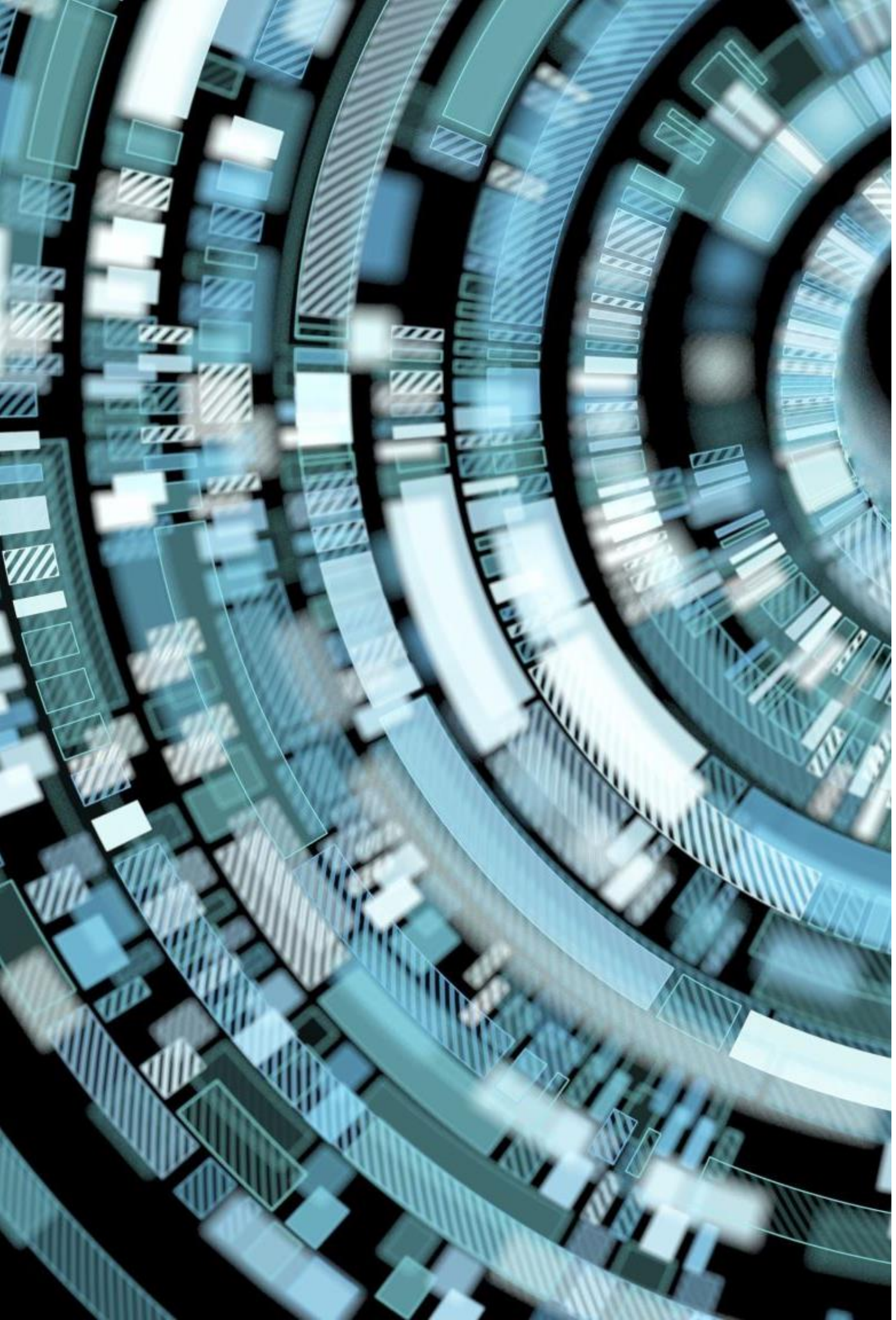
Computer System Design & Application

计算机系统设计与应用A

陶伊达 （TAO Yida）
taoyd@sustech.edu.cn

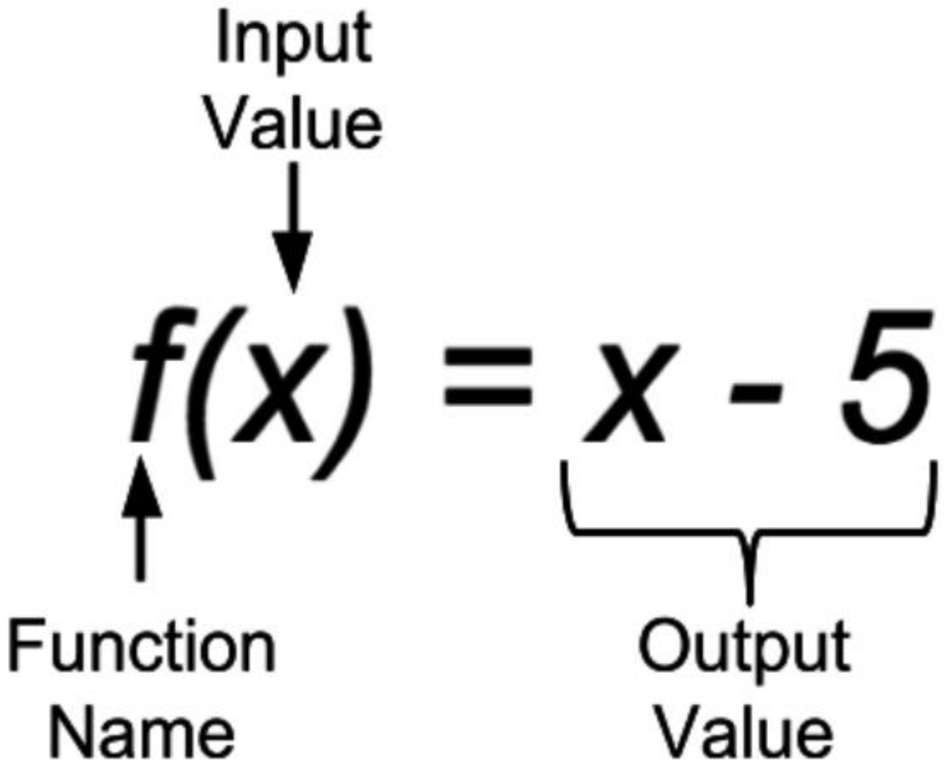


Lecture 3

- Functional Programming
- Lambda Expressions

What is a function?

