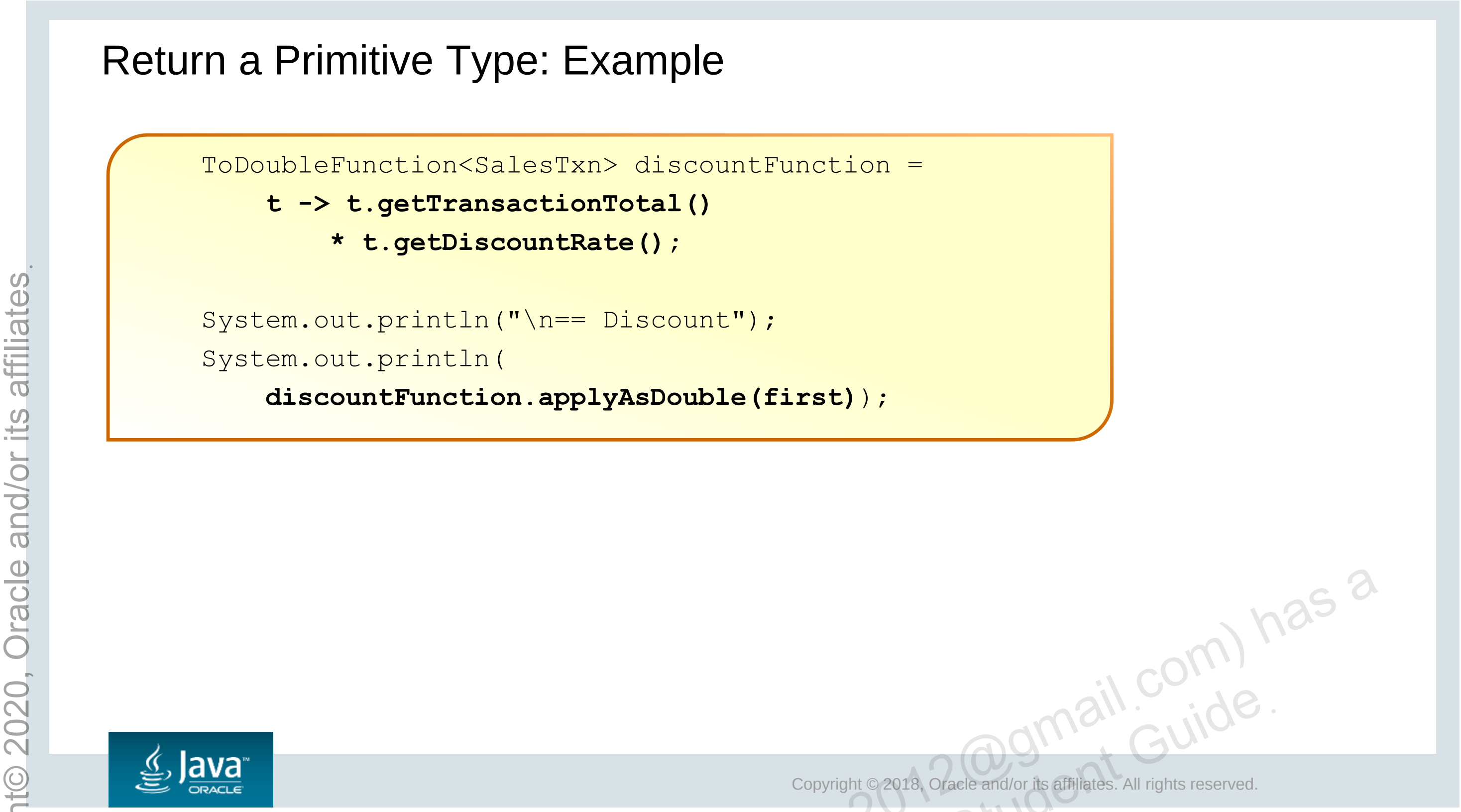
If you look at the API docs, there are a number of primitive interfaces that mirror the main types: `Predicate`, `Consumer`, `Function`, `Supplier`. These are provided to avoid the negative performance consequences of auto-boxing and unboxing.



Return a Primitive Type

```
package java.util.function;

public interface ToDoubleFunction<T> {

    public double applyAsDouble(T t);
}
```

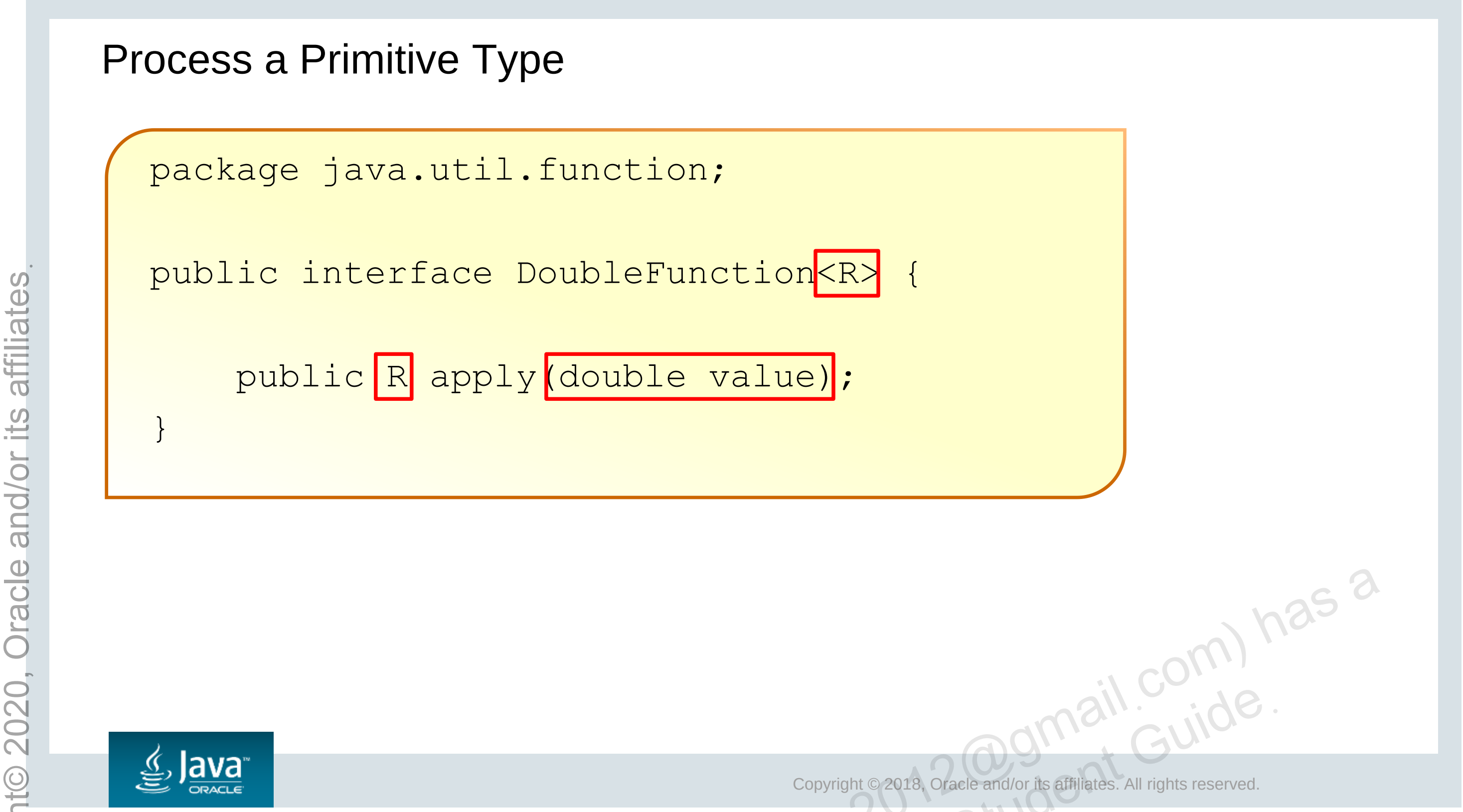
The `ToDoubleFunction` interface takes a generic type and returns a `double`.



