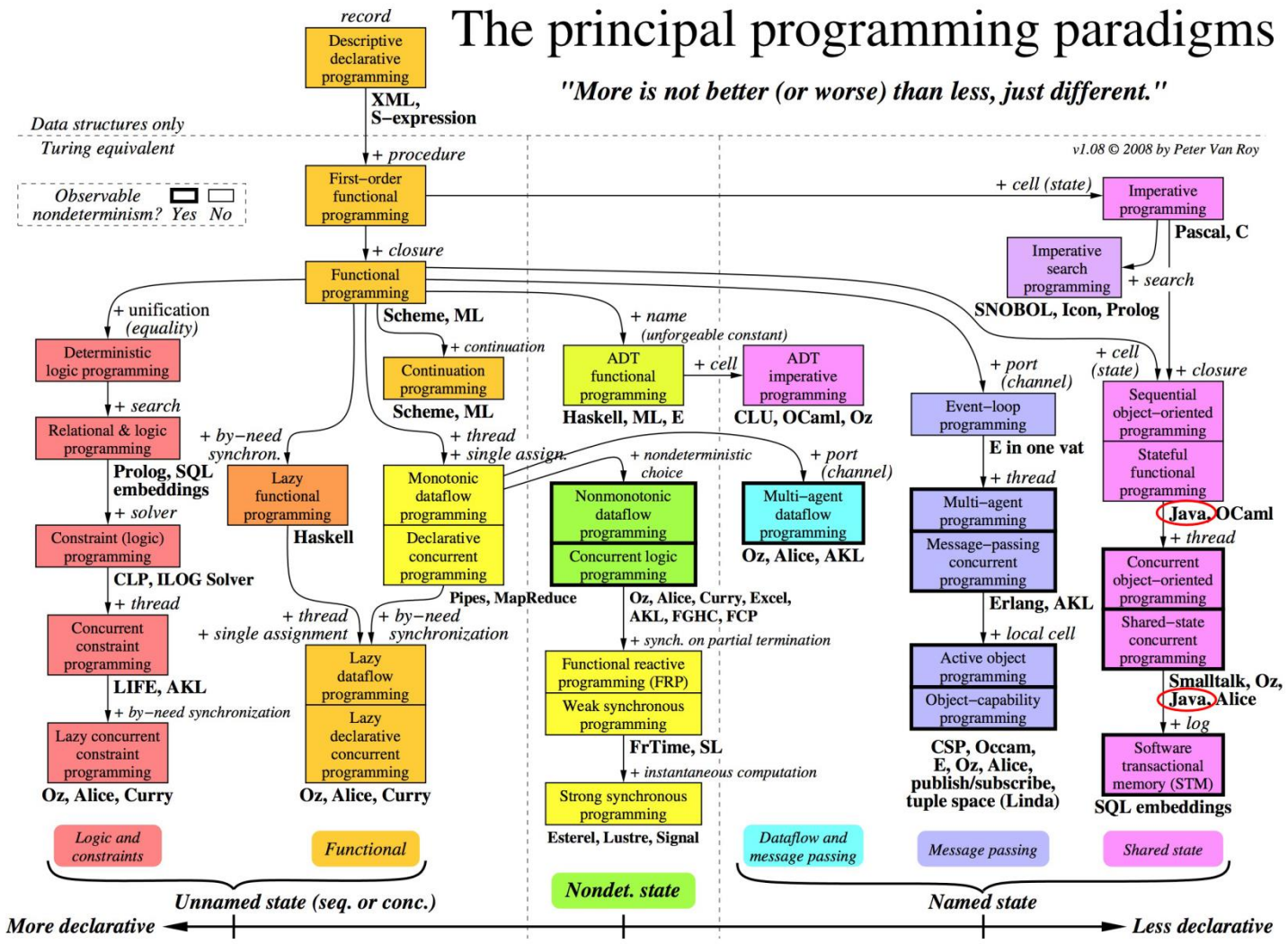
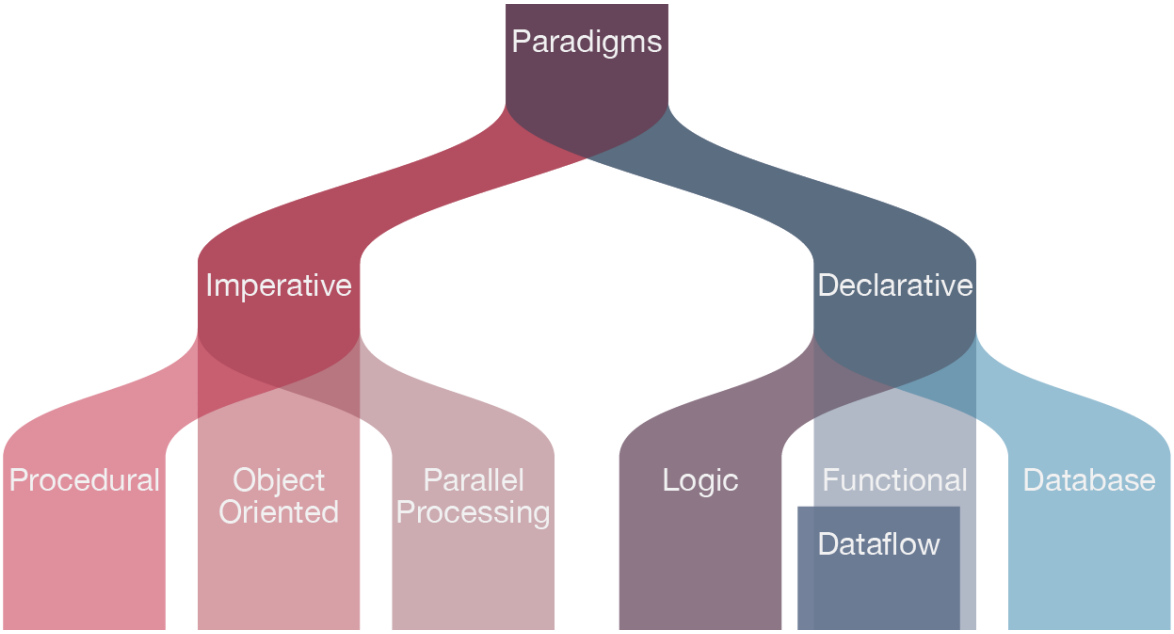


รูป 0-1 The principal programming paradigm



รูป 0-2 ประเภทของภาษา

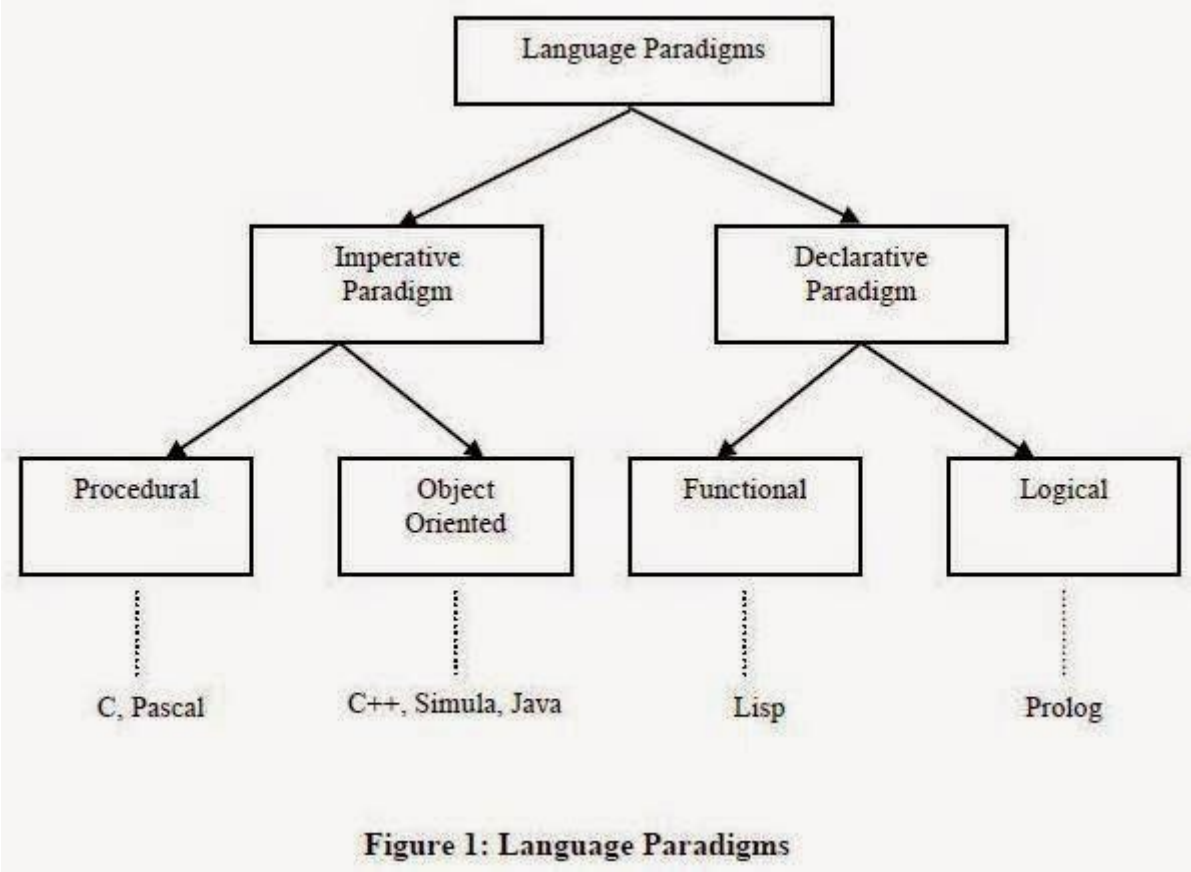
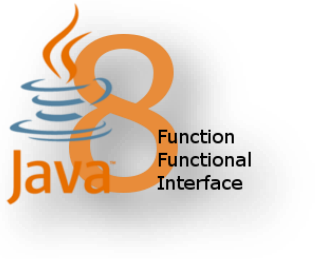
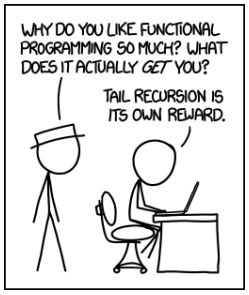
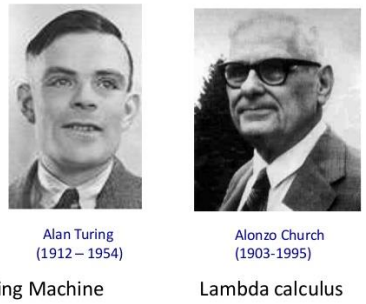


