

Java Built-In Lambda Interfaces

lambda expression ต้องมีความสอดคล้องกับ functional interface

เราสามารถใช้ interface กับ lambda ได้ แต่ต้องมี abstract method เพียงอันเดียวใน functional interface

เราจะเห็นใน java api จะมี function interface เยอะแยะ ที่สามารถใช้กับ lambda expression ได้ เช่น `java.lang Runnable` หรือ `java.lang.Comparable`

อย่างไรก็ตาม java8 มี new functional interfaces เพื่อใช้กับ lambda โดยเฉพาะ เพื่อให้ครอบคลุม common scenarios

ตัวอย่าง 2 สถานะการณ์พื้นฐานคือ การกรองข้อมูลตามเงื่อนไขเฉพาะ และ test เงื่อนไขบางอย่าง

ใน chapters ก่อนหน้านี้ เราจะใช้

```
interface Searchable {  
    boolean test(Car c);  
}
```

ปัญหาคือเราต้องเขียน interface แบบนี้ในแต่ละ program เพื่อที่จะใช้ หรือ link ไปหาที่มีอยู่

new functional interface ถูกสร้างใน `java.util.function` package

มีทั้งหมด 5 ตัว ดังนี้

- `Predicate<T>`
- `Consumer<T>`
- `Function<T,R>`
- `Supplier<T>`
- `UnaryOperator<T>`

กรณีที่ T และ R ถูกแสดงเป็น generic types T จะแสดงถึง parameter type และ R จะหมายถึง return type

แชร์ เทคนิคการจำ pcs uf of tag aa = if input return in/out

นอกจากนี้ พวกเค้ายังมีของพิเศษ สำหรับ case ที่ input parameter เป็น primitive type จริงๆมีแค่ `int` , `long` , `double` และ `boolean` เฉพาะ `Supplier` ดังตัวอย่าง

- `IntPredicate`
- `LongConsumer`
- `BooleanSupplier`

เลือกคำแนะนำให้เหมาะสมกับ primitive type

เพิ่มเติม 4 binary versions ที่จะใช้ในกรณีส่ง 2 parameters

- `BiPredicate<T,R>`
- `BiConsumer<T,U>`
- `BiFunction<T,U,R>`

- BinaryOperator<T>

กรณีที่ T , U และ R ถูกแสดงเป็น generic types T , U หมายถึง parameter type และ R หมายถึง return type

table ข้างล่างจะแสดง list of interface ทั้งหมด
คุณไม่ต้องจำมันทั้งหมด เพียงแค่เข้าใจมันทั้งหมด

| Functional Interface | Primitive Versions |
|----------------------|--|
| Predicate<T> | IntPredicate LongPredicate DoublePredicate |
| Consumer<T> | IntConsumer LongConsumer DoubleConsumer |
| Function<T, R> | IntFunction<R> IntToDoubleFunction IntToLongFunction LongFunction<R> LongToDoubleFunction LongToIntFunction DoubleFunction<R> DoubleToIntFunction DoubleToLongFunction ToIntFunction<T> ToDoubleFunction<T> ToLongFunction<T> |
| Supplier<T> | BooleanSupplier IntSupplier LongSupplier DoubleSupplier |
| UnaryOperator<T> | IntUnaryOperator LongUnaryOperator DoubleUnaryOperator |

| Functional Interface | Primitive Versions |
|----------------------|---|
| BiPredicate<L, R> | |
| BiConsumer<T, U> | ObjIntConsumer<T> ObjLongConsumer<T> ObjDoubleConsumer<T> |
| BiFunction<T, U, R> | ToIntBiFunction<T, U> ToLongBiFunction<T, U> ToDoubleBiFunction<T, U> |
| BinaryOperator<T> | IntBinaryOperator LongBinaryOperator DoubleBinaryOperator |

Predicate

Predicate เป็น statement ที่จะ true หรือ false ขึ้นอยู่กับค่าของ variables functional interface นี้คุณสามารถเรียกใช้ได้ทุกที่ ที่คุณต้องการจะตรวจสอบเงื่อนไข true , false (boolean condition)

วิธีการกำหนด interface นี้

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    // Other default and static methods
    // ...
}
```

function descriptor (method signature) คือ

T -> boolean

วิธีใช้แบบ anonymous class

```
Predicate<String> startWithA = new Predicate<String>(){
    @Override
    public boolean test(String t){
        return t.startsWith("A");
    }
}
boolean result = startWithA.test("Arthur");
```

วิธีใช้ด้วย lambda expression