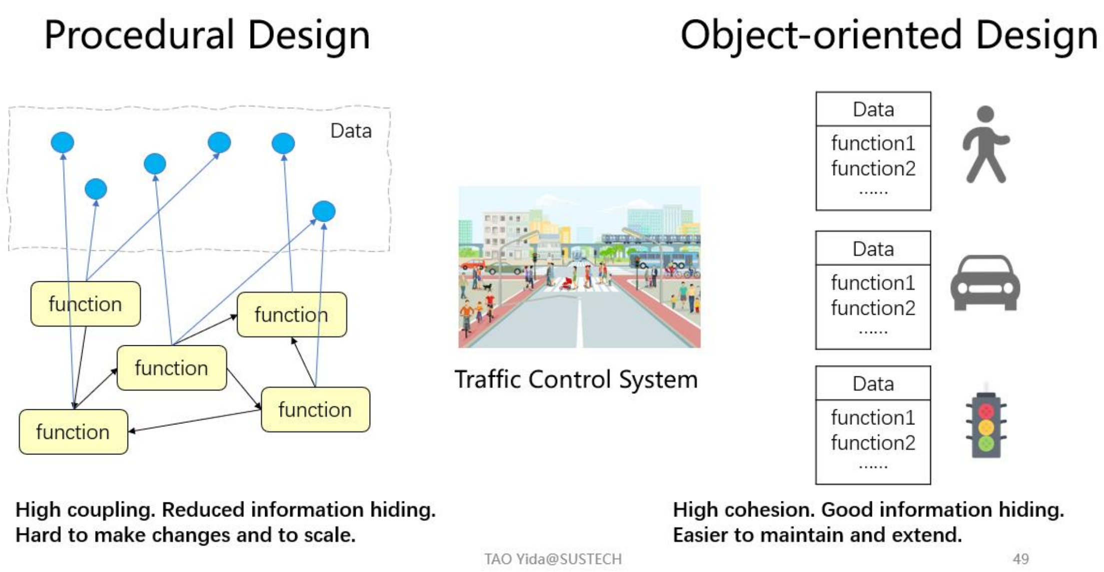
- [Mathematics] A function from a set X to a set Y assigns to each element of X exactly one element of Y
- Maps input to output



x	f(x)
1	-4
2	-3
3	-2
4	-1

What is Functional Programming?

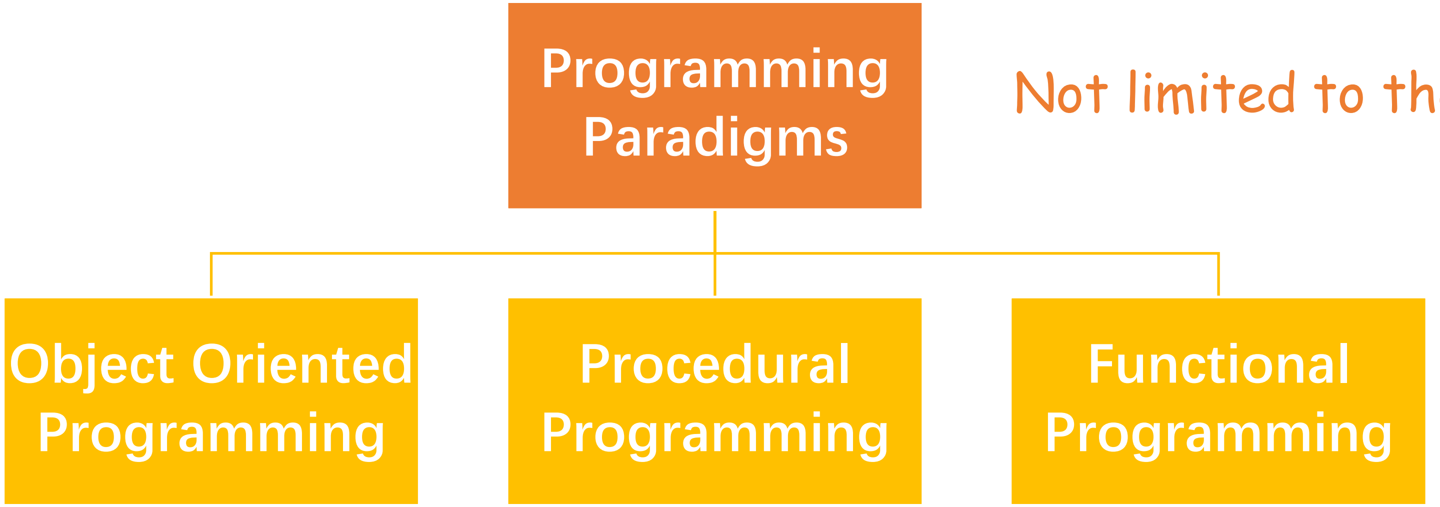
What types of programming paradigms have we seen so far?



- Functional programming is a **programming paradigm** (编程范式)
- A programming paradigm refers to the way of thinking about things and the way of solving problems

编程范式好比武功门派，博大精深且自成体系。
--摘自《冒号课堂：编程范式与OOP思想》

Programming Paradigms



Not limited to these 3!