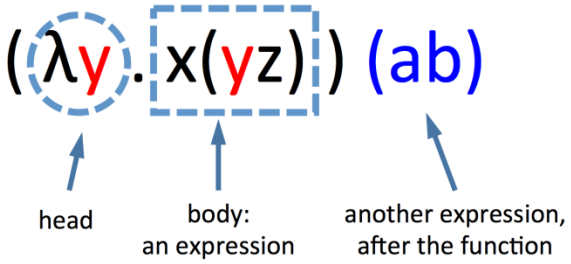
Figure 1: Language Paradigms



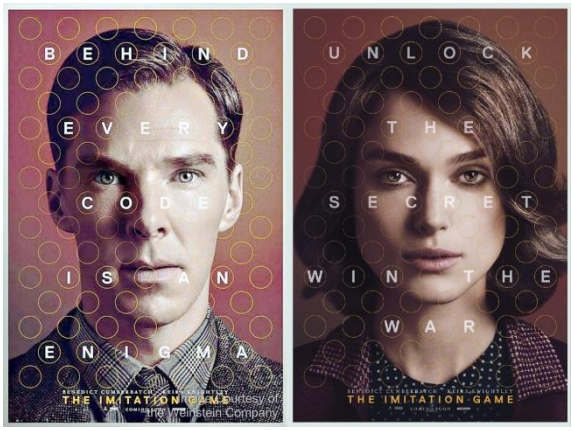


Functional Programming





มีพื้นฐานมาจาก Lambda calculus ซึ่งเป็นคณิตศาสตร์แขนงหนึ่ง ซึ่งคิดค้นโดย Alonzo Church ผู้เป็นอาจารย์ของ Alan Turing เจ้าของ Turing machine (คนเดียวกับตัวละครในภาพยนตร์เรื่อง The Imitation Game) เครื่องที่เป็นแบบจำลองทางคณิตศาสตร์ที่มีความสามารถระดับเดียวกับคอมพิวเตอร์ในปัจจุบัน แต่มีหน่วยความจำไม่จำกัด