```
Predicate<String> startWithA = t -> t.startsWith("A");
boolean result = startWithA.test("Arthur");
```

ใน interface นี้ยังมี default method ดังนี้

```
default Predicate<T> and(Predicate<? super T> other)
default Predicate<T> or(Predicate<? super T> other)
default Predicate<T> negate()
```

method เหล่านี้ จะ return การเรียงเรียงของ Predicate ในการแสดง short-circuiting logical AND และ OR ของ predicate และอีกอย่างคือการปฏิเสธ logical(logical negation)

Short-circuiting หมายถึง ใน other predicate จะไม่ได้ตรวจสอบ เพราะว่าใน predicate แรก สามารถที่จะ predict ผลลัพธ์ได้เลย เช่น กรณีที่ first predicate return false ใน case AND หรือ return true ใน case OR

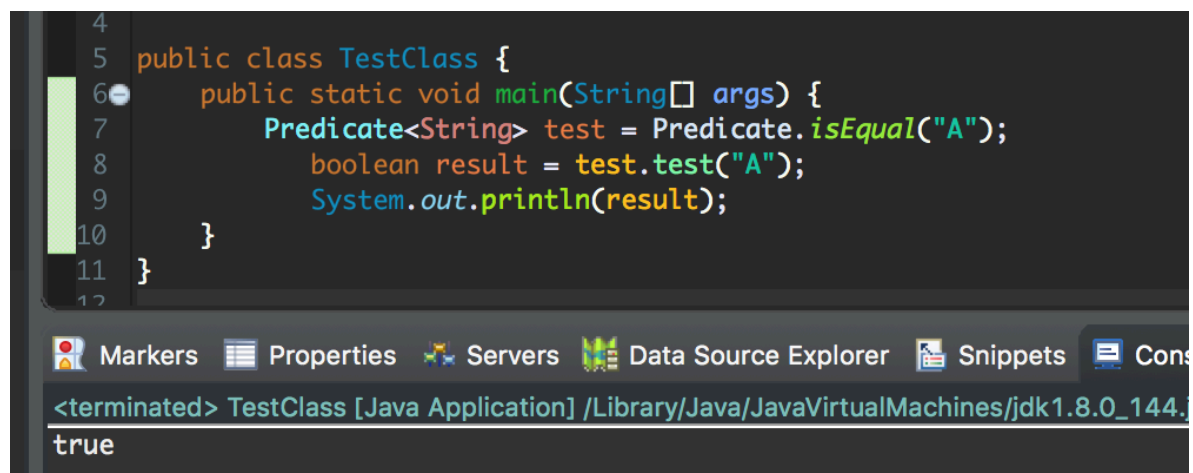
method พวกนี้ มีประโยชน์มากในการรวมกันของ predicates และทำให้ง่ายต่อการอ่าน code ดังตัวอย่าง

```
Predicate<String> startWithA = t -> t.startsWith("A");
Predicate<String> endsWithA = t -> t.endsWith("A");
boolean result = startWithA.and(endsWithA).test("Hi");
```

นอกจากนี้ ยังมี static method

```
static <T> Predicate<T> isEqual(Object targetRef)
```

จะ return Predicate ใน test ถ้า 2 argument มีค่าเท่ากันเหมือนกับ Object.equals(Object,Object)



```
4
5 public class TestClass {
6     public static void main(String[] args) {
7         Predicate<String> test = Predicate.isEqual("A");
8         boolean result = test.test("A");
9         System.out.println(result);
10    }
11 }
12
```

Markers Properties Servers Data Source Explorer Snippets Console

<terminated> TestClass [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_144.j

true

นอกจากนี้ยังมี primitive version สำหรับ int , long และ double พวกนี้ไม่ได้ extend จาก Predicate

จากตัวอย่าง เป็นการประกาศ IntPredicate

@FunctionalInterface

```
public interface IntPredicate {
    boolean test(int value);
    // And the default method : and , or , negate
}
```

แทนที่จะใช้

```
Predicate<Integer> even = t -> t % 2 == 0;  
boolean result = even.test(5);
```

คุณสามารถใช้