Organize data and logics in **objects** (fields and methods)

Organize code as sequential **procedures** (may change same data)

Organize code as **functions** (behaviors are separated from the data, which won't be changed)

Functional Programming (函数式编程)

- Basic unit of computation is function
- Primary characteristics
  - Functions are treated as first-class citizens
  - No side effects
  - Immutability
  - Recursion

First-class functions

Functions are treated like any other variables

```
function me() {
  return '👱‍♂️';
}

greet(me);
```

Functions can be passed as arguments

```
const greet = function () {
    console.log('👋');
}

// The greet variable is now a f
greet();
```

Functions can be assigned to a variable

```
// #3 Return as values from other functions
function Promises() {
  return new Promise((resolve, reject) ⇒ {
    resolve('🈳');
  });
}
```

Functions can be returned

Image source : https://www.webtips.dev/webtips/javascript-interview/first-class-functions

Higher Order Functions

- A higher order function
  - Takes another function as argument
  - Returns another function as result

- A common usage scenario: data processing
  - `map()`, `filtering()`, `reduce()`, etc.
  - More on these later

No Side Effects

- Side effects: Events that are caused by a system with a **limited scope**, whose effects are felt **outside of that scope**
- Pure functions have no side effects (cannot change external states)

Impure function	**Side Effects**
writeFile(filename)	External files are changed
updateDatabase(table)	External database table is changed
sendAjaxRequest(request)	External server state is changed

No Side Effects

- Pure functions **always** produce the same output for the same input (regardless of the history)

Impure function	Input	Possible Output
writeFile(filename)	filename	Success or failures given file state
queryDatabase(table)	table	Different results given table state
sendAjaxRequest(request)	request	Different responses given server state

No Side Effects

- Pure functions **always** produce the same output for the same input (regardless of the history)

Is this a pure function?

```
global_list = []
def append_to_list(x):
    global_list.append(x)
    print(global_list)
```

Side effects

- `global_list` is implicitly changed even though it is not declared as the input to the function
- The output of the function changed even though the input remains the same

Immutability

- Variables, once defined, never change their values (eliminate side effects)
- Pure functional programs do not have assignment statements

```
x = x + 10
```

Non-functional style

```
int plusTen(int x)
{
        return x+10;
}
```

Functional style

What about loops?

- No “while” or “for” loop in functional programming
- How do we perform iterations though?

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

```
def factorial(n):
    fact = 1
    while n >= 1:
        fact = fact * n
        n = n - 1
    return fact
```

Using loops

```
def factorial(n):
    if n <= 0: return 1
    return n * factorial(n-1)
```

Using recursive functions (which invoke themselves)
Or higher order functions (e.g., map)

Advantages of Functional Programming

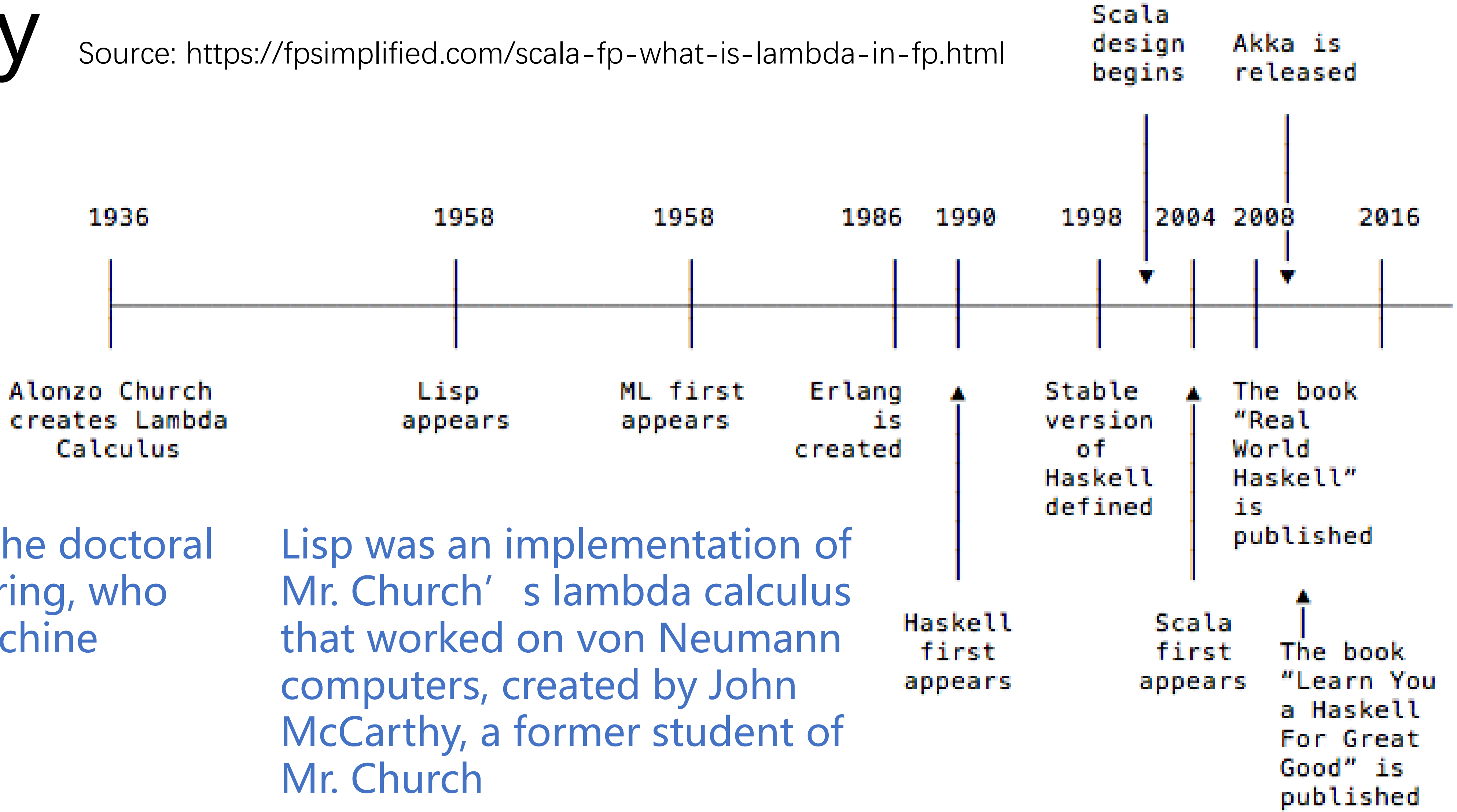
- Easy to debug, test, and parallelize
  - Same input => same output (deterministic)
  - No side effects
  - Data are immutable
- Complexity is dramatically reduced (architectural simplicity)
  - The only interaction with the external system is via the argument and return value (API)

OO style: Object methods interact with the object states

Procedural style: external state is often manipulated from within the function

History

Source: https://fpsimplified.com/scala-fp-what-is-lambda-in-fp.html



Alonzo Church is the doctoral advisor of Alan Turing, who created Turing Machine

Lisp was an implementation of Mr. Church's lambda calculus that worked on von Neumann computers, created by John McCarthy, a former student of Mr. Church

Functional Programming Languages

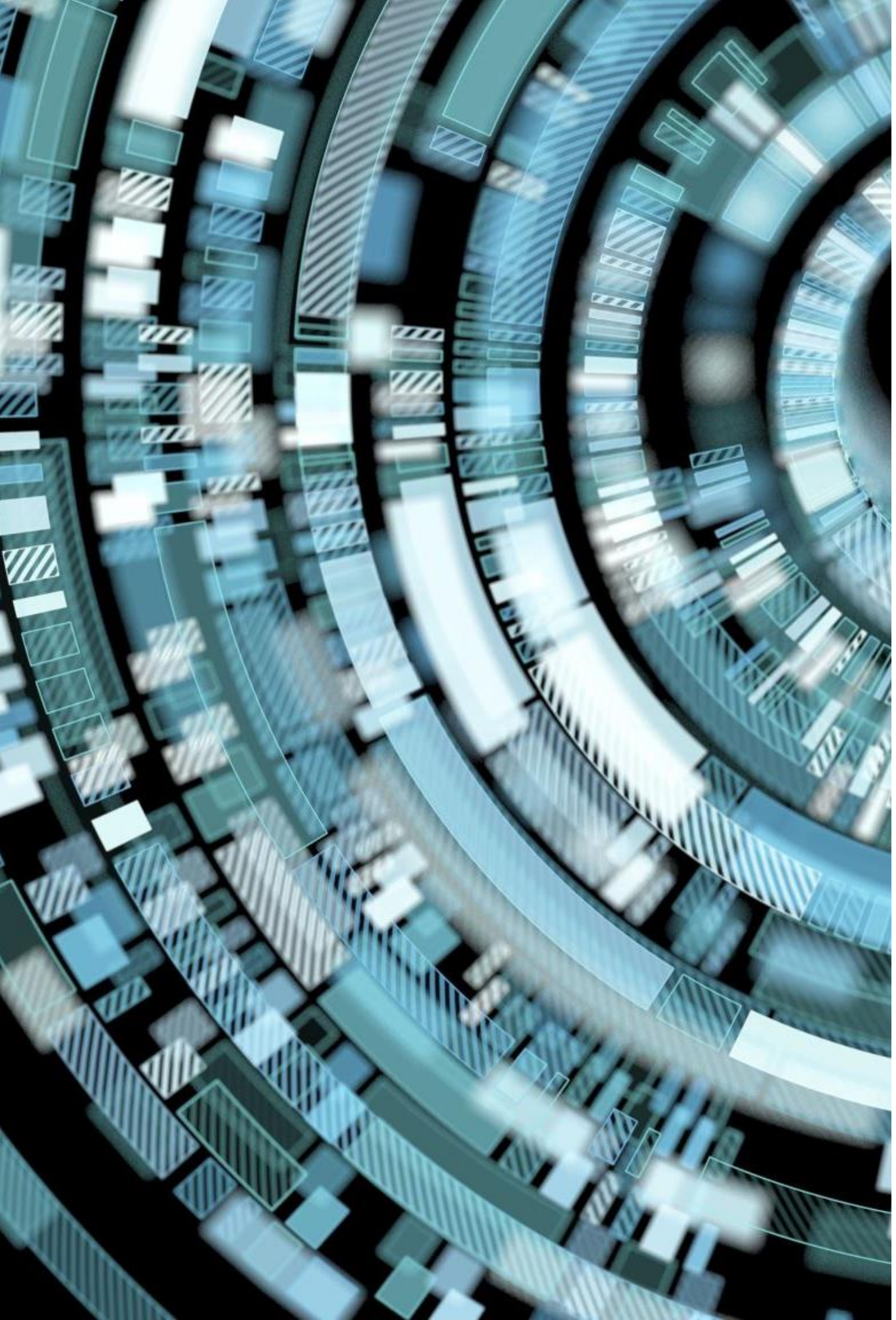
- Lisp, Erlang, Haskell, Clojure, Scala, F#, Python, Javascript, Kotlin, Rust, Swift, etc.

- FP is used in big companies
  - **Whatsapp** needs only 50 engineers for its 900M users, using Erlang
  - **Huawei** adopts Rust to engineer trustworthy software systems
  - **NVIDIA** uses Haskell for the backend development of its GPUs
  - **Facebook** uses Haskell to fight spams
  - ……

Functional Programming in Java

- Different programming paradigms are not necessarily mutually exclusive
  - Python also supports OOP
  - Java also supports the functional styles of programming

- Java 8 introduces functional programming abilities
  - Lambda expressions
  - Streams API



Lecture 3

- Functional Programming
- Lambda Expressions
  - Syntax
  - Type inference
  - Use cases
  - Method references
  - Java Functional Interfaces

Java Lambda Expressions

- Introduced in Java 8
  - Java’s first step into functional programming
- A Java lambda expression
  - is an anonymous function with no name/identifier
  - can be created without belonging to any class
  - can be passed as a parameter to another function
  - are callable anywhere in the program

Lambda Syntax

Arrow token

```
(param1, param2) -> {lambda expressions}
```

Left part – lambda parameters

- No function name
- Parentheses could be omitted for a single parameter
- Multiple parameters are separated by comma (,)
- `()` is used if no parameter is needed

Right part – lambda expressions

- Curly braces could be omitted for a single expression
- Multiple expressions are separated by a semi-colon (;)
- Can have a `return` statement
- Local assignments and control structures (`if`, `for`) are allowed (but probably less common).

Lambda Usage

- Lambda: a shortcut to define an implementation of a functional interface

- Functional interface is an interface with a single abstract method (e.g., `Comparator<T>` interface only has one abstract method `int compare(T o1, T o2)`)

Example: sorting a string list by element's length

```java
public class StringComparator implements Comparator<String>{
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
}
Collections.sort(strList, new StringComparator());
```

Classic way

- Explicitly creating a class that implements the Comparator interface
- Verbose

```java
Collections.sort(strList, new Comparator<String>() {
        public int compare(String s1, String s2) {
            return Integer.compare(s1.length(), s2.length());
        }
});
```

Using the anonymous class

- Don't have to declare a name for it
- Declare and instantiate the class at the same time
- Anonymous class can be used only once
- Shorter code, but still verbose

Using lambda in Java 8

```java
Collections.sort(strList, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

Matching Lambdas to Functional Interfaces

```
Collections.sort(strList, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

```
Collections.sort(List<T> list,          Comparator<? super T> c )
```

1. Lambda is matched to the Comparator interface

```
int compare(T o1, T o2)
```

2. Lambda is matched to `Comparator.compare(T o1, T o2)` method, which is the only abstract(unimplemented) method in the `Comparator` interface

3. The parameter and return type are deduced by compiler using type inference

Matching Lambdas to Functional Interfaces

1. Matching lambda to interface
   - By comparing caller method's parameter type

2. Matching lambda to interface's abstract method
   - The interface must have only one abstract (unimplemented) method

3. Matching method parameters and return type
   - Parameter types and return type must match

All of these must be satisfied to replace a functional interface with lambda successfully!

Matching lambda to interface's abstract method

- From Java 8, an interface could have both default methods and static methods (method with implementations/concrete methods)
- A functional interface must have only one abstract (unimplemented) method
  - But it can have multiple default and static methods

```
public interface MyInterface {

    void printIt(String text);

    default public void printUtf8To(String text, OutputStream outputStream){
        try {
            outputStream.write(text.getBytes("UTF-8"));
        } catch (IOException e) {
            throw new RuntimeException("Error writing String as UTF-8 to OutputStream", e);
        }
    }

    static void printItToSystemOut(String text){
        System.out.println(text);
    }
}
```

Source: http://tutorials.jenkov.com/java/lambda-expressions.html

Example

- **This interface can be implemented by lambda**
- **Although it has 3 methods, only 1 is unimplemented**

Lambda Parameters

- Zero Parameter

```
() -> System.out.println("No parameter");
```

- One Parameter

```
(param) -> System.out.println("1 parameter:"+ param);
param -> System.out.println("1 parameter:"+ param);
```

- Multiple Parameters

```
(param1, param2) -> System.out.println("2 parameters:" + param1 + ", " + param2);
```

Type Inference

- Compiler obtains most of the type information from *generics*
- Compiler won't be able to infer types if raw type (e.g., `List`) is used instead of the parameterized type (e.g., `List<String>`)

List is a raw type. References to generic type List<E> should be parameterized

```
List strList = new ArrayList();
strList.add("abc");
strList.add("bcd");
Collections.sort(strList, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

The method length() is undefined for the type Object

Use `List<String> strList = new ArrayList<String>()` to remove all the warnings and errors!

Lambda Function Body

- One statement

```
(param) -> System.out.println("1 parameter:"+ param);
```

- Multiple statements (with curly braces)

```
(param) -> {
    System.out.println("1 parameter:"+ param);
    return 0;
}
```

- Return

```
(param1, param2) -> {return param1 > param2;}
(param1, param2) -> param1 > param2;
```

In case all your lambda expression is doing is to calculate a return value and return it, you can specify the return value in a shorter way.

Does the code work?

```
String s1 = "";
Comparator<String> comp = (s1, s2) -> s1.length() - s2.length();
```

Lambda expression's parameter s1 cannot redeclare another local variable defined in an enclosing scope.

Local variable str defined in an enclosing scope must be final or effectively final

```
String str = "";
Comparator<String> comp = (s1, s2) -> {  str = str + " test";
                                        return s1.length() - s2.length();};
```

```
String str = "";
Comparator<String> comp = (s1, s2) -> { System.out.println(str);
                                        return s1.length() - s2.length();};
```

√ The local variable could be accessed without changing its value

More Use Cases

- Use Case I: create & "instantiate" a functional interface

```java
public interface MyInterface{

    // abstract method
    double getPiValue();

}


MyInterface ref = () -> 3.1415;
System.out.println("Pi = " + ref.getPiValue());}
```

- The lambda expression implements the abstract method; therefore, we could "instantiate" the interface
- Strictly, we are instantiating an anonymous class that implements MyInterface

More Use Cases

- Use Case II: executing the same operation when iterating elements

```
List<String> strList = new ArrayList<String>();
strList.add("abc");
strList.add("bcd");

//print every element in the list
strList.forEach(elem -> System.out.println(elem));
```

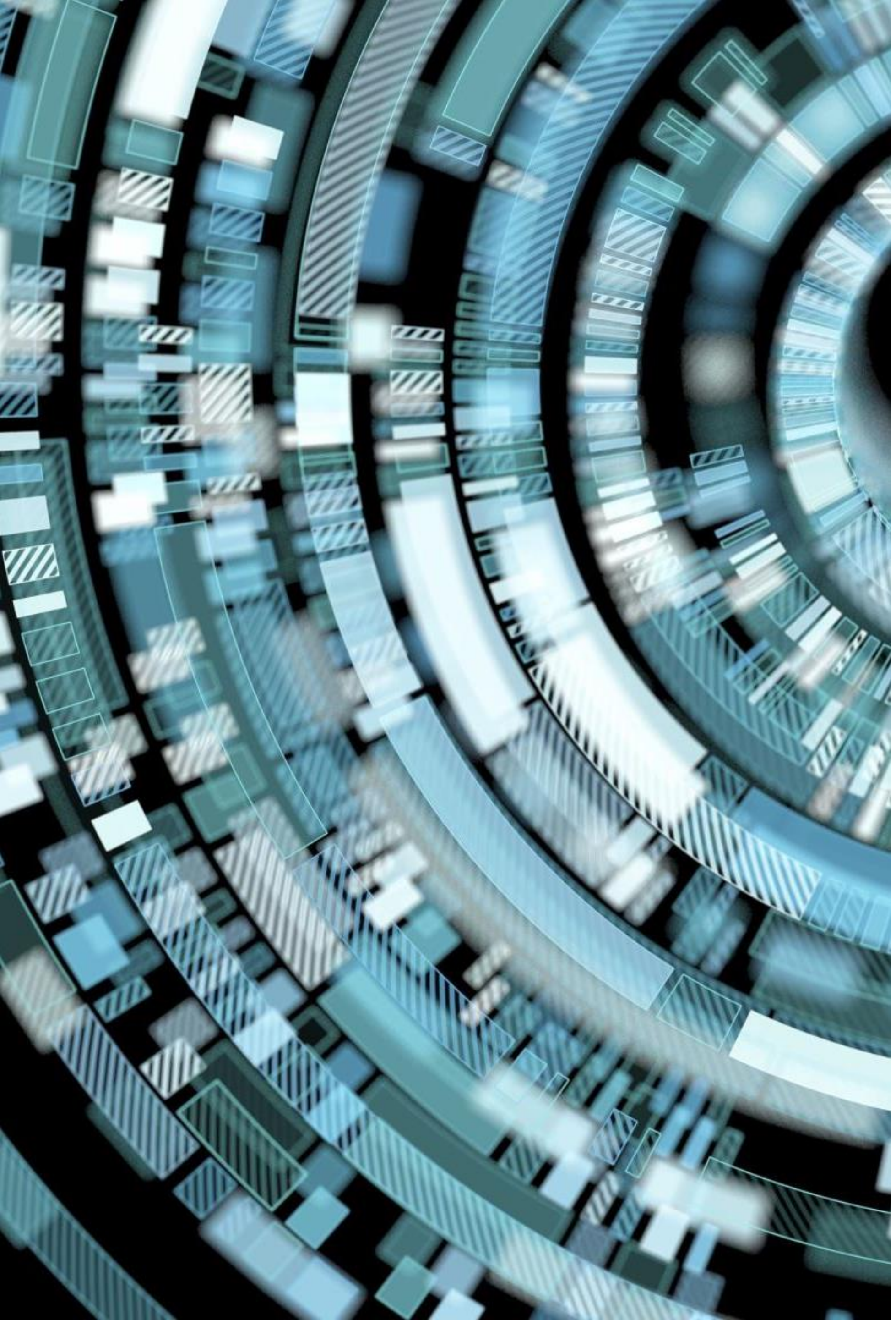
```
default void forEach(Consumer<? super T> action)
```

Only 1 abstract method

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

```
void accept(T t)
```

Performs this operation on the given argument.



Lecture 3

- Functional Programming
- Lambda Expressions
  - Syntax
  - Type inference
  - Use cases
  - Method references
  - Java Functional Interfaces

Method Reference

- Sometimes, a lambda expression does nothing but call an existing method
- Method reference allows us to refer to this existing method by name, which is often (but not always) easier to read

```
public interface MyInterface{
        public void print(String s);
}
```

```
// Using lambda expression
MyInterface ref = s -> System.out.println(s);

// Using method reference
MyInterface ref = System.out::println;
```

Class that owns the method (receiver)

The double colon indicates that this is a method reference

The method name

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

Static Method

- Methods declared with the `static` keyword
- Can be called without creating an object of a class (no need to `new` an instance)
- E.g., `Integer.parseInt`, `Math.min`

Instance Method

- Methods declared not with the static keyword
- Requiring an object of its class to be created before it can be called

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

Syntax

```
TypeName::staticMethod
```

Example

```
Integer::parseInt
```

Lambda Equivalent

```
str -> Integer.parseInt(str)
```

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

```
System.out::println
```

executes `println` on a sepcific instance of `PrintStream`, which is the `System.out` instance

Syntax

```
InstanceName::MethodName
```

- Here the **InstanceName (object reference)** represents any object instance.
- We call the **InstanceName** bounded receiver since the receiver is bounded to the instance.

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

```
public interface Deserializer {
    public int deserialize(String v1);
}
```

```
public class StringConverter {
    public int convertToInt(String v1){
        return Integer.valueOf(v1);
    }
}
```

```
StringConverter stringConverter = new StringConverter();

Deserializer des = stringConverter::convertToInt;
```

Example from http://tutorials.jenkov.com/java/lambda-expressions.html

- `convertToInt` and `deserialize` has the same signature
- The lambda uses a `StringConverter` instance `stringConverter` and refer to its `convertToInt` method

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

```
Syntax
ClassName::MethodName
```

- **ClassName** is the name of the class, such as **String**, **Integer**.
- We call **ClassName** unbounded receiver since the receiver instance is bounded later.
- Unbound receivers allow us to use instance methods as if they were static methods; However, the creation of instances are still required, but are deferred (decided later)

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

```java
public interface Transformer {
    String transform(String s);
}
```

This lambda

```java
Transformer transformer = (s) -> s.toLowerCase();
String res = transformer.transform( s: "This is Java");
```

Could be replaced by this method reference

```java
Transformer transformer = String::toLowerCase;
String res = transformer.transform( s: "This is Java");
```

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

```java
public interface Finder {
    public int find(String s1, String s2);
}
```

This lambda

```java
Finder finder = (s1, s2) -> s1.indexOf(s2);
int res = finder.find("this is a test","test");
```

Could be replaced by this method reference

```java
Finder finder = String::indexOf;
int res = finder.find("this is a test","test");
```

`(arg0, rest) -> arg0.MethodName(rest)` can be `ClassName::MethodName` (arg0 is of type `ClassName`)

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

Example

```
Supplier<String> s  = String::new
```

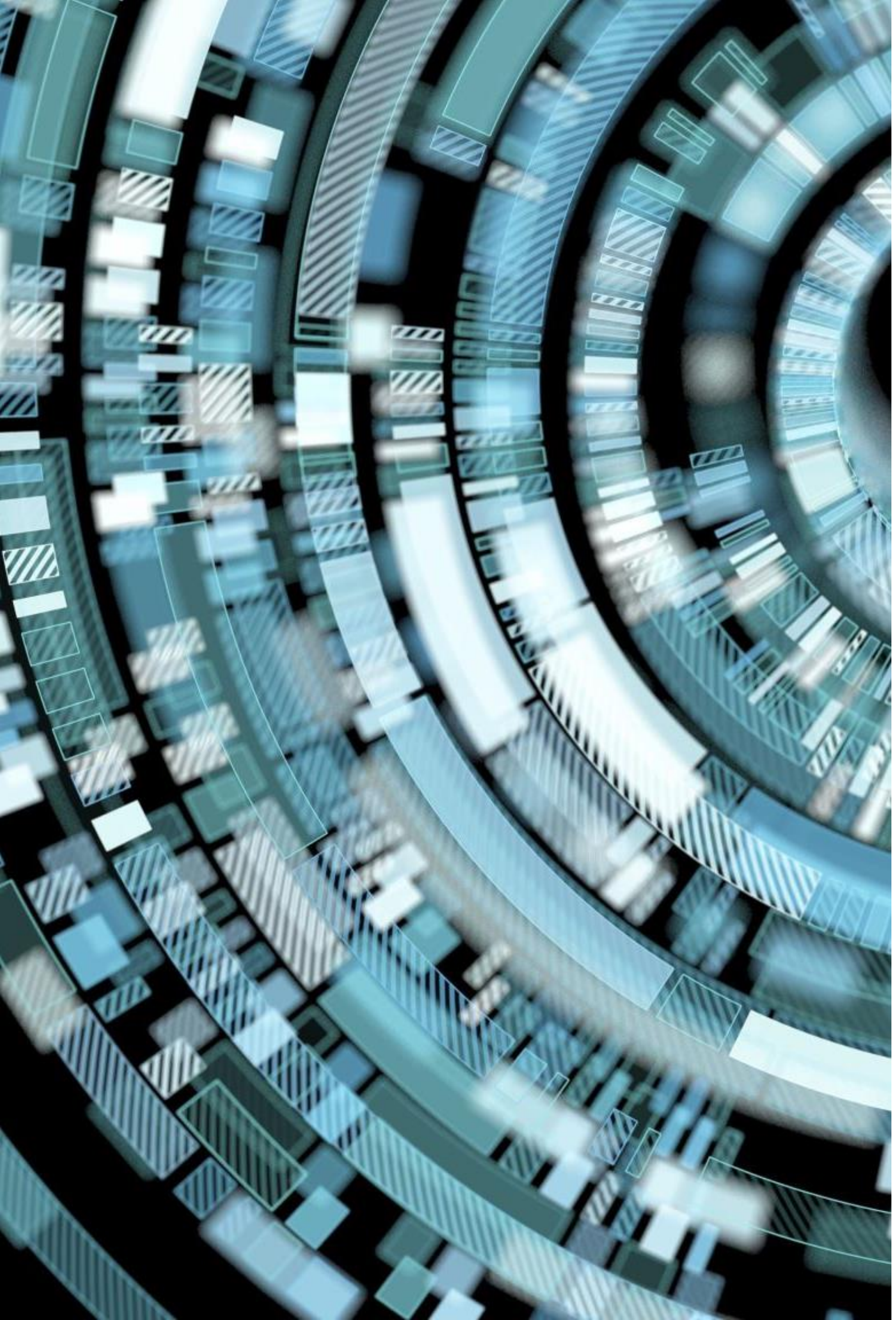
Lambda Equivalent

```
Supplier<String> s  = () ->  new String();
```

Same signature

```
public interface Supplier<T>
```

The `Supplier` interface has one abstract method `T get()`



Lecture 3

- Functional Programming
- Lambda Expressions
  - Syntax
  - Type inference
  - Use cases
  - Method references
  - Java Functional Interfaces

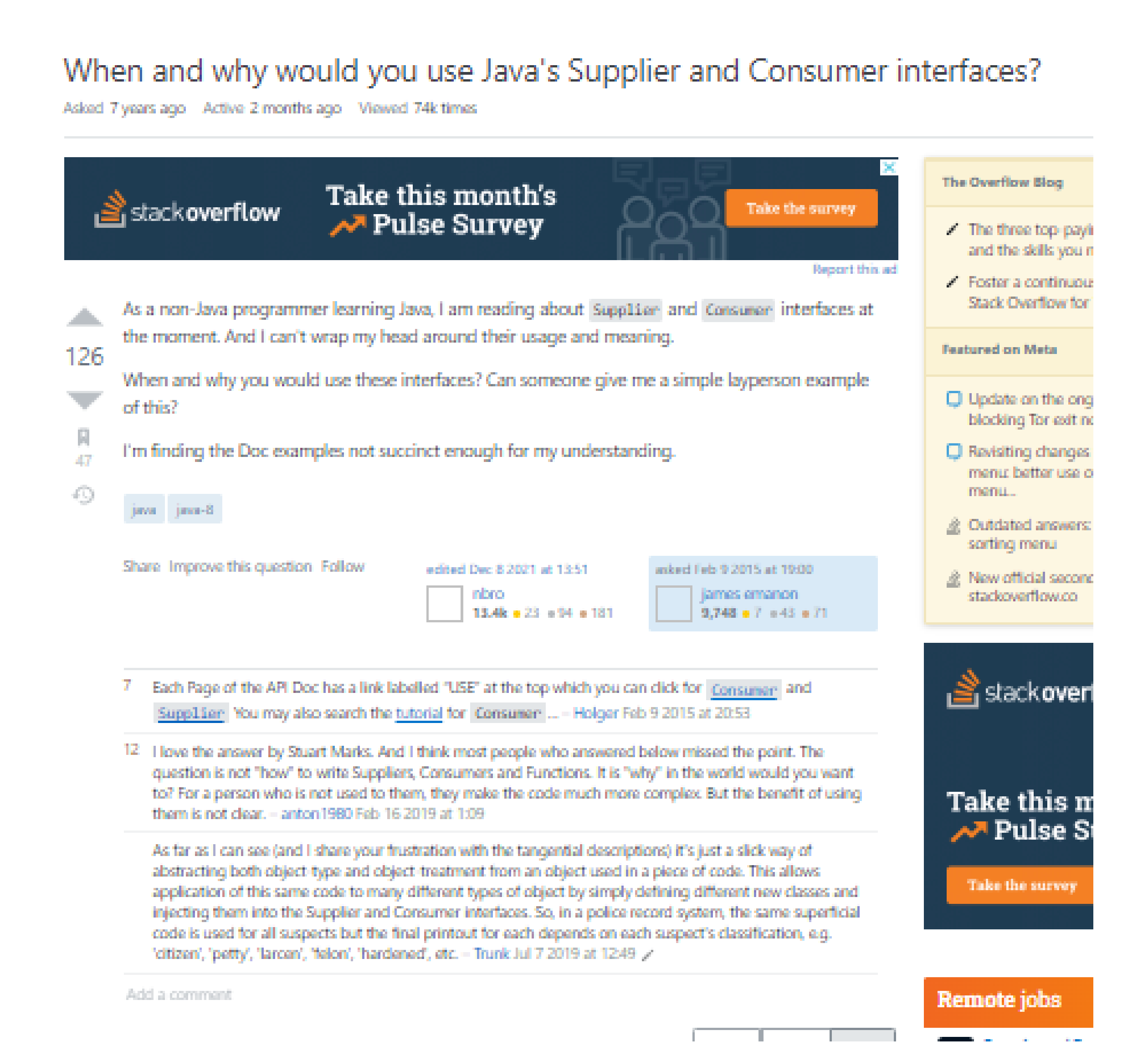
Built-in Functional Interfaces

- The `java.util.function` package
- Well defined set of general-purpose functional interfaces
  - All have only one abstract method
  - Lambda can be used wherever these interfaces are used
  - They are used extensively in Java class libraries, especially with the Streams API (later)

- We've already met them
  - `Consumer<T>` (page 31)
  - `Supplier<T>` (page 42)

Why do we need functional interfaces?

- Think of it as simply “functions”
- A functional interface defines the structure of that function (i.e., parameter types and return type)
- Functions can be used as parameters, and simplified using lambdas, thus making the code more compact



https://stackoverflow.com/questions/28417262

Consumer<T>

represents a function that takes an argument of type T and returns nothing (consume it)

```
List<String> strList = new ArrayList<String>();
strList.add("abc");
strList.add("bcd");

//print every element in the list
strList.forEach(elem -> System.out.println(elem));
```

```
public void forEach(Consumer<? super E> action)
```

```
void accept(T t)
```

Performs this operation on the given argument.

Supplier<T>

The `Supplier` interface has one abstract method `T get()`

represents a function that takes no argument and returns (supplies) a value of type T

```java
public class SupplierInterface {
    //Supplier function declarations.
    Supplier<String>  textSupplier     = () -> "Hello SW Test Academy!";
    Supplier<Integer> numberSupplier   = () -> 1234;
    Supplier<Double>  randomSupplier   = () -> Math.random();
    Supplier<Double>  randomSupplierMR = Math::random; //With Method Refe

    @Test
    public void supplierTest() {
        //Calling Supplier functions.
        System.out.println(textSupplier.get());
        System.out.println(numberSupplier.get());
        System.out.println(randomSupplier.get());
        System.out.println(randomSupplierMR.get());
    }
}
```

Example: https://www.swtestacademy.com/java-functional-interfaces/

Uses of Supplier in java.util.stream

Methods in java.util.stream that return Supplier

Modifier and Type	Method and Description
Supplier<A>	Collector.supplier() A function that creates and returns a n

Methods in java.util.stream with parameters of type Supplier

Modifier and Type	Method and Description
<R> R	Stream.collect(Supplier<R> sup Performs a **mutable reduction** ope
<R> R	DoubleStream.collect(Supplier< Performs a **mutable reduction** ope
<R> R	IntStream.collect(Supplier<R> s Performs a **mutable reduction** ope
<R> R	LongStream.collect(Supplier<R> Performs a **mutable reduction** ope
static DoubleStream	StreamSupport.doubleStream(Sup Creates a new sequential or paralle
static <T> Stream<T>	Stream.generate(Supplier<T> s)

Predicate<T>

The **`Predicate`** interface has one abstract method **`boolean test(T t)`**

represents a function that takes a value of type T and returns a boolean

```
default boolean removeIf(Predicate<? super E> filter)
```

Removes all of the elements of this collection that satisfy the given predicate.

```
List<String> list = new ArrayList<>();
list.add("This");
list.add("is");
list.add("a");
list.add("Java");
list.add("Course");
list.removeIf(e -> e.length()<3);
list.forEach(System.out::println);
```

Common Functional Interfaces

The `Operator` interfaces represent functions whose result and argument types are the same

The `Predicate` interface represents functions who take an argument and return a boolean

The `Function` interface represents functions whose result and argument types could differ

Interface	Function Signature	Example
`UnaryOperator<T>`	`T apply(T t)`	`String::toLowerCase`
`BinaryOperator<T>`	`T apply(T t1, T t2)`	`BigInteger::add`
`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`
`Function<T,R>`	`R apply(T t)`	`Arrays::asList`
`Supplier<T>`	`T get()`	`Instant::now`
`Consumer<T>`	`void accept(T t)`	`System.out::println`

Table from "Effective Java"

More Use Cases

https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach5

Lambda Expressions

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, su
unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such
functionality as method argument, or code as data.

The previous section, Anonymous Classes, shows you how to implement a base class without giving it a na
class seems a bit excessive and cumbersome. Lambda expressions let you express instances of single-me

This section covers the following topics:

- Ideal Use Case for Lambda Expressions
  - Approach 1: Create Methods That Search for Members That Match One Characteristic
  - Approach 2: Create More Generalized Search Methods
  - Approach 3: Specify Search Criteria Code in a Local Class
  - Approach 4: Specify Search Criteria Code in an Anonymous Class
  - Approach 5: Specify Search Criteria Code with a Lambda Expression
  - Approach 6: Use Standard Functional Interfaces with Lambda Expressions
  - Approach 7: Use Lambda Expressions Throughout Your Application
  - Approach 8: Use Generics More Extensively
  - Approach 9: Use Aggregate Operations That Accept Lambda Expressions as Parameters

Next Lecture

- Java 8 Stream API