Return a Primitive Type: Example

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

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

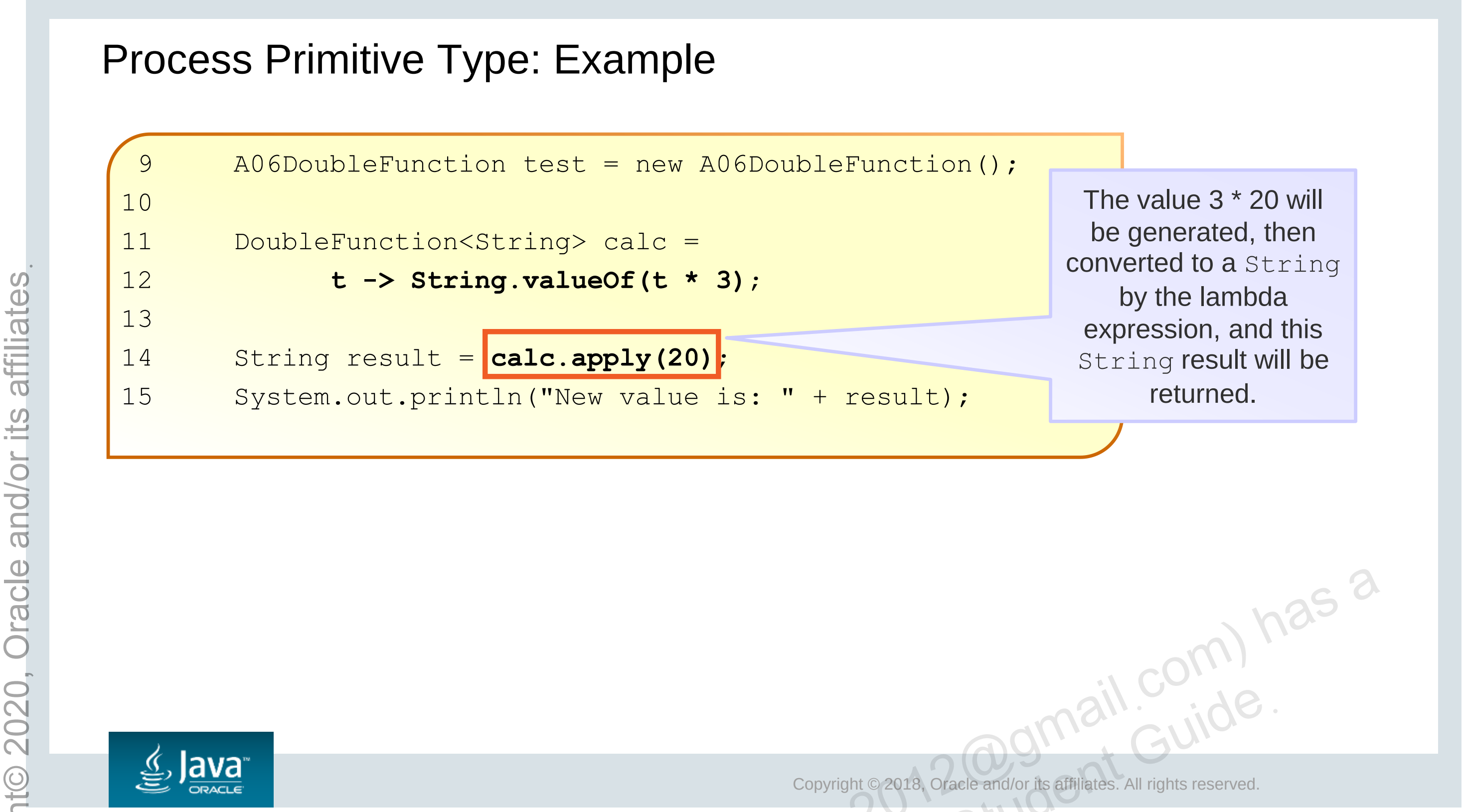
This example calculates a value from a transaction and returns a `double`. Notice that the method name changes a little, but this is still a `Function`. Pass in one type and return something else, in this case a `double`.



Process a Primitive Type

```
package java.util.function;

public interface DoubleFunction<R> {

    public R apply(double value);
}
```

Notice that a `DoubleFunction` specifies only one generic type, but a `Function` takes two. The `apply` method takes a `double` and returns the generic type. So the `double`, in this case, is the input and the generic type is the output.

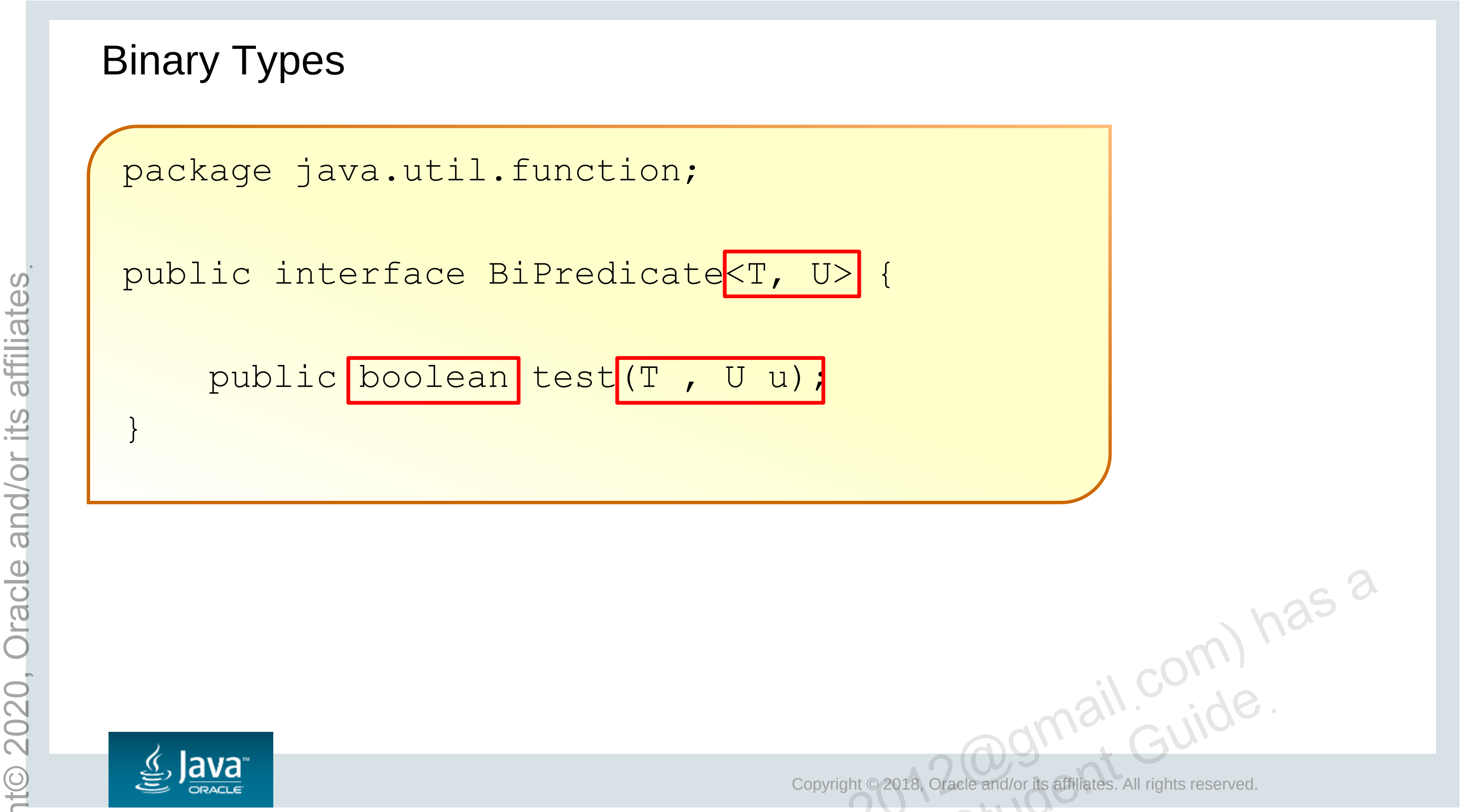


Process Primitive Type: Example

```
A06DoubleFunction test = new A06DoubleFunction();

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

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

The value 3 * 20 will be generated, then converted to a String by the lambda expression, and this String result will be returned.

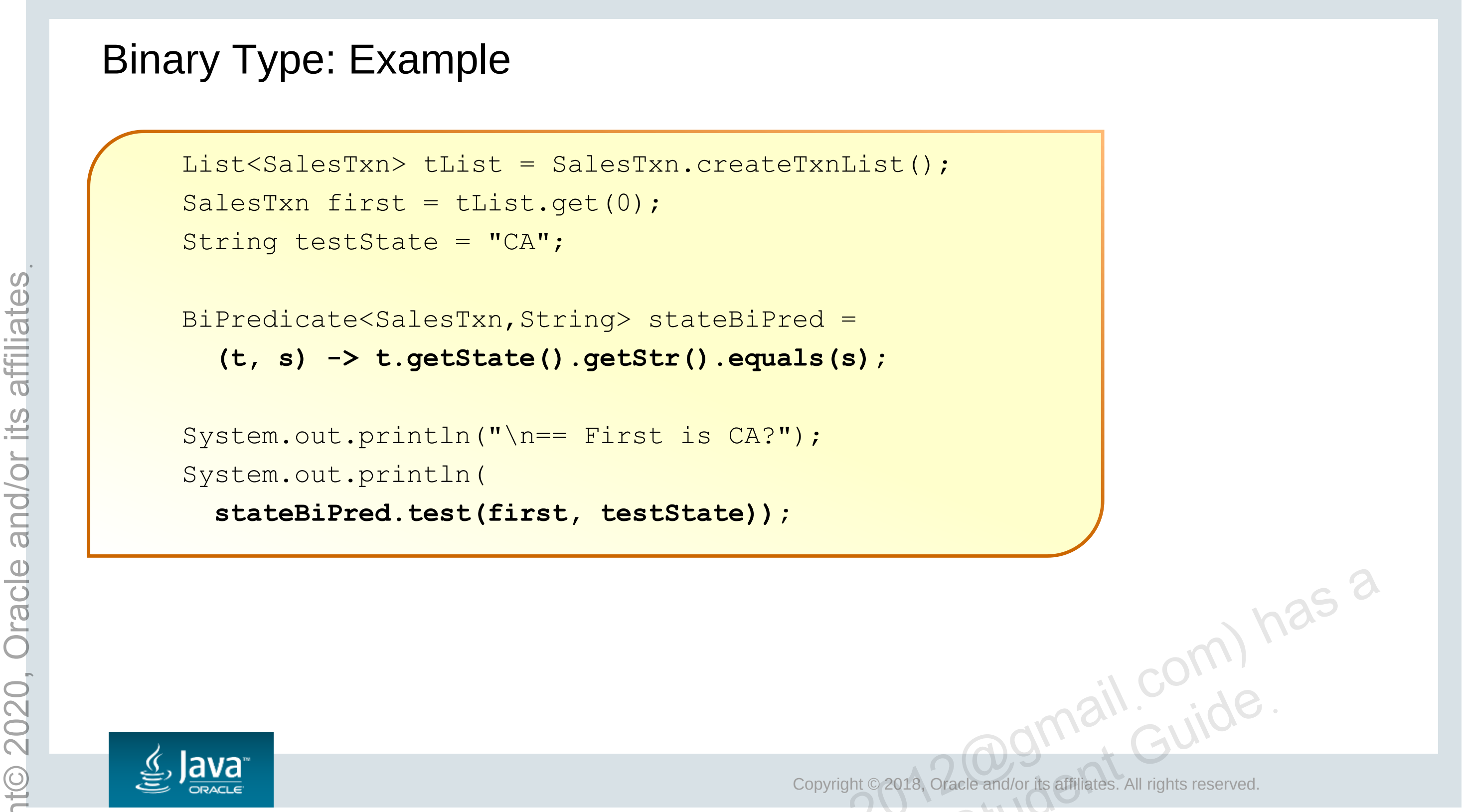
The example computes a value and then returns the result as a `String`.



Binary Types

```
package java.util.function;

public interface BiPredicate<T, U> {

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

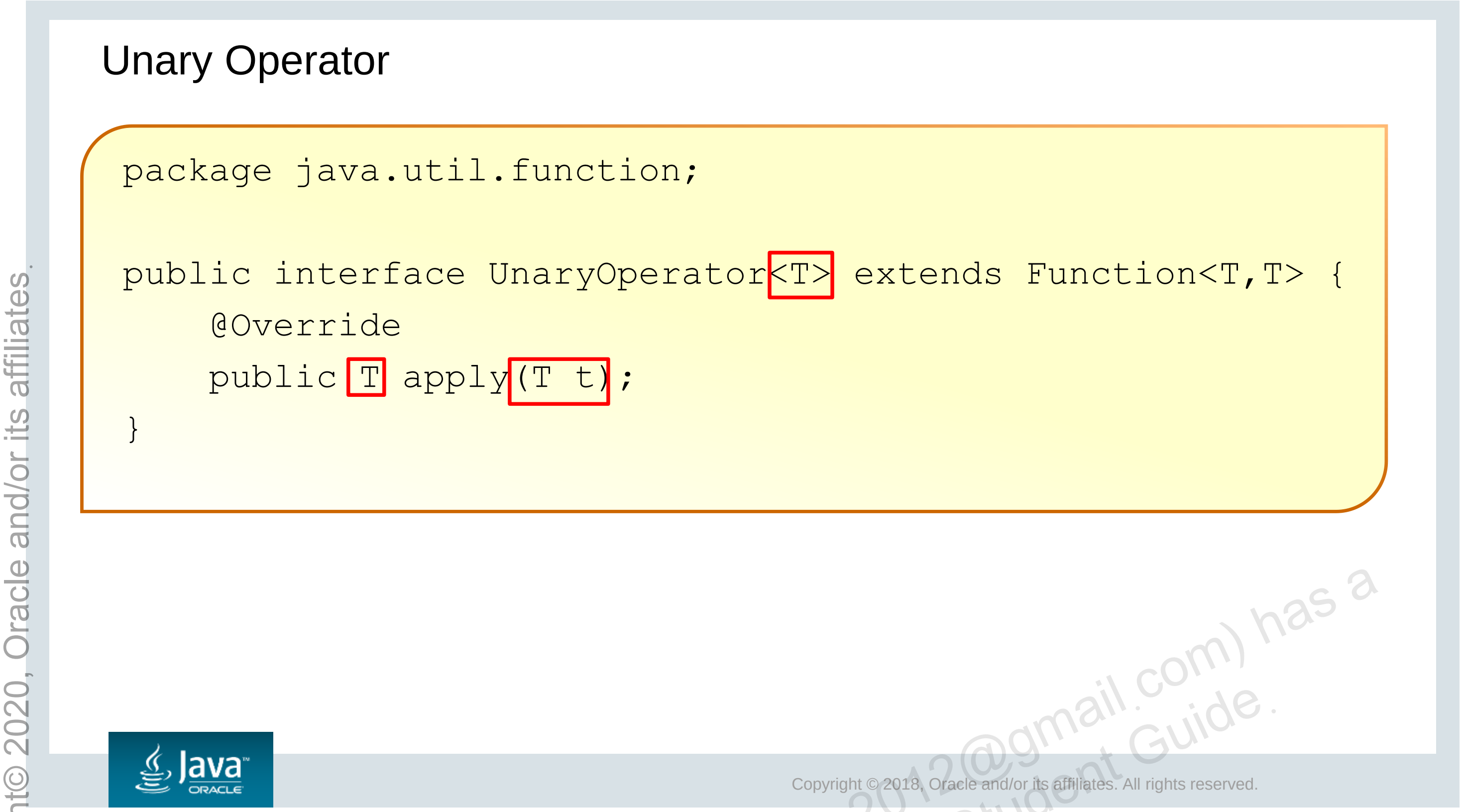
The binary version of the standard interfaces allows two generic types as input. In this example, the `BiPredicate` takes two parameters and returns a `boolean`.



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));
```

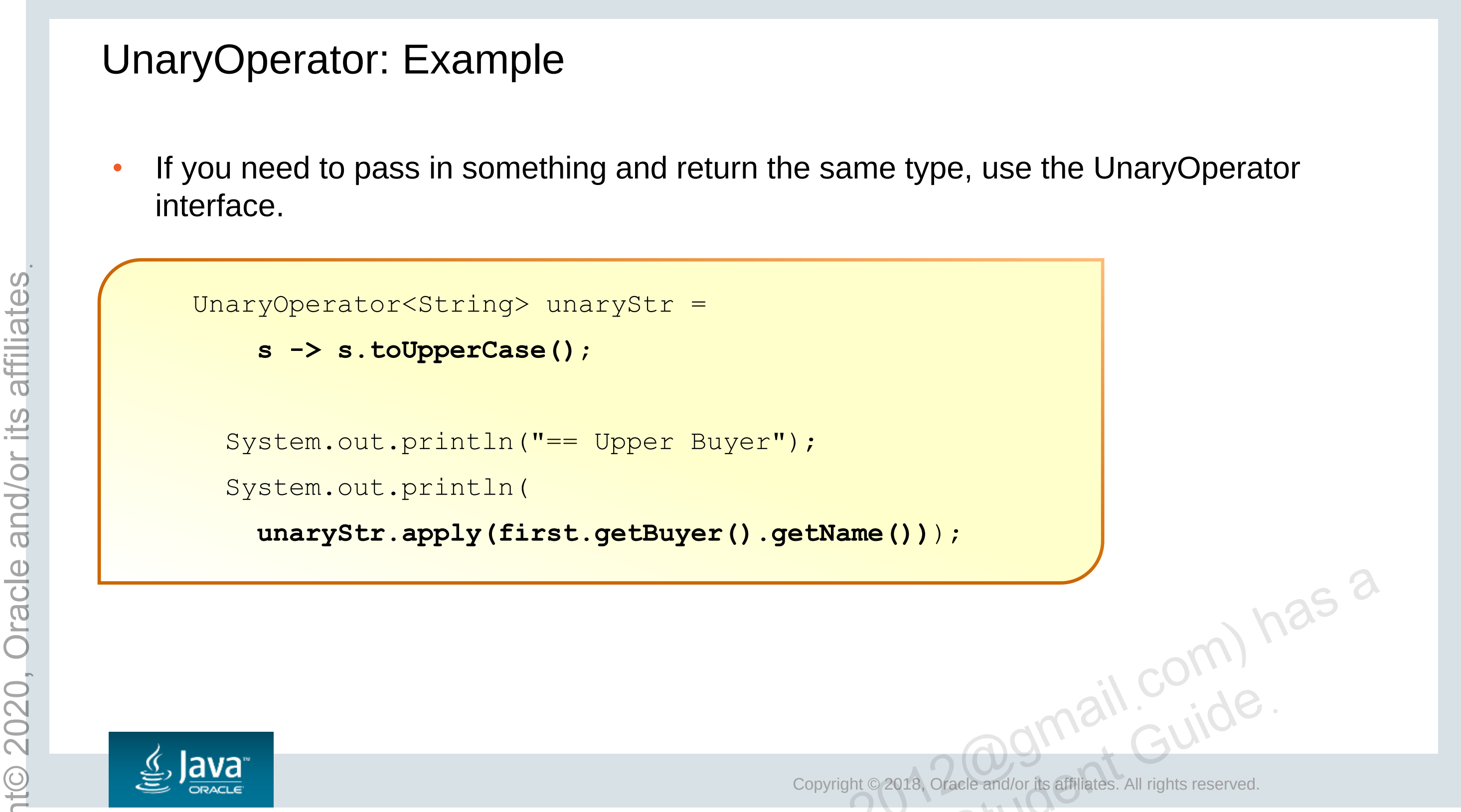
This example takes a `SalesTxn` and a `String` to do a comparison and return a result. The `test` method merely takes two parameters instead of one.



Unary Operator

```
package java.util.function;

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

The `UnaryOperator` takes a class as input and returns an object of the same class.



UnaryOperator: Example

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

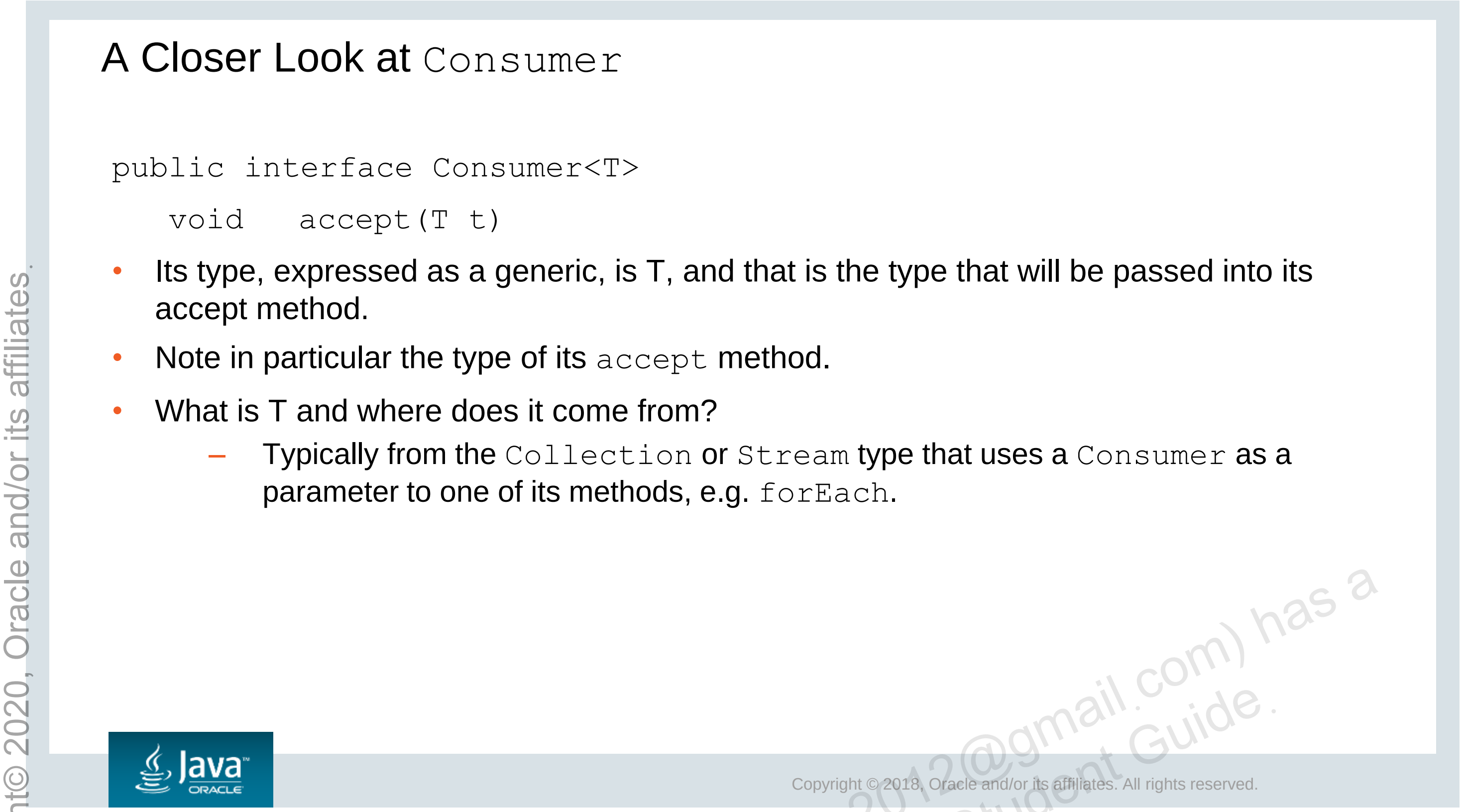
```
UnaryOperator<String> unaryStr =
    s -> s.toUpperCase();

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

The `UnaryOperator` interface takes a generic type and returns that same type. This example takes a `String` and returns the `String` in uppercase.

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

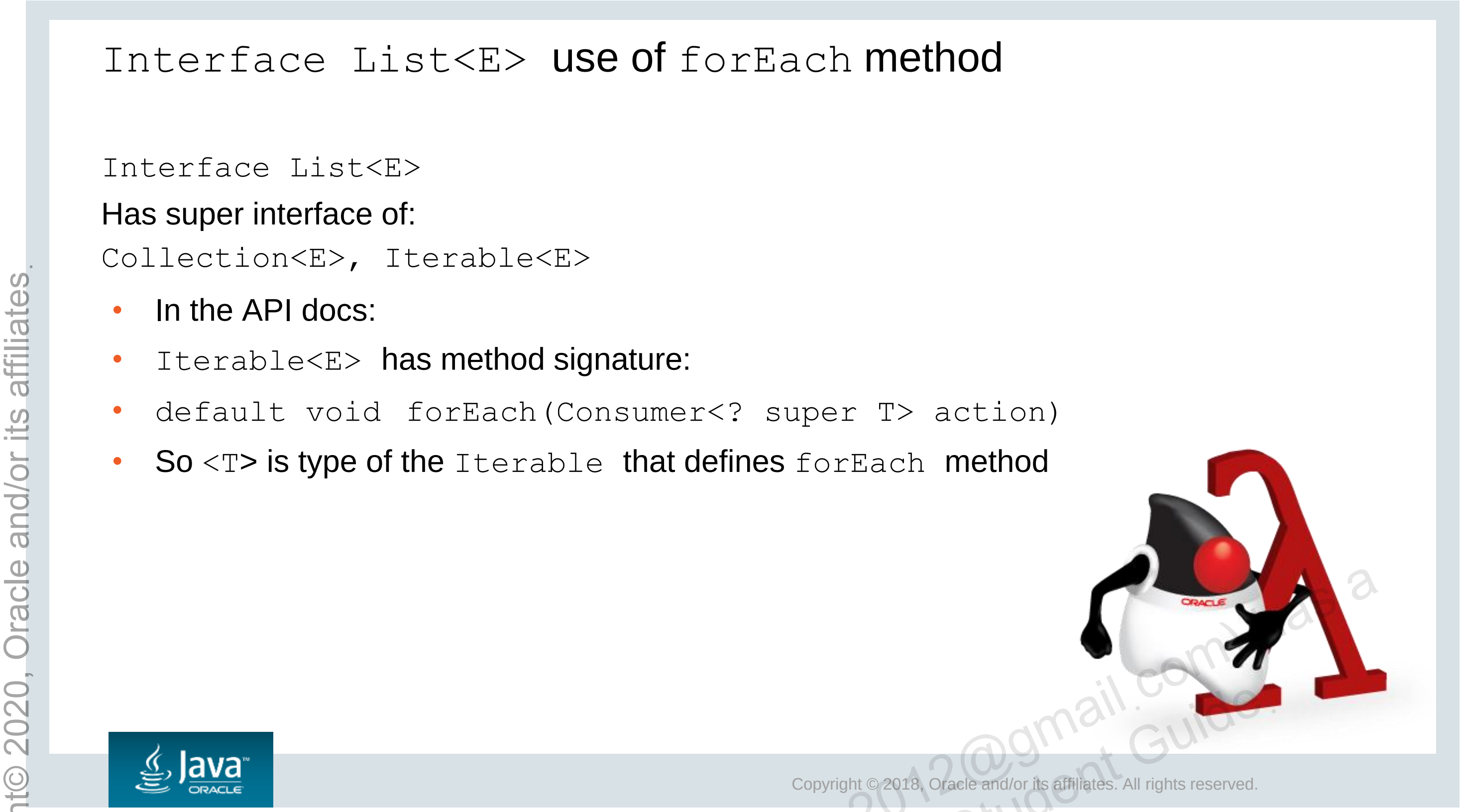
When using the built-in functional interfaces, generic wildcard statements are used frequently. The two most common wildcards you will see are listed in the slide.



A Closer Look at `Consumer`

```
public interface Consumer<T>
   void   accept(T t)
```

- Its type, expressed as a generic, is T, and that is the type that will be passed into its accept method.
- Note in particular the type of its `accept` method.
- What is T and where does it come from?
  - Typically from the `Collection` or `Stream` type that uses a `Consumer` as a parameter to one of its methods, e.g. `forEach`.



`Interface List<E>` use of `forEach` method

`Interface List<E>`

Has super interface of:

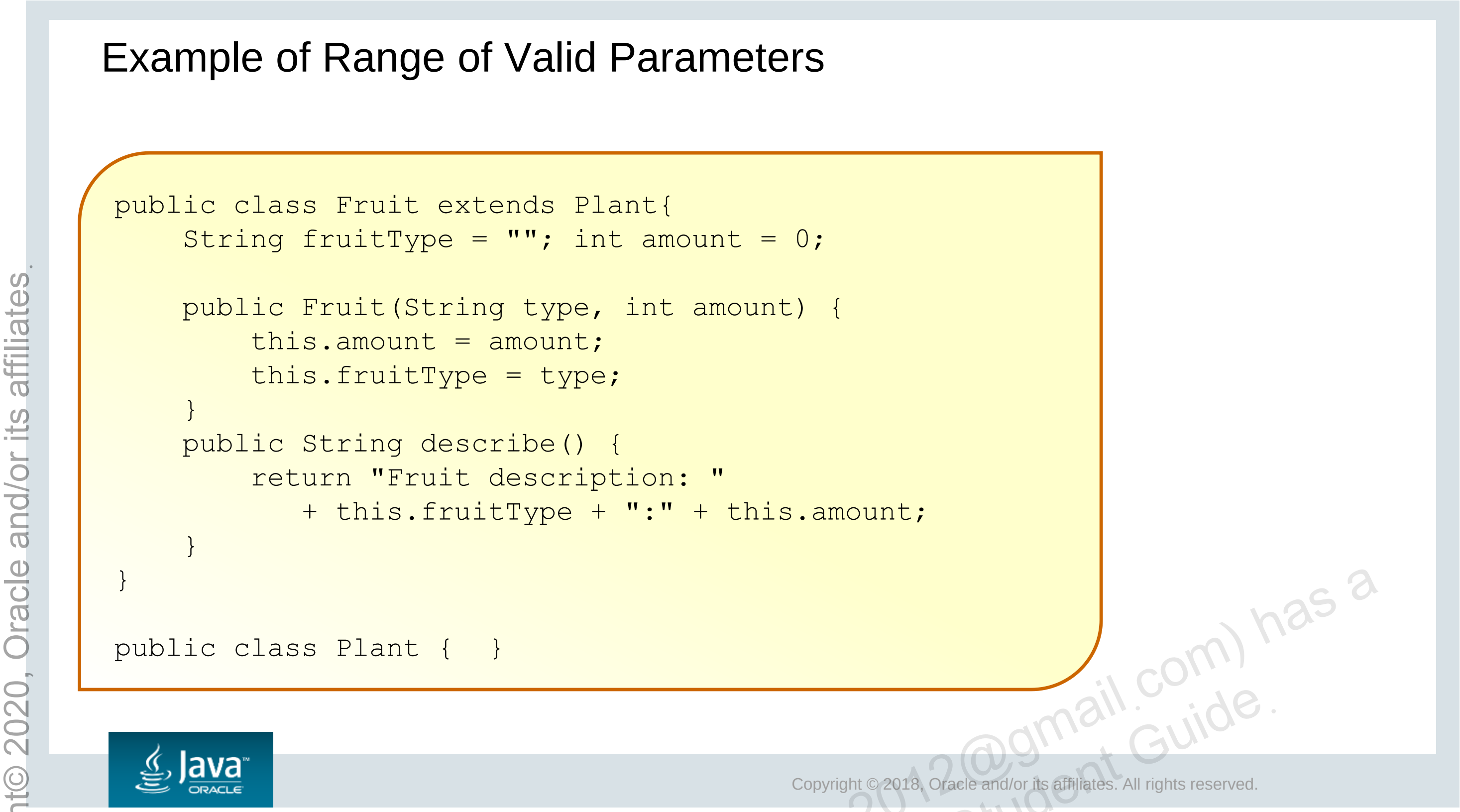
`Collection<E>, Iterable<E>`

- In the API docs:
- `Iterable<E>` has method signature:
- `default void forEach(Consumer<? super T> action)`
- So `<T>` is type of the `Iterable` that defines `forEach` method

Note then that the Consumer passed into the forEach method is the type of the List or Stream OR a supertype of that type.

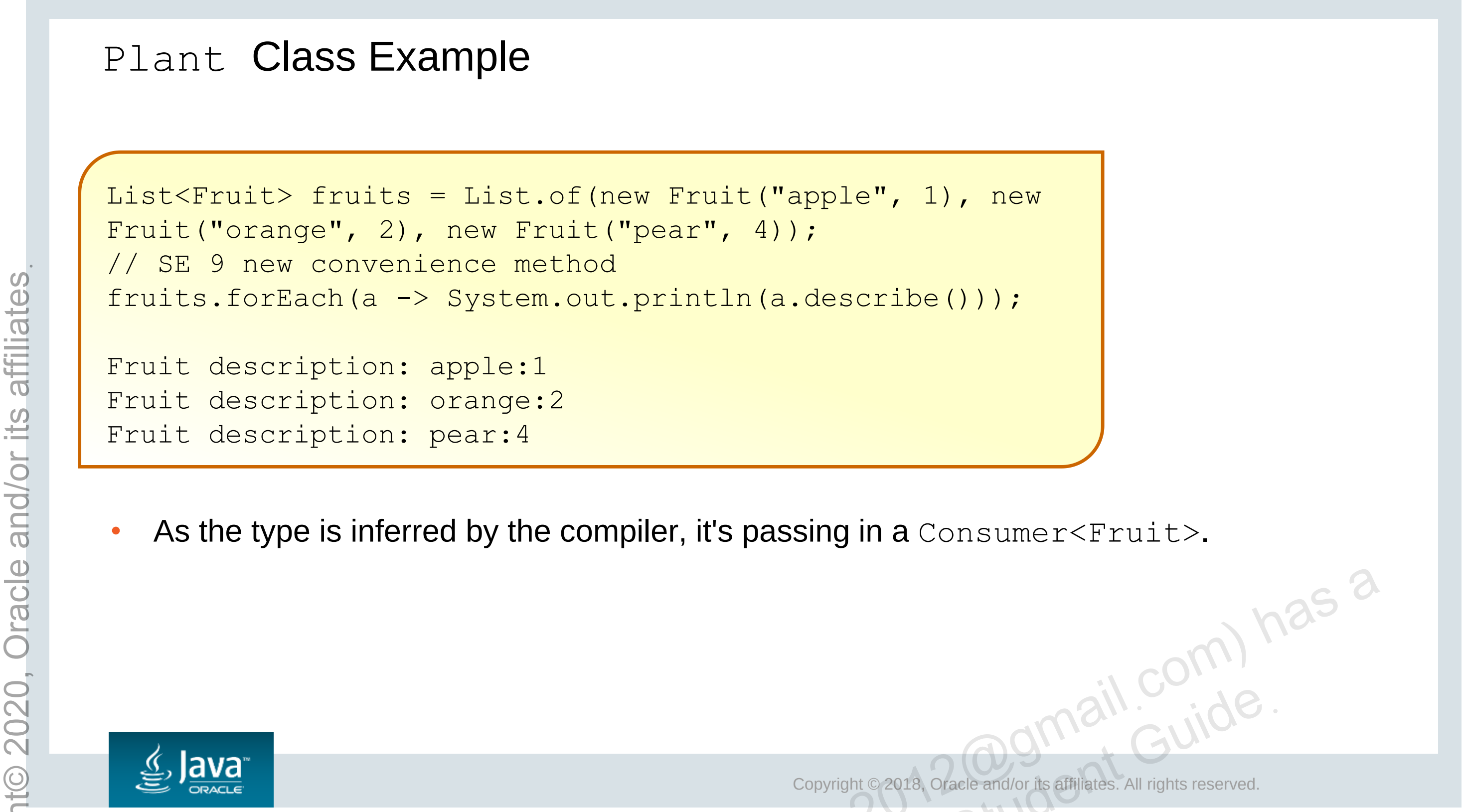
But in the case where lambda is inferred, i.e. the type is not declared, the type will be automatically that of the List or Stream.

But either by using an anon inner class OR by specifying the type explicitly in lambda, a supertype can be passed.



Example of Range of Valid Parameters

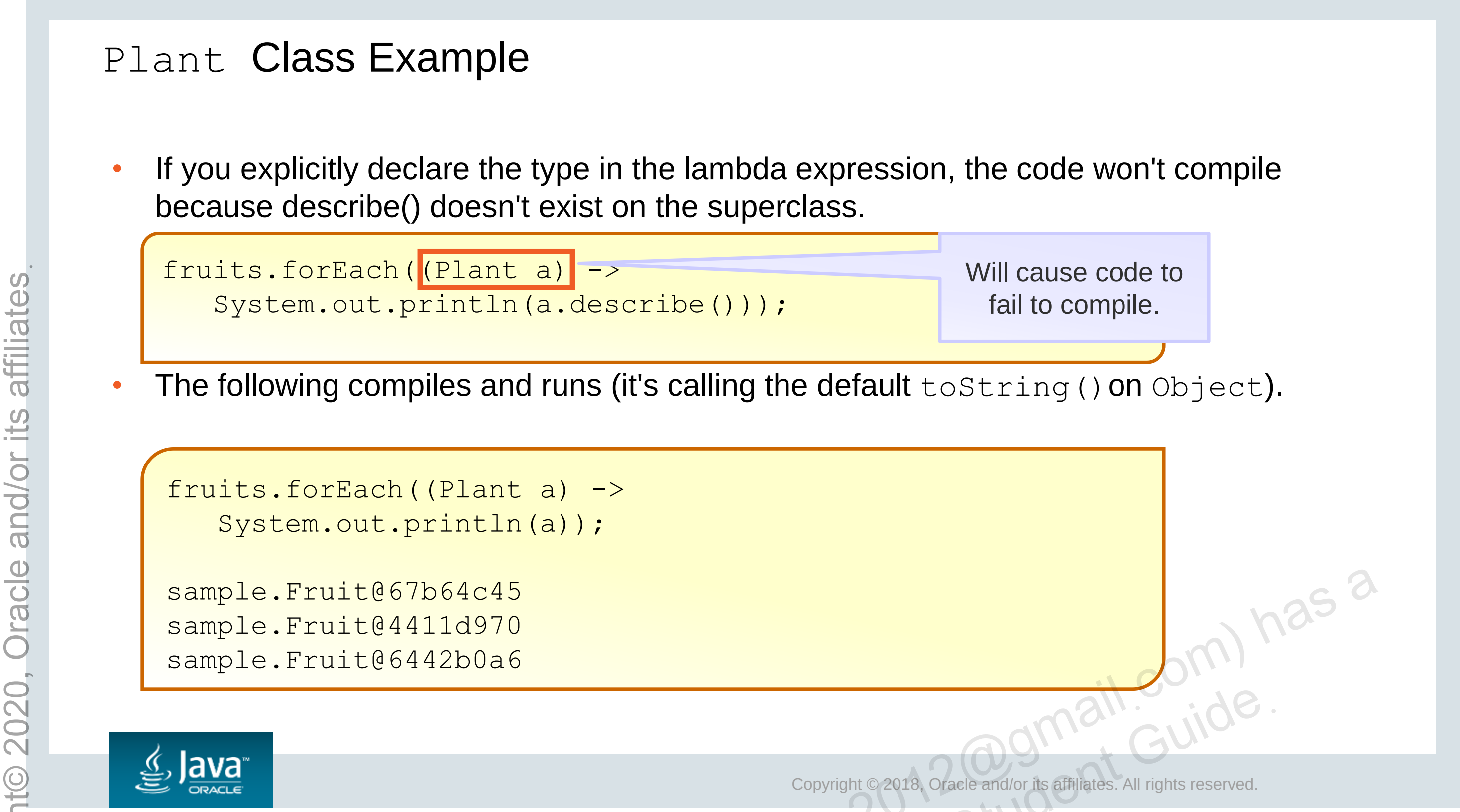
```
public class Fruit extends Plant{
    String fruitType = ""; int amount = 0;

    public Fruit(String type, int amount) {
        this.amount = amount;
        this.fruitType = type;
    }
    public String describe() {
        return "Fruit description: "
           + this.fruitType + ":" + this.amount;
    }
}

public class Plant {   }
```



`Plant` Class Example

```
List<Fruit> fruits = List.of(new Fruit("apple", 1), new
Fruit("orange", 2), new Fruit("pear", 4));
// SE 9 new convenience method
fruits.forEach(a -> System.out.println(a.describe()));

Fruit description: apple:1
Fruit description: orange:2
Fruit description: pear:4
```

- As the type is inferred by the compiler, it's passing in a `Consumer<Fruit>`.



Plant Class Example

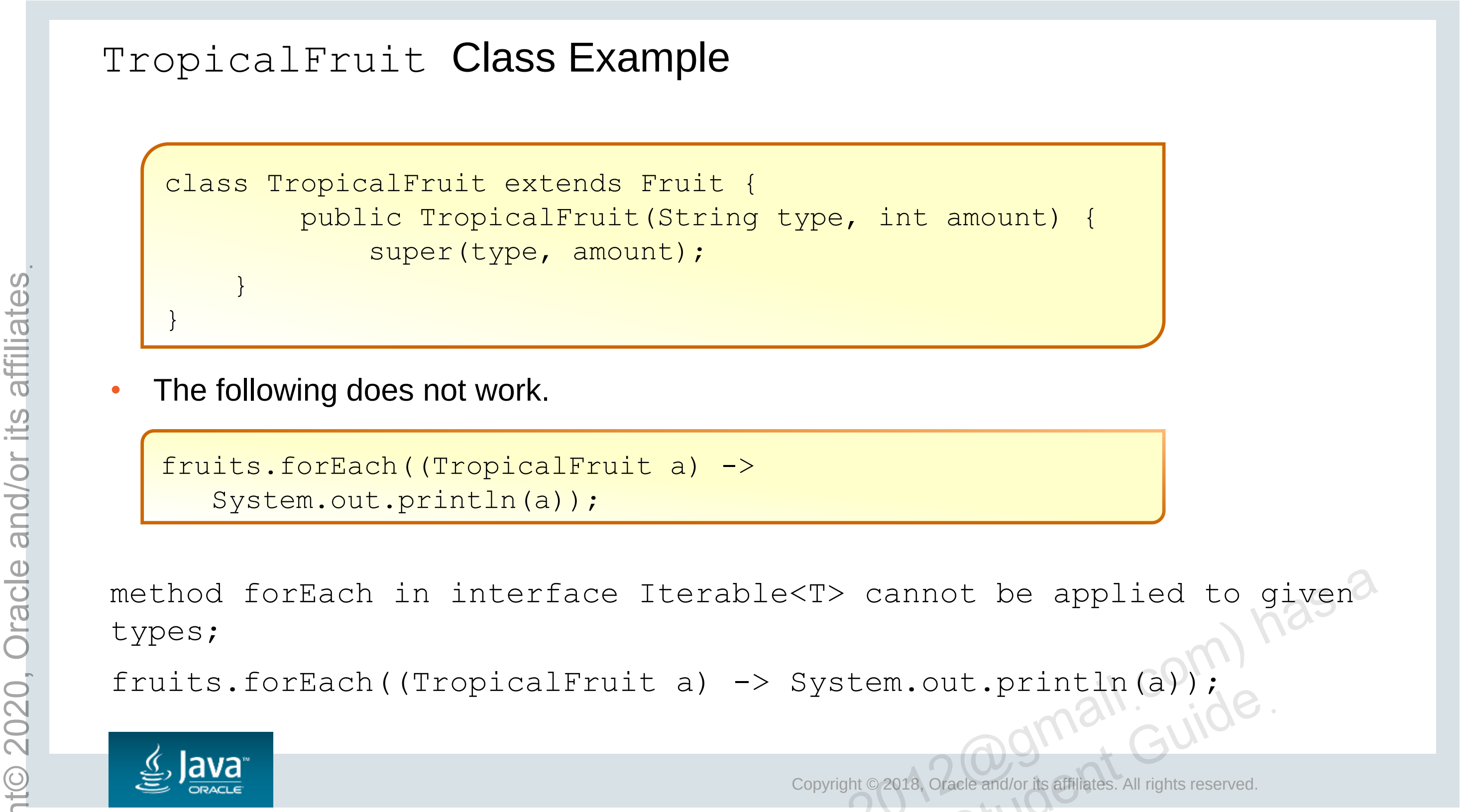
- If you explicitly declare the type in the lambda expression, the code won't compile because describe() doesn't exist on the superclass.

```
fruits.forEach((Plant a) ->
   System.out.println(a.describe()));
```

Will cause code to fail to compile.

- The following compiles and runs (it's calling the default `toString()` on `Object`).

```
fruits.forEach((Plant a) ->
   System.out.println(a));

sample.Fruit@67b64c45
sample.Fruit@4411d970
sample.Fruit@6442b0a6
```



TropicalFruit Class Example

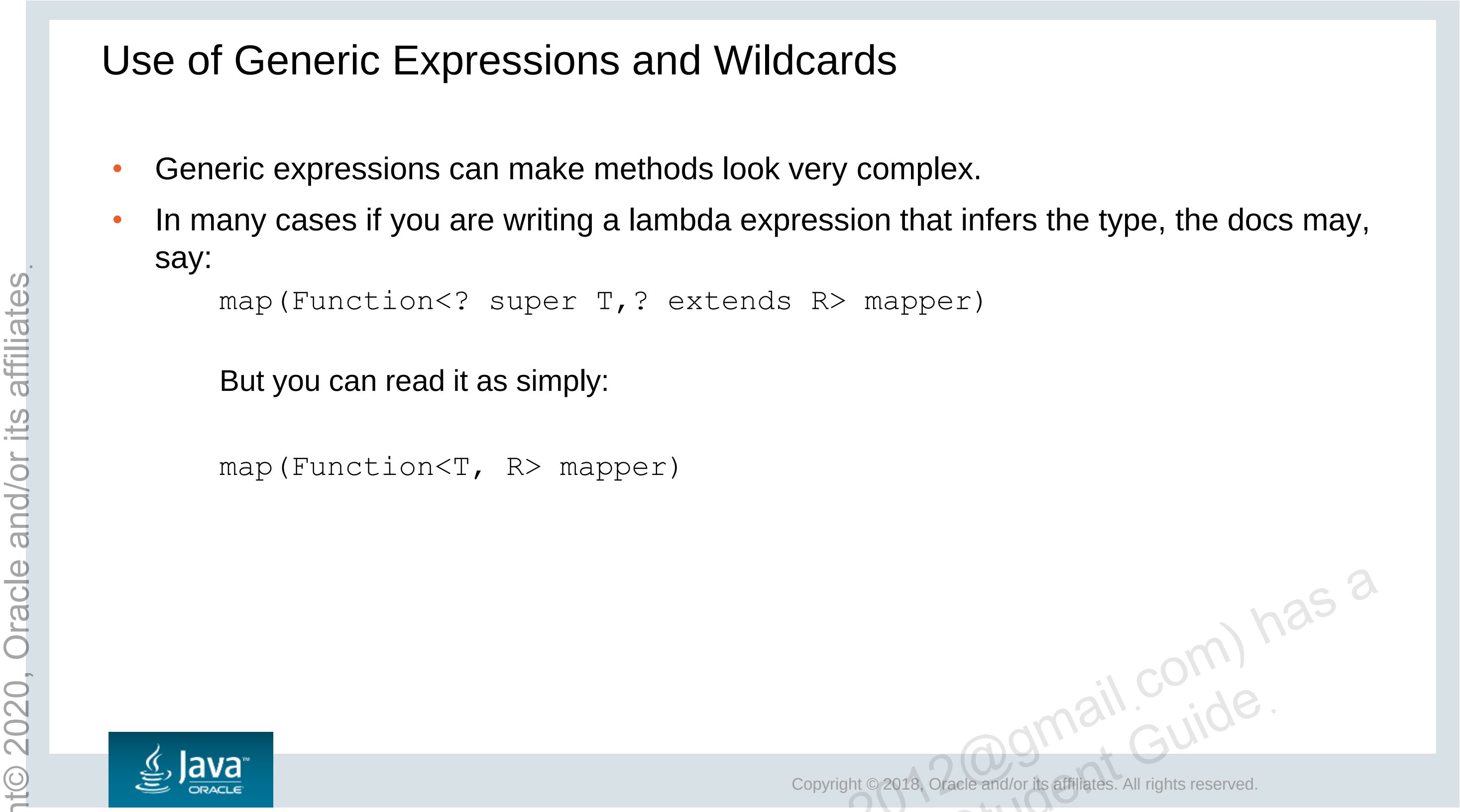
```
class TropicalFruit extends Fruit {
        public TropicalFruit(String type, int amount) {
            super(type, amount);
    }
}
```

- The following does not work.

```
fruits.forEach((TropicalFruit a) ->
   System.out.println(a));
```

```
method forEach in interface Iterable<T> cannot be applied to given
types;
fruits.forEach((TropicalFruit a) -> System.out.println(a));
```

```
method forEach in interface Iterable<T> cannot be applied to given types;
     fruits.forEach((TropicalFruit a) -> System.out.println(a));
                                     ^
  required: Consumer<? super Fruit>
  found: (TropicalF[...]ln(a)
  reason: argument mismatch; Consumer<TropicalFruit> cannot be converted to
Consumer<? super Fruit>
  where T is a type-variable:
    T extends Object declared in interface Iterable
```



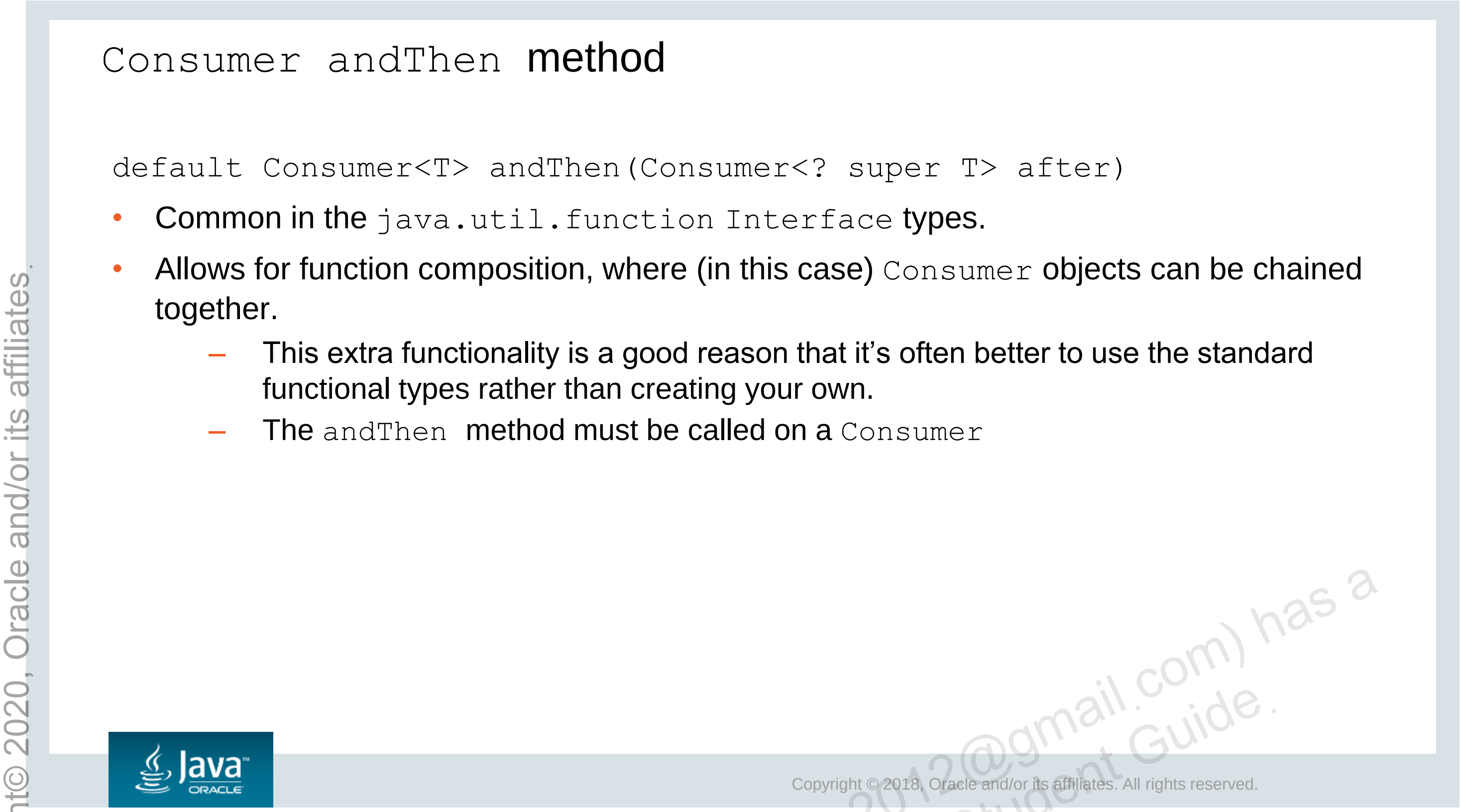
Use of Generic Expressions and Wildcards

- Generic expressions can make methods look very complex.
- In many cases if you are writing a lambda expression that infers the type, the docs may, say:

```
map(Function<? super T,? extends R> mapper)
```

But you can read it as simply:

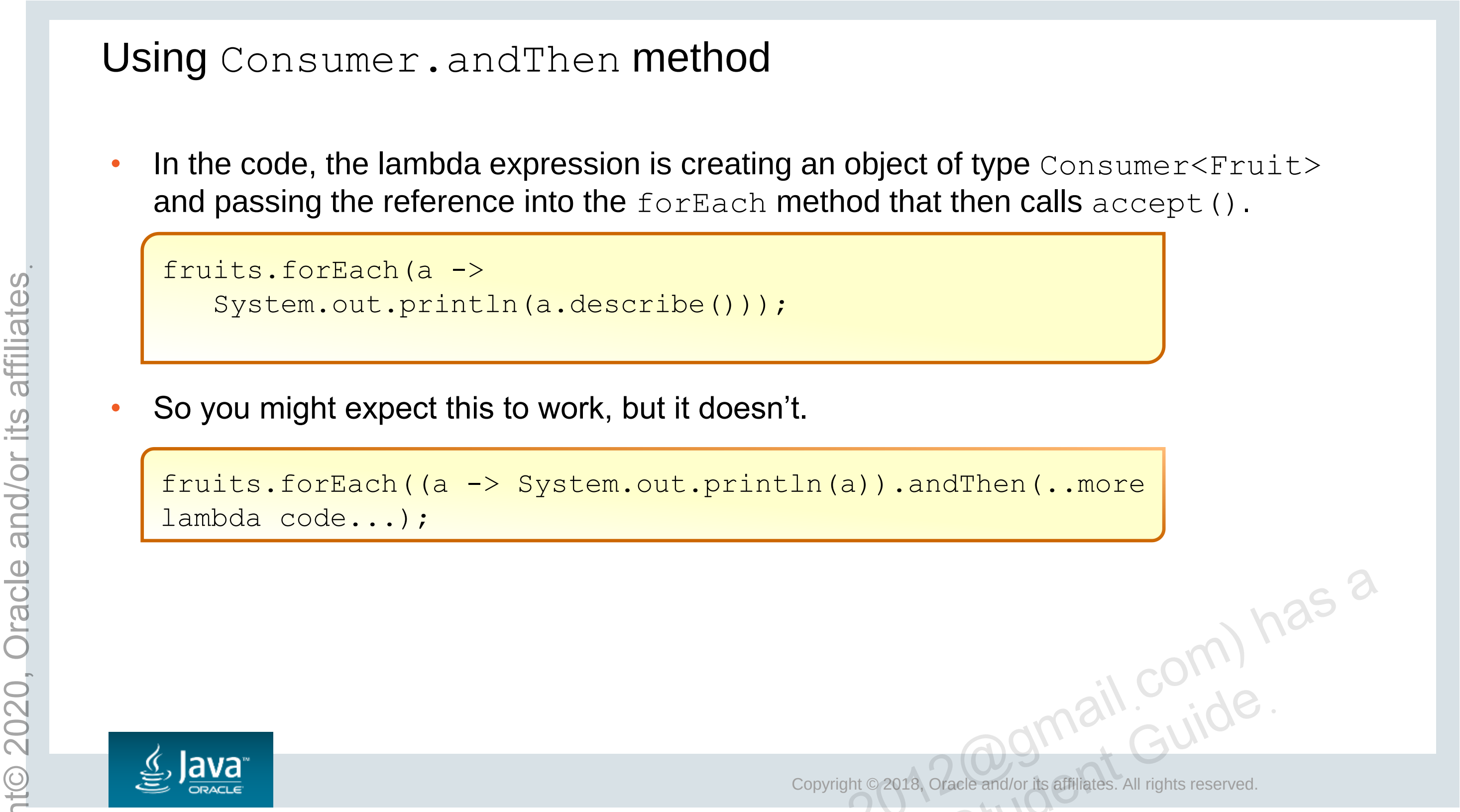
```
map(Function<T, R> mapper)
```



Consumer andThen method

```
default Consumer<T> andThen(Consumer<? super T> after)
```

- Common in the java.util.function Interface types.
- Allows for function composition, where (in this case) Consumer objects can be chained together.
  - This extra functionality is a good reason that it's often better to use the standard functional types rather than creating your own.
  - The andThen method must be called on a Consumer



Using `Consumer.andThen` method

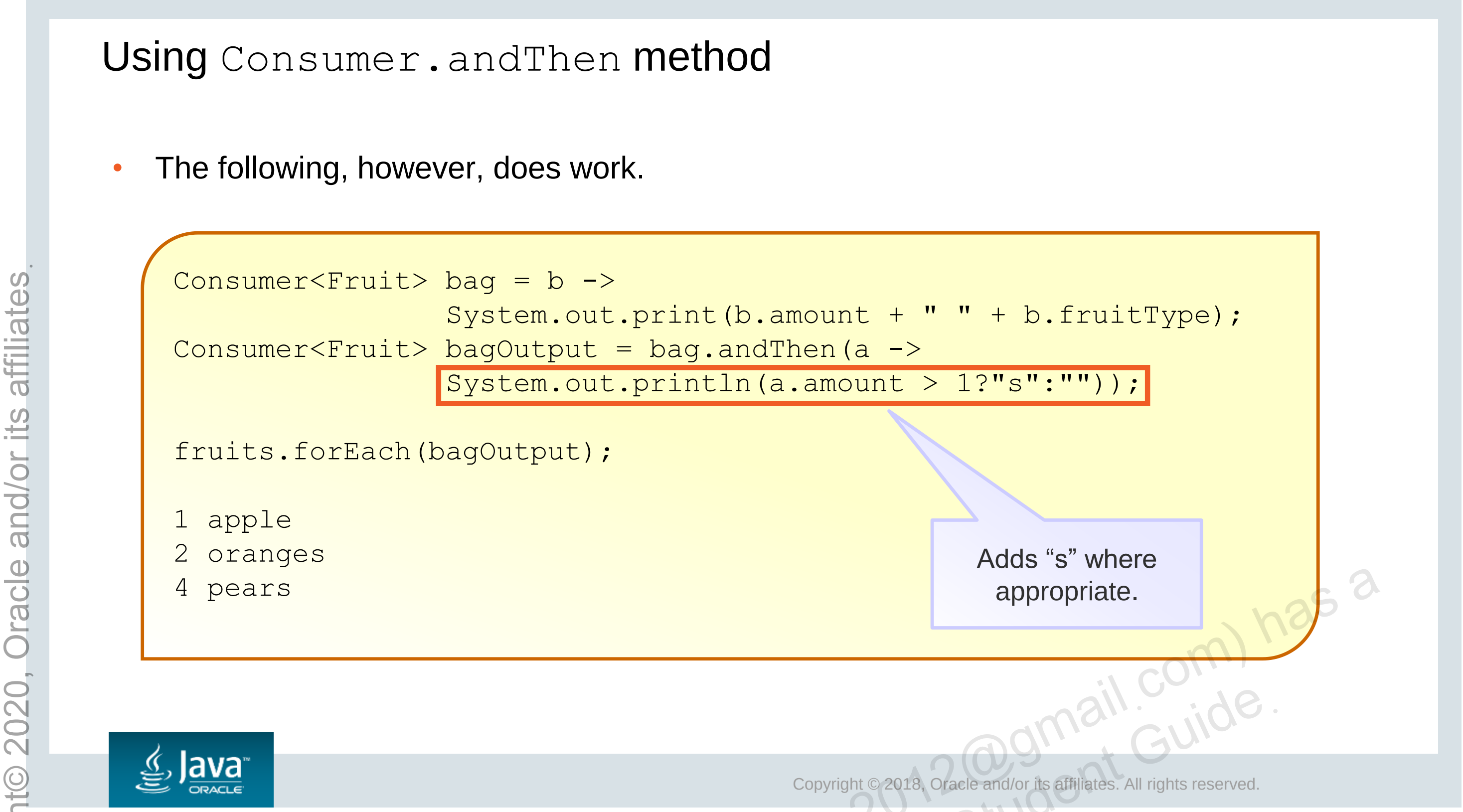
- In the code, the lambda expression is creating an object of type `Consumer<Fruit>` and passing the reference into the `forEach` method that then calls `accept()`.

```
fruits.forEach(a ->
   System.out.println(a.describe()));
```

- So you might expect this to work, but it doesn't.

```
fruits.forEach((a -> System.out.println(a)).andThen(..more
lambda code...);
```

The second example doesn't work because while a -> `System.out.println(a)` is the lambda expression that represents a `Consumer` when its method is called it returns `void` and can't therefore have the `andThen` method tacked on to the end. It won't compile, and the compiler will report a `void` referencing problem.



Using `Consumer.andThen` method

- The following, however, does work.

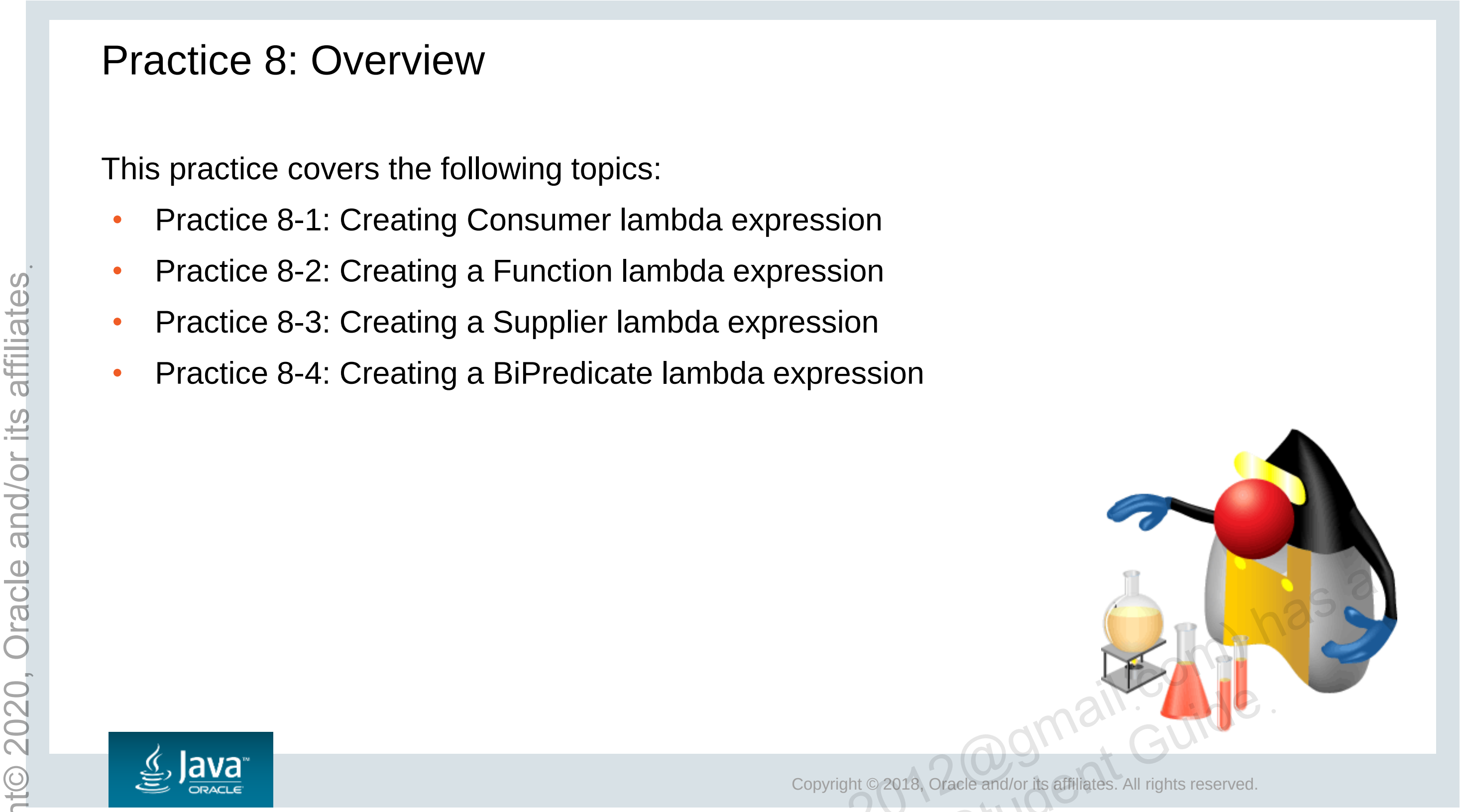
```
Consumer<Fruit> bag = b ->
                System.out.print(b.amount + " " + b.fruitType);
Consumer<Fruit> bagOutput = bag.andThen(a ->
                System.out.println(a.amount > 1?"s":""));

fruits.forEach(bagOutput);

1 apple
2 oranges
4 pears
```

Adds “s” where appropriate.

Summary

In this lesson, you should have learned how to:

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



Practice 8: Overview

This practice covers the following topics:

- Practice 8-1: Creating Consumer lambda expression
- Practice 8-2: Creating a Function lambda expression
- Practice 8-3: Creating a Supplier lambda expression
- Practice 8-4: Creating a BiPredicate lambda expression