จากเครื่องนี้ เราจะสามารถอธิบาย Algorithm ใดๆก็ตามด้วย Turing machine ได้ ซึ่งภายหลัง Turing เองก็ได้ทำการพิสูจน์ว่า Lambda calculus กับ Turing machine นั้นสมมูลกัน สามารถใช้แทนกันได้ทุกกรณี นั่นคือ เราสามารถใช้ functional programming มา Implement algorithm ใดก็ได้ที่ computable หรือพูดง่ายๆก็คือ โปรแกรมที่เราเคยเขียนได้ สามารถแปลงมาเป็นแบบ Functional Programming ได้หมด

ตัวอย่างภาษาที่เห็นใช้ชัดเจน เช่น Lisp, Clojure, Scala, Erlang, Javascript ฯลฯ

คุณสมบัติพื้นฐาน

1. **Pure Function (มีคุณสมบัติ Idempotence)**

   ฟังก์ชันรูปแบบที่ง่ายที่สุด ฟังก์ชันแบบปกติ เรียกใช้งาน และได้ผลลัพธ์เท่าเดิม

```
function add(a, b){
    return a + b;
}
```

2. **First-Class Function (มีคุณสมบัติ Referential transparency)**

   สามารถเอาฟังก์ชัน ใส่ตัวแปรได้ (ยกตัวอย่างเช่น ภาษา Javascript เป็นต้น) โดยจะเรียกว่า Anonymous Function

```
var add = function(a, b){
    return a + b;
}
```

3. **Higher-Order Function**

   ฟังก์ชั่นที่ Return ค่าคำตอบกลับมาเป็นฟังก์ชันได้ (ยกตัวอย่างเช่น ภาษา Javascript เป็นต้น)

```
var add = function(a){
    return function(b){
        return a + b;
    }
}
```

   เวลาเรียกใช้งาน

```
var add2 = add(2);
var ans = add2(3);
```

4. **Closures**

   อ้างอิงจาก High-Order Function ตัว Closure คือ ตัวฟังก์ชันตัวในสามารถเรียกตัวแปรข้างนอกนำมาใช้ได้ทั้งหมด แต่ด้านนอกใช้ตัวแปรของข้างในไม่ได้

5. **Immutable State**

   เก็บรักษา State เอาไว้ แม้ว่าจะมีการระบุค่าตัวใหม่ลงไปก็ตาม จะได้ค่าเท่าเดิม

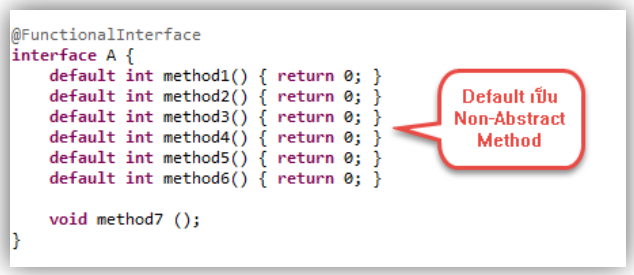
```
let x = 5;
x = 6;

print_int x;
```

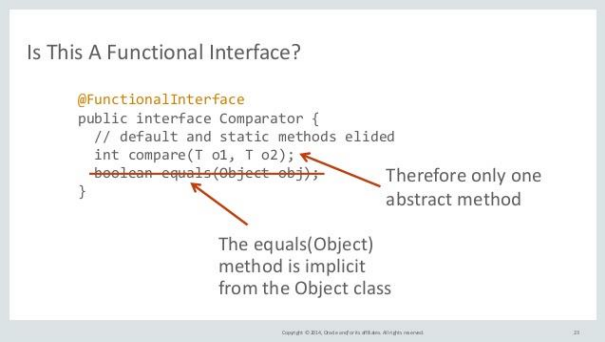
Functional Interfaces

จุดสำคัญ (Key Points)

- ✓ A functional interface is any interface that has exactly one abstract method.
  เป็น Interface ที่มีแค่ 1 Abstract Method เท่านั้น เรียกว่า **"*Single Abstract Method* (SAM) interfaces"**
- ✓ Since default methods have an implementation, they are not abstract so a functional interface can have any number of them.
  นับตั้งแต่นำ Default Methods มาใช้ และมันก็ไม่ได้เป็น Abstract ทำให้เราสามารถประกาศใช้เท่าไหร่ก็ได้



- ✓ If an interface declares an abstract method with the signature of one of the methods of java.lang.Object, it doesn't count toward the functional interface method count.
  ถ้า Interface ประกาศ Abstract Method ที่เป็น Signature ของคลาส Object จะไม่ถูกนับเป็น Abstract Method ในเรื่อง Functional Interface



Signature of Class Object Method

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

จุดสำคัญ (Key Points) (ต่อ)

- ✓ A functional interface is valid when it inherits a method that is equivalent but not identical to another.

  Functional Interface จะใช้ได้ก็ต่อเมื่อมัน inherit ตัว Method ที่คุณสมบัติเทียบเท่า แต่หน้าตาไม่เหมือนกัน

The situation is complicated by the possibility that two interfaces might have methods that are not identical but are related by erasure. For example, the methods of the two interfaces

```
interface Foo1 { void bar(List<String> arg); }
interface Foo2 { void bar(List arg); }
```

are said to be *override-equivalent*. If the (functional) superinterfaces of an interface contain override-equivalent methods, the *function type* of that interface is defined as a method that can legally override all the inherited abstract methods. In this example, if

```
interface Foo extends Foo1, Foo2 {}
```

then the function type of `Foo` is

```
void bar(List arg);
```

- ✓ An empty interface is not considered a functional interface.

  Empty Interface จะไม่ถูกเรียกว่าเป็น Functional Interface

```
@FunctionalInterface
interface EmptyInterface {

}
```

- ✓ A functional interface is valid even if the @FunctionalInterface annotation would be omitted.

  Functional Interface สามารถใช้ได้ ถึงแม้จะละเว้นประกาศ @FunctionalInterface

```
/*
interface IdenticalFunctional {
    void method(List l);
}
*/

@FunctionalInterface
interface IdenticalFunctional {
    void method(List l);
}
```

- ✓ Functional interfaces are the basis of lambda expressions.

  Functional interface ทั้งหมดล้วนเป็นรากฐานของเรื่อง Lambda Expression (บทที่ 9)

```
@FunctionalInterface
interface LambdaFunction {
    void call();
}

class BasisLambda {
    public static void workWithAnonymous(LambdaFunction lambdaFunction){
        lambdaFunction.call();
    }

    public static void main(String[] args) {
    // Anonymous Inner Class : JDK 1.1+
        workWithAnonymous(new LambdaFunction(){
            @Override
            public void call()
            {
                System.out.println("I am Anonymous Inner Class Function");
            }
        });

    // Lambda Expression : JDK 8+
        LambdaFunction lambdaFunction = () -> System.out.println("I am Lambda Function");
        lambdaFunction.call();
    }
}
```

บทสรุปการประกาศใช้ Functional Interface

```java
@FunctionalInterface
interface Functional {
    default void description() {

    }
    default void optional(int choice) {

    }

    boolean equals(Object o);
    int hashCode();
    String toString();

    void method(List<Double> l);

    // abstract void method();

    static void method2(){

    };
}

/*
interface IdenticalFunctional {
    void method(List l);
}
*/

@FunctionalInterface
interface IdenticalFunctional {
    void method(List l);
}

@FunctionalInterface
interface MainFunctional extends Functional, IdenticalFunctional {}

@FunctionalInterface
interface EmptyInterface {

}

@FunctionalInterface
interface LambdaFunction {
    void call();
}

class BasisLambda {
    public static void workWithAnonymous(LambdaFunction lambdaFunction){
        lambdaFunction.call();
    }

    public static void main(String[] args) {
    // Anonymous Inner Class : JDK 1.1+
        workWithAnonymous(new LambdaFunction(){
            @Override
            public void call()
            {
                System.out.println("I am Anonymous Inner Class Function");
            }
        });

    // Lambda Expression : JDK 8+
        LambdaFunction lambdaFunction = () -> System.out.println("I am Lambda Function");
        lambdaFunction.call();
    }
}
```

Default Method เป็น Non-Abstract Method

Object Method (Signature) เป็น Non-Abstract Method

(Abstract) Method สามารถประกาศได้เพียง 1 Abstract Method (SAM)

Static Method เป็น Non-Abstract Method

@FunctionalInterface สามารถละเว้นได้

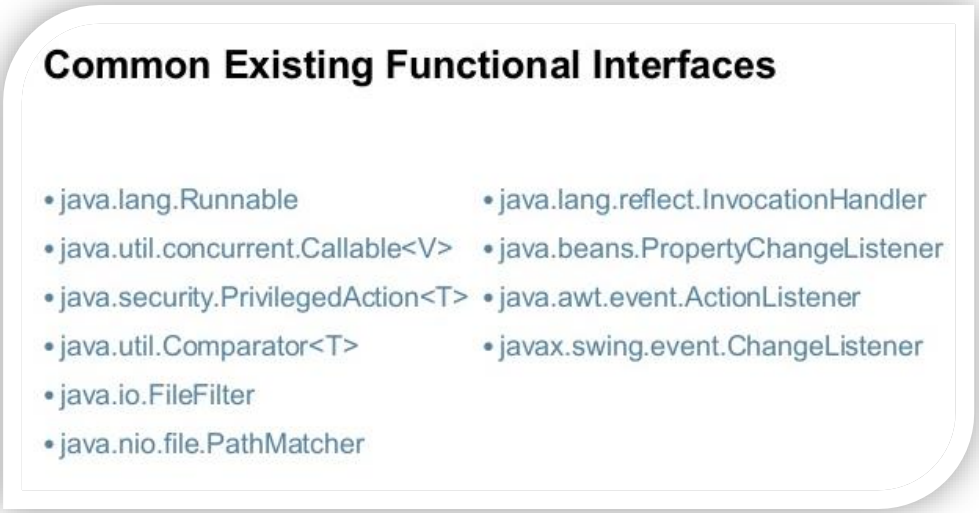
Override-Equivalent ด้วย Type Erasure เป็น Abstract Method

Empty Interface ไม่มีคุณสมบัติ Functional

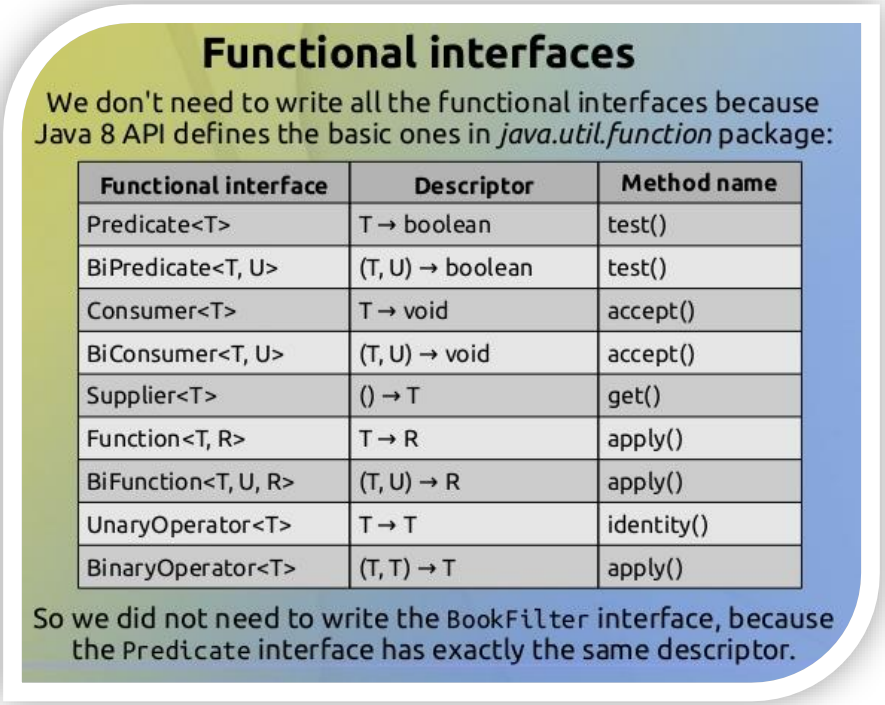
Functional Interfaces สามารถ Implement ผ่าน Anonymous Inner Class

Functional Interfaces เป็นรากฐานของ Lambda Expression

Common Functional Interfaces in JDK 1-7



Newer in JDK 8



Functional interfaces

We don't need to write all the functional interfaces because Java 8 API defines the basic ones in *java.util.function* package:

Functional interface	Descriptor	Method name
Predicate<T>	T → boolean	test()
BiPredicate<T, U>	(T, U) → boolean	test()
Consumer<T>	T → void	accept()
BiConsumer<T, U>	(T, U) → void	accept()
Supplier<T>	() → T	get()
Function<T, R>	T → R	apply()
BiFunction<T, U, R>	(T, U) → R	apply()
UnaryOperator<T>	T → T	identity()
BinaryOperator<T>	(T, T) → T	apply()

So we did not need to write the `BookFilter` interface, because the `Predicate` interface has exactly the same descriptor.