```
IntPredicate even = t -> t % 2 == 0;  
boolean result = even.test(5);
```

ทำไม?

เพื่อหลีกเลี่ยงการ conversion (auto boxing) ระหว่าง Integer กับ int และทำงานกับ primitive type โดยตรง

การประกาศแบบ primitive version ไม่ต้องมี generic type ในการเรียกใช้ parameters ของ functional Interface จะถูกจำกัดไว้

เนื่องจากการทำ conversion (auto boxing) wrapper type (Integer) to primitive type (int) ใช้ memory มากขึ้น และจะส่งผลกับ performance cost

java เลยให้ Primitive version มาเพื่อหลีกเลี่ยงการทำ auto boxing operations เมื่อ inputs or outputs เป็น primitive

Consumer

consumer เป็น operation สำหรับรับค่า argument เดียว และไม่ return อะไร เพียงแค่จะทำอะไรบางอย่างกับ argument

การประกาศ functional Interface

@FunctionalInterface

```
public interface Consumer<T> {  
    void accept(T t);  
    // And a default method  
    //..  
}
```

Functional descriptor (method signature) คือ

T -> void

ตัวอย่างใช้กับ anonymous class

```
Consumer<String> consumeStr = new Consumer<String>(){  
    @Override  
    public void accept(String t){  
        System.out.println(t);  
    }  
}  
consumeStr.accept("Hi");
```

วิธีใช้ผ่าน Lambda Expression

```
Consumer<String> consumeStr = t -> System.out.println(t);  
consumeStr.accept("Hi");
```

ใน Consumer ยังมี default method ให้เราด้วยคือ
default Consumer<T> andThen(Consumer<? super T> after)
ใน method นี้จะ return ส่วนประกอบในการดำเนินการของ Consumer ตามลำดับ การทำงานของ Consumer ตาม operation of the parameter

method นี้มีประโยชน์มากในการรวมกันของ Consumer ทำให้ Code ง่ายต่อการอ่าน
Consumer<String> first = t -> System.out.println("First:" + t);
Consumer<String> second = t -> System.out.println("Second:" + t);
first.andThen(second).accept("Hi");

The output is

First: Hi

Second: Hi

สังเกตว่า both Consumer ใช้ argument ตัวเดียวกัน และลำดับการทำงาน

Consumer ยังมี primitive version อีกด้วย int , long และ double พวกนี้ไม่ได้ extend Consumer

ตัวอย่างนี้ จะเป็นการประกาศ IntConsumer

@FunctionalInterface

```
public Interface IntConsumer {  
    void accept(int value);  
    default IntConsumer andThen(IntConsumer after){  
        //...  
    }  
}
```

แทนที่เราจะใช้

```
int[] a = { 1 , 2 ,3 , 4 ,5 , 6 , 7 ,8 , 9};
```

```
printList(a , t-> System.out.println(t));
```

```
//...
```

```
void printList(int[] a , Consumer<Integer> c) {
```

```
    for(int i : a){
```

```
        c.accept(i);
```

```
    }
```

```
}
```

เราสามารถใช้เป็น

```
int[] a = { 1 , 2 ,3 , 4 ,5 , 6 , 7 ,8 , 9};
```

```
printList(a , t-> System.out.println(t));
```

```
//...
```

```
void printList(int[] a , IntConsumer c) {
```

```
    for(int i : a){
```

```
        c.accept(i);
```

```
    }
```

```
}
```

Function

Function เป็น operation ที่รับ input argument ชนิดหนึ่ง (certain type) แล้วก่อให้เกิด result ใหม่ที่เป็นประเภทใหม่ (another type)

โดยปกติจะใช้ในการ convert หรือ transform object ไป another (เปลี่ยนหรือแปลงจากวัตถุหนึ่งไปอีกวัตถุหนึ่ง)

การประกาศ Functional Interface

```
@FunctionalInterface
```

```
public interface Function<T,R> {  
    R apply(T t);  
    // Other default and static methods  
    // ...  
}
```

Function descriptor (method signature) คือ

T -> R

ยกตัวอย่าง method

```
void round(double d , Function<Double , Long> f) {  
    long result = f.apply(d);  
    System.out.println(result);  
}
```

การใช้ผ่าน anonymous class

```
round(5.4 , new Function<Double , Long>() {  
    Long apply(Double d) {  
        return Math.round(d);  
    }  
});
```

การใช้งานผ่าน Lambda

```
round(5.4 , d -> Math.round(d));
```

ใน Interface นี้ยังมี default method อีกด้วย

```
default <V> Function<V,R> compose(  
    Function<? super V,? extends T> before)  
default <V> Function<T,V> andThen(  
    Function<? super R,? extends V> after)
```

ความแตกต่างระหว่าง method คือ

compose จะทำตัวเองเป