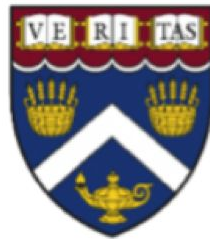


# CSCI E-88A Introduction to Functional and Stream Processing for Big Data Systems

Harvard University Extension, Spring 2020

Marina Popova, Edward Sumitra



Lecture 04 - Introduction To FP Concepts in Java

@Marina Popova, Edward Sumitra

# Agenda

- Baby steps into FP with Java
- Functional Interfaces
- Core Java Functional Interfaces



# OOP or FP ??

- It's not OR - it is AND
- History Lesson: <https://medium.com/javascript-scene/the-forgotten-history-of-oop-88d71b9b2d9f>
- <https://medium.com/javascript-scene/can-you-avoid-functional-programming-as-a-policy-7bd0570bcfb2>
- real OOP most important principles:
  - Encapsulation ==> internal state (and behavior) "hiding"
  - communication between objects via "public APIs" only (message passing)
  - dynamic or late binding ==> object/components can be changed at the runtime

**The essence of FP really boils down to:**

- Program with functions
- Avoid shared mutable state & side effects

Incidentally, according to Alan Kay, the instigator for all modern OOP, the essence of OOP is:

- Encapsulation
- Message Passing

So OOP is just another approach to **avoiding shared mutable state and side-effects.**

***"Clearly, the opposite of FP is not OOP  
The opposite of FP is unstructured, procedural programming"***

# In to the Functional World

Before we begin - let's remember that all of this is not Rocket Science! It may sound like it, but that's because of the Great "FP Terminology Barrier" as Alexander Alvin named it:

<https://alvinalexander.com/scala/fp-book/great-fp-terminology-barrier>

For example:  
what is a **Functor** ???

it is a name for a bunch of "things"  
that can be iterated and mapped  
over ... like this:

```
val xs = List(1, 2, 3).map(_ * 2)
```

## And the moral is ...

In this version of Earth's history, someone beat you to the invention of "things that can be mapped over," and for some reason — possibly because they had a mathematics background — they made this declaration:

*"Things that can be mapped over shall be called ...  
Functor."*

Huh?

History did not record whether the **Ballmer Peak**, caffeine, or other chemicals were involved in that decision.

In this book, when I use the phrase, "Functional Programming Terminology Barrier," this is the sort of thing I'm referring to. If a normal human being had discovered this technique, they might have come up with a name like `ThingsThatCanBeMappedOver`, but a mathematician discovered it and came up with the name, "Functor."

*Moral: A lot of FP terminology comes from mathematics.  
Don't let it get you down.*

# In to the Functional World

More of Terminology Barrier examples:

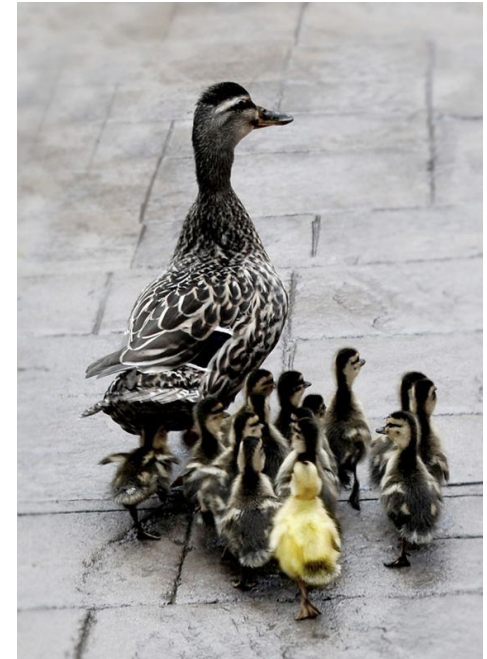
Term	Definition
combinator	Per the <a href="#">Haskell wiki</a> , this has two meanings, but the common meaning is, “a style of organizing libraries centered around the idea of combining things.” This refers to being able to combine functions together like a Unix command pipeline, i.e., <code>ps aux   grep root   wc -l</code> .
higher-order function	A function that takes other functions as parameters, or whose result is a function. ( <a href="https://docs.scala-lang.org">docs.scala-lang.org</a> )
lambda	Another word for “anonymous function.”

# Anonymous classes - refresher

- Anonymous classes are "one-off" classes that have no name (odd ducks :) )
- they are defined and created at the same time - "inline" in some other class using new() construct
- cannot be re-used anywhere else

The main use:

- create one-off/dynamic classes that are only needed in a specific context and do not "deserve" to have a separate class defined
- anonymous inner classes can either implement an interface or extend a class, but they can't do both at the same time
- they are not independent classes - they are either:
  - sub-classes of a concrete/abstract class OR
  - anonymous implementations of an interface



Ref: <http://cs-fundamentals.com/java-programming/java-anonymous-inner-classes.php>  
<https://www.geeksforgeeks.org/anonymous-inner-class-java/>

@Marina Popova, Edward Sumitra



# Anonymous classes - from Interfaces

body of the new class!

```
@Test
public void testDoTrick_anonymous_from_interface() {
    ActionResult result = AnimalManager.trainForTricks(
        new ITrainable() {
            public ActionResult doTrick(String trickName) {
                System.out.println("I always do tricks!");
                return ActionResult.SUCCESS;
            }
        },
        trickName);
    assertEquals(ActionResult.SUCCESS, result);
}
```

# FP: Functional Interfaces

## What are Functional Interfaces?

- **Any interface with exactly one abstract (not-implemented) method**
- it can have other static or default methods
- example existing Functional interfaces: Runnable, Iterable, ...
- true FI is pure: it is intended to be implemented by stateless classes
  - however, some FIs are more pure than others :)
  - no way to ensure that the implementing classes are truly stateless at compile time
  - the true pure FIs are in the `java.util.function` package



# Functional Interfaces: Example

```
package cscie88a.basics4;  
  
public interface ITrainable {  
    public ActionResult doTrick(String trickName);  
  
    default public ActionResult doTrickForTreat(String trickName, String treatName) {  
        System.out.println("I love the " + treatName +  
            " and will happily do the trick [" + trickName + "] !!");  
        return ActionResult.SUCCESS;  
    }  
}
```

*single abstract method*

*other (implemented) method:*

# Functional Interfaces

FIs are often annotated with the `@FunctionalInterface` annotations

- This is an optional annotation
- it verifies that the "single abstract method" requirement is met

## Side note on Annotations in Java

<https://www.geeksforgeeks.org/annotations-in-java/>

- start with `@`
- associate metadata with the annotated elements
- can give hints to compiler - change the compilation behavior

```
package cscie88a.fp1;

import cscie88a.basics4.ActionResult;

@FunctionalInterface
public interface ITrainableFP {

    public ActionResult doTrick(String trickName);

    default public ActionResult doTrickForTreat(String trickName, String treatName) {
        System.out.println("I love the " + treatName +
            " and will happily do the trick [" + trickName + "] !!");
        return ActionResult.SUCCESS;
    }
}
```

*we've added the annotation*

# Functional Interfaces

What if we try to add one more abstract method?

The compiler (and IDE) will yell at you!

```
package cscie88a.fp1;

import cscie88a.basics4.ActionResult;

@FunctionalInterface
public interface ITrainableFP {

    public ActionResult doTrick(String trickName);

    public ActionResult doOneMoreTrick(String trickName);

    default public ActionResult doTrickForTreat(String trickName, String treatName) {
        System.out.println("I love the " + treatName +
            " and will happily do the trick [" + trickName + "] !!");
        return ActionResult.SUCCESS;
    }
}
```

Multiple non-overriding abstract methods found in interface cscie88a.fp1.ITrainableFP

# Functional Interfaces and Lambdas

## Why are Functional Interfaces so important??

Because they can be implemented not only by regular and anonymous classes but also by Lambdas!

## What is a Lambda??

- a Lambda is a FUNCTION that can be created outside of any class
- can be passed around and used as a parameter or a returned value from other functions/methods
- can be executed on-demand! (important for Stream processing later on)

Ref: <http://tutorials.jenkov.com/java/lambda-expressions.html>

# Lambdas - examples

Let's see how we can write Lambdas - from the most verbose understandable) to the most concise syntax

```
package cscie88a.fp1;

import cscie88a.basics4.ActionResult;

public class AnimalManagerFP {

    // one-argument method
    public static ActionResult trainToRun(ITrainableFP animalToTrain){
        String trickName = "run";
        System.out.println("About to ask subject [" + animalToTrain.toString() + "] to run ...");
        return animalToTrain.doTrick(trickName);
    }
}
```

*our Functional Interface*

```
/**
 * the regular implementation way
 */
@Test
public void testDoRun_concrete(){
    CatFP sneaky = new CatFP("Sneaky");
    ActionResult result = AnimalManagerFP.trainToRun(sneaky);
    assertEquals(ActionResult.FAILURE, result);
}
```

```
/**
 * the anonymous class way
 */
@Test
public void testDoRun_anonymous(){
    ActionResult result = AnimalManagerFP.trainToRun(
        new ITrainableFP() {
            @Override
            public ActionResult doTrick(String trickName) {
                System.out.println("I hate running!");
                return ActionResult.FAILURE;
            }
        }
    );
    assertEquals(ActionResult.FAILURE, result);
}
```

# Lambdas - examples

```
/**
 * the first Lambda function - as an implementation of the ITr
 * as close to the class syntax as possible
 */
@Test
public void testDoRun_lambda1(){
    ActionResult result = AnimalManagerFP.trainToRun(
        (String trickName) -> {
            System.out.println("I hate running!");
            return ActionResult.FAILURE;
        }
    );
    assertEquals(ActionResult.FAILURE, result);
}
```

```
/**
 * the first Lambda function - as an implementation of the ITrainable interface;
 * simplifying syntax: omitting brackets and type for a single method argument
 */
@Test
public void testDoRun_lambda2(){
    ActionResult result = AnimalManagerFP.trainToRun(
        trickName -> {
            System.out.println("I hate running!");
            return ActionResult.FAILURE;
        }
    );
    assertEquals(ActionResult.FAILURE, result);
}
```

```
/**
 * the anonymous class way
 */
@Test
public void testDoRun_anonymous(){
    ActionResult result = AnimalManagerFP.trainToRun(
        new ITrainableFP() {
            @Override
            public ActionResult doTrick(String trickName) {
                System.out.println("I hate running!");
                return ActionResult.FAILURE;
            }
        }
    );
    assertEquals(ActionResult.FAILURE, result);
}
```

the most concise Lambda

```
@Test
public void testDoRun_lambda3(){
    ActionResult result = AnimalManagerFP.trainToRun(
        trickName -> ActionResult.FAILURE
    );
    assertEquals(ActionResult.FAILURE, result);
}
```

# Lambdas - examples

A few more examples of Lambda functions:

- no arguments, no return value
- no arguments, with return value
- multiple arguments

AnimalManagerFP class has methods using new interfaces

```
public static ActionResult doAnyTrick1(ITrainable1 animalToTrain) {
    animalToTrain.doAnyTrick();
    return ActionResult.SUCCESS;
}

public static ActionResult doAnyTrick2(ITrainable2 animalToTrain) {
    return animalToTrain.doAnyTrick();
}

public static ActionResult doManyTricks(ITrainable3 animalToTrain1) {
    return animalToTrain1.doManyTricks("jump", "roll");
}
```

```
package cscie88a.fp1;

/**
 * FI with no arguments and no return values
 */
@FunctionalInterface
public interface ITrainable1 {

    void doAnyTrick();
}
```

```
/**
 * Lambda function implementing ITrainable1 interface -
 * with no arguments and no return value;
 * no action in the body
 */
@Test
public void testDoAnyTrick1_doNothing(){
    ActionResult result = AnimalManagerFP.doAnyTrick1(
        () -> { }
    );
    assertEquals(ActionResult.SUCCESS, result);
}
```



# Lambdas - examples

```
/**
 * FI with a return value
 */
@FunctionalInterface
public interface ITrainable2 {

    ActionResult doAnyTrick();
}
```

```
/**
 * Lambda function implementing ITrainable2 interface -
 * with no arguments but with a return value
 */
@Test
public void testDoAnyTrick2(){
    ActionResult result = AnimalManagerFP.doAnyTrick2(
        () -> ActionResult.SUCCESS
    );
    assertEquals(ActionResult.SUCCESS, result);
}
```

```
/**
 * FI with multiple arguments
 */
@FunctionalInterface
public interface ITrainable3 {

    ActionResult doManyTricks(String trick1, String trick2);
}
```

```
/**
 * Lambda function implementing ITrainable3 interface -
 * with multiple arguments
 */
@Test
public void testDoAnyTrick3(){
    ActionResult result = AnimalManagerFP.doManyTricks(
        (trick1, trick2) -> ActionResult.SUCCESS
    );
    assertEquals(ActionResult.SUCCESS, result);
}
```

# Lambdas - Type Inference

How are Lambda functions matched with the corresponding FI?

This is what is called **Type Inference**

There are a few rules:

- Does the interface have only one abstract (unimplemented) method?
- Does the parameters of the lambda expression match the parameters of the single method?
- Does the return type of the lambda expression match the return type of the single method?



©Warren Photographic

*NOTE: Lambda parameter type inference does NOT use the lambda's body to determine the type, only the context*

# Lambdas - Type Inference - Examples

```
class TypeInferenceTest {  
  
    @Test  
    public void testTypesInference_withReturnType(){  
        ITrainableFP lambdaRef1 = trickName -> {  
            return ActionResult.FAILURE;  
        };  
        ActionResult result = AnimalManagerFP.trainToRun(lambdaRef1);  
  
        Object lambdaRef2 = trickName -> {  
            return ActionResult.FAILURE;  
        };  
        |  
        ActionResult result2 = AnimalManagerFP.trainToRun(lambdaRef2);  
    }  
  
    @Test  
    public void testTypesInference_noArgs(){  
        ITrainable1 lambdaRef1 = () -> {};  
  
        ITrainable1 lambdaRef4 = (trickName) -> {};  
  
        Object lambdaRef2 = () -> {};  
  
        Object lambdaRef3 = (ITrainable1) () -> {};  
    }  
}
```

Target type of a lambda conversion must be an interface

# Lambdas - Method references

Lambda functions can be written as method references - if this is all they need to do!

Types of Method references:

- reference to static methods
- reference to instance methods - no specific class instance - will not cover for now
- reference to a specific class instance method
- reference to a constructor - will not cover for now

more examples of method references: <https://dzone.com/articles/java-lambda-method-reference>

# Static Method References

```
@FunctionalInterface
public interface IAdoptable {

    public boolean readyForAdoption();
}
```

```
public class AdoptionService {

    public ActionResult tryToAdopt(IAdoptable someoneToAdopt) {
        if (someoneToAdopt.readyForAdoption()) {
            return ActionResult.SUCCESS;
        } else
            return ActionResult.FAILURE;
    }
}
```

```
public abstract class AbstractAnimalFP {

    private AnimalType animalType;
    private String name;
    private boolean hasCurrentShots;

    public AbstractAnimalFP(AnimalType animalType, String name) {...}

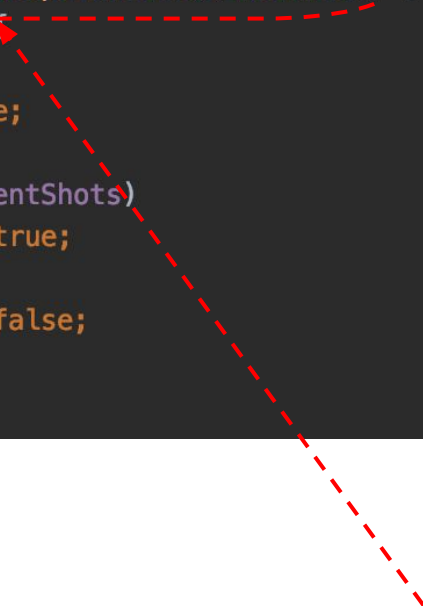
    // class level method - always returns TRUE
    public static boolean checkForAdoptionStatusStatic() {
        return true;
    };
}
```

```
@Test
public void testMethodReference_static(){
    AdoptionService service = new AdoptionService();
    IAdoptable lambdaMethodRef = AbstractAnimalFP::checkForAdoptionStatusStatic;
    ActionResult result = service.tryToAdopt( lambdaMethodRef );
    assertEquals(ActionResult.SUCCESS, result);
}
```

# Instance Method References

## AbstractAnimalFP.java

```
// Instance level method - can use instance state
public boolean checkForAdoptionStatusInstance(),{
    switch (animalType){
        case TIGER:
            return false;
        default:
            if (hasCurrentShots)
                return true;
            else
                return false;
    }
};
```

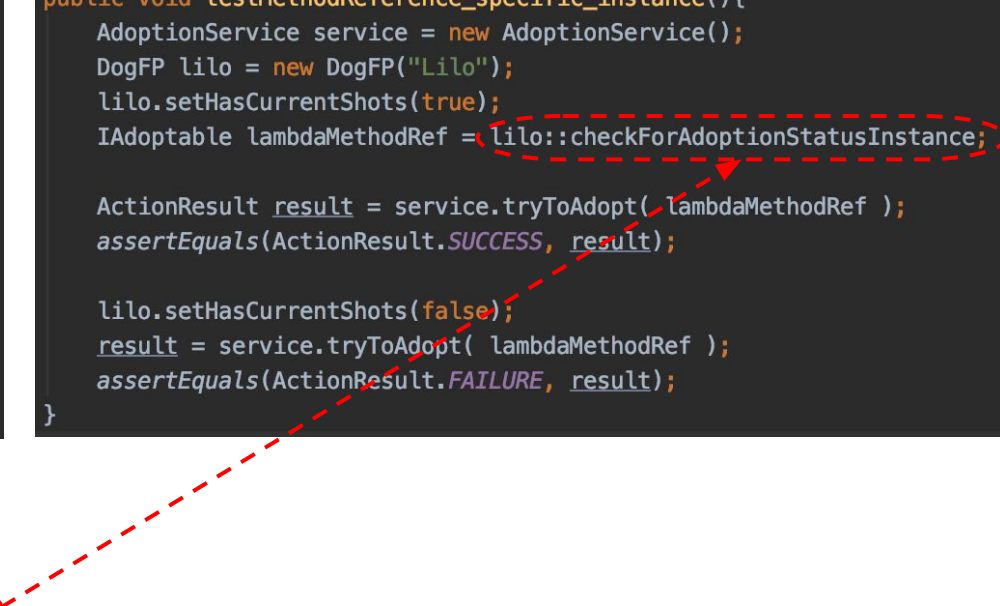


## AdoptionTests.java

```
@Test
public void testMethodReference_specific_instance(){
    AdoptionService service = new AdoptionService();
    DogFP lilo = new DogFP("Lilo");
    lilo.setHasCurrentShots(true);
    IAdoptable lambdaMethodRef = lilo::checkForAdoptionStatusInstance;

    ActionResult result = service.tryToAdopt( lambdaMethodRef );
    assertEquals(ActionResult.SUCCESS, result);

    lilo.setHasCurrentShots(false);
    result = service.tryToAdopt( lambdaMethodRef );
    assertEquals(ActionResult.FAILURE, result);
}
```



# Method References vs Lambda expressions?

**Whatever feel more readable to you!**

Great answer to this question by Brian Goetz:

<https://stackoverflow.com/questions/24487805/lambda-expression-vs-method-reference>

This question is very similar in spirit to "when should I use a named class vs an anonymous class"? And the answer is the same: *when you find it more readable*. There are certainly cases that are definitely one or definitely the other but there's a host of grey in the middle, and judgment must be used.

The theory behind method refs is simple: *names matter*. If a method has a name, then referring to it by name, rather than by an imperative bag of code that ultimately just turns around and invokes it, is often (but not always!) more clear and readable.



# Method References vs Method invocation ?

## Big difference!

*AbstractAnimalFP::checkForAdoptionStatusStatic*

vs

*AbstractAnimalFP.checkForAdoptionStatusStatic()*

Method referencing allows for a "lazy" execution of the method - which is very important in Streams

More Examples: <https://dzone.com/articles/a-little-lazy-lambda-tutorial>

# Method References and Lambda expressions

**There is much more to learn !**

... but we won't go there in this class:

- Lambdas - access to static vs instance variables
- anonymous classes vs Lambdas - how many instances are created with each??
  - hint: anonymous classes are implemented as a new instance , every time...
- method references to instance methods of ANY instance of a class!
- Lambdas with generics
- and much more ...

Research on your own!

Very interesting blog on the internals of Lambdas:

<https://medium.com/@edouard.kaiser/lambda-and-method-reference-133867e19c01>

And for the most inquisitive:

<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html>



10 Funny Pet Memes

# Functional Interfaces - Core Java

the Core FIs provided by Java are in the `java.util.function` package

There are a few groups of FIs with related behavior:

- general type FIs like `Function`, `Supplier`, `Consumer` , `Predicate`
- multi-parameter specialized FIs like `BiFunction`
- additional derived FIs like `UnaryOperator` and `BinaryOperator`
- specialized FIs for primitive parameter types like `IntFunction`, `DoubleConsumer`, ...

The most important FI to understand is **Function** !

# Function<T, R>

Function<T, R> is the simplest and most basic Functional interface in the java.util.function package: <https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html>

```
@FunctionalInterface
```

```
public interface Function<T,R>
```

```
Represents a function that accepts one argument and produces a result.
```

```
This is a functional interface whose functional method is apply(Object).
```

## Type Parameters:

```
T - the type of the input to the function
```

```
R - the type of the result of the function
```

Wait, what is this  
<T, R> stuff ???



We have not talked about Generics yet ....

[https://www.tutorialspoint.com/java/java\\_generics.htm](https://www.tutorialspoint.com/java/java_generics.htm)

*It is a way to allow methods and classes not to be "tied" to a specific type for arguments or variable or return types - but to be able to use different types for all of those.... more on this next .....*

# More on Generics

*“Java Generics are a language feature that allows for definition and use of generic types and methods.”*

Purpose? Type safety!

You declare methods and classes with Generic parameters:  
`List<E>`

And you provide specific types for instances of those classes and methods

```
List<String> myListOfStrings = new LinkedList<String>
```

```
java.util
```

## **Interface List<E>**

### **Type Parameters:**

E - the type of elements in this list

### **All Superinterfaces:**

`Collection<E>`, `Iterable<E>`

Modifier and Type	Method and Description
boolean	<b>add(E e)</b> Appends the specified element

Good ref: <https://howtodoinjava.com/java/generics/complete-java-generics-tutorial/>

# Back to Function<T, R>

```
@FunctionalInterface  
public interface Function<T,R>
```

Represents a function that accepts one argument and produces a result.

This is a functional interface whose functional method is `apply(Object)`.

## Type Parameters:

T - the type of the input to the function

R - the type of the result of the function

```
// can be implemented as any FI - via anonymous class  
Function<String, Integer> calcFnSafe = new Function<String, Integer>() {  
    @Override  
    public Integer apply(String s) {  
        if (s != null)  
            return s.length();  
        else  
            return 0;  
    }  
};
```

have to implement

Good Ref:

<https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/Get-the-most-from-Java-Function-interface-with-this-example>

# Function<T, R> - examples

```
public class CoreFIExamples {  
  
    public static void calculateLength(Function<String, Integer> calculatorFunction, String inputString) {  
        Integer resultLength = calculatorFunction.apply(inputString);  
        System.out.println("Calculated length for string: " + inputString + " --> " + resultLength);  
    }  
}
```

any implementation of the **Function<String, Integer>** can be passed as an argument to the *calculateLength()* method !

```
@Test  
public void testCalculations(){  
    // can be implemented as any FI - via anonymous class  
    Function<String, Integer> calcFnSafe = new Function<String, Integer>() {  
        @Override  
        public Integer apply(String s) {  
            if (s != null)  
                return s.length();  
            else  
                return 0;  
        }  
    };  
  
    Function<String, Integer> calcFn1 = inputString -> inputString.length();  
    Function<String, Integer> calcFn_plus10 = inputString -> inputString.length() + 10;  
    Function<String, Integer> calcFn_double = inputString -> inputString.length() * 2;  
  
    String inputString = "class";  
    CoreFIExamples.calculateLength(calcFn1, inputString);  
    CoreFIExamples.calculateLength(calcFn_plus10, inputString);  
    CoreFIExamples.calculateLength(calcFn_double, inputString);  
  
    // we could also provide an in-line lambda as implementation:  
    CoreFIExamples.calculateLength(  
        inputStringArg -> inputStringArg.length() + 100,  
        inputString);  
}
```



# Function<T, R> - examples

```
public static ActionResult useFunctionToTrain(Function<ITrainableFP, ActionResult> fn, ITrainableFP trainingSubject) {  
    ActionResult result = fn.apply(trainingSubject);  
    return result;  
}
```

you can use other Functional Interfaces as parameter types to the **Function<T, R>** as well !

```
// implementation of the Function<> interface - as an anonymous class  
// and use external state for trick name  
String currentTrickName = "jump";  
Function<ITrainableFP, ActionResult> myTrainingFunction1 = new Function<ITrainableFP, ActionResult>() {  
    @Override  
    public ActionResult apply(ITrainableFP trainingSubject) {  
        return trainingSubject.doTrick(currentTrickName);  
    }  
};  
  
// implementation of the Function<> interface - as a lambda function  
Function<ITrainableFP, ActionResult> myTrainingFunction2 = trainingSubject -> {  
    return trainingSubject.doTrick(trickName: "some trick");  
};  
  
// implementation of the Function<> interface - as a yet another lambda function  
// always returns FAILURE  
Function<ITrainableFP, ActionResult> myTrainingFunction3 = trainingSubject -> {  
    return ActionResult.FAILURE;  
};
```

```
// service method  
ActionResult result1 = CoreFIEExamples.useFunctionToTrain(myTrainingFunction1, sneaky);  
// Sneaky is in a good mood - training should succeed  
assertEquals(ActionResult.SUCCESS, result1);  
ActionResult result2 = CoreFIEExamples.useFunctionToTrain(myTrainingFunction2, sneaky);  
assertEquals(ActionResult.SUCCESS, result2);  
ActionResult result3 = CoreFIEExamples.useFunctionToTrain(myTrainingFunction3, sneaky);  
assertEquals(ActionResult.FAILURE, result3);
```

# Function<T, R> - continue

Summary: Function<> FI is a very general interface - it defines a template for a behavior that takes an object of one type and returns an object of some other (or the same!) type

The most common use of Function FI is in map-like methods

There are many specialized versions of the Function<T, R> interface:

- for specific types input and output parameters
  - IntFunction, LongFunction, ToIntFunction, ...
- for no input or no output values or specialized cases:
  - Supplier, Consumer, Predicate ....

```
java.util.concurrent.atomic
java.util.concurrent.locks
java.util.function
java.util.jar
java.util.logging
java.util.prefs
```

## java.util.function

### Interfaces

```
BiConsumer
BiFunction
BinaryOperator
BiPredicate
BooleanSupplier
Consumer
DoubleBinaryOperator
DoubleConsumer
DoubleFunction
DoublePredicate
DoubleSupplier
DoubleToIntFunction
DoubleToLongFunction
DoubleUnaryOperator
Function
IntBinaryOperator
IntConsumer
IntFunction
IntPredicate
IntSupplier
IntToDoubleFunction
IntToLongFunction
IntUnaryOperator
LongBinaryOperator
LongConsumer
LongFunction
LongPredicate
LongSupplier
LongToDoubleFunction
LongToIntFunction
LongUnaryOperator
ObjDoubleConsumer
ObjIntConsumer
ObjLongConsumer
Predicate
Supplier
ToDoubleBiFunction
ToDoubleFunction
ToIntBiFunction
ToIntFunction
ToLongBiFunction
ToLongFunction
```

# Supplier<T>

- Supplier interface is used for generation/producing of objects of a specific type
- It takes no parameters in and produces a result - object of that type
- It's very simple but very widely used in many other core interfaces, classes and libraries
- very important for streams!

## Interface Supplier<T>

### Type Parameters:

T - the type of results supplied by this supplier

```
@FunctionalInterface  
public interface Supplier<T>
```

Represents a supplier of results.

There is no requirement that a new or distinct result be returned each time the supplier is invoked.

This is a functional interface whose functional method is `get()`.

# Supplier<T> - anonymous class example

CoreExamplesTest.java

```
@Test
public void testSupplierAndConsumer_anonymousImpl(){
    Supplier<AbstractAnimalFP> catSupplier = new Supplier<AbstractAnimalFP>() {
        @Override
        public AbstractAnimalFP get(){
            CatFP newCat = new CatFP("SuppliedCat_Abstract");
            System.out.println("I'm supplying a new cat: " + newCat);
            return newCat;
        }
    };
};
```

**this is the key!**

you could also provide a Lambda or some other concrete implementation of the Supplier interface

# Supplier<T> - Lambda example

CoreExamplesTest.java

```
Random random = new Random();  
Supplier<AbstractAnimalFP> catSupplier = () -> {  
    boolean hasCurrentShots = random.nextBoolean();  
    CatFP newCat = new CatFP("SuppliedCat");  
    newCat.setHasCurrentShots(hasCurrentShots);  
    return newCat;  
};
```

# Consumer<T>

Consumer interface is used for "consuming" objects of a specific type  
It takes one parameter in - and produces nothing as a result  
It's very simple but also very widely used

## Interface Consumer<T>

### Type Parameters:

T - the type of the input to the operation

### All Known Subinterfaces:

`Stream.Builder<T>`

From Java docs:

Represents an operation that accepts a single input argument and returns no result. *Unlike most other functional interfaces, Consumer is expected to operate via side-effects.*

This is a functional interface whose functional method is ***accept(Object)***.

# Consumer<T> - anonymous impl example

CoreExamplesTest.java

```
Consumer<AbstractAnimalFP> animalConsumer = new Consumer<AbstractAnimalFP>() {  
    @Override  
    public void accept(AbstractAnimalFP abstractAnimalFP) {  
        System.out.println("I'm accepting an animal: " + abstractAnimalFP);  
    }  
};
```

this is the key!

you could also provide a Lambda or some other concrete implementation of the Consumer interface



# Consumer<T> - Lambda example

CoreExamplesTest.java

```
Consumer<AbstractAnimalFP> animalConsumer = animal -> {  
    if (animal.isHasCurrentShots()) {  
        System.out.println("We can accept this animal");  
    } else {  
        System.out.println("We can NOT accept this animal");  
    }  
};
```

# Consumer<T> and Supplier<T> full example

CoreExamplesTest.java

```
// example of implementing Supplier/Consumer FIs using anonymous classes
@Test
public void testSupplierAndConsumer_anonymousImpl(){
    Supplier<AbstractAnimalFP> catSupplier = new Supplier<AbstractAnimalFP>() {
        @Override
        public AbstractAnimalFP get() {
            CatFP newCat = new CatFP("SuppliedCat_Abstract");
            System.out.println("I'm supplying a new cat: " + newCat);
            return newCat;
        }
    };

    Consumer<AbstractAnimalFP> animalConsumer = new Consumer<AbstractAnimalFP>() {
        @Override
        public void accept(AbstractAnimalFP abstractAnimalFP) {
            System.out.println("I'm accepting an animal: " + abstractAnimalFP);
        }
    };

    CoreFIEExamples.chainSupplierAndConsumer(catSupplier, animalConsumer);
}
```

CoreFIEExamples.java

```
/* a more verbose but easier to inspect/debug implementation
 */
public static void chainSupplierAndConsumer(Supplier<AbstractAnimalFP> animalSupplier,
                                           Consumer<AbstractAnimalFP> animalConsumer) {

    int numberOfRuns = 5;
    for (int i=0; i<numberOfRuns; i++){
        System.out.println("Run #" + i);
        AbstractAnimalFP animal = animalSupplier.get();
        System.out.println("supplied animal: " + animal);
        animalConsumer.accept(animal);
    }
}
```

# Lambdas - looking ahead

## Why are Lambdas so important?

They are used as the core concept and building block in Streams APIs -  
to implement parallel stream processing!

as we will see in the next lecture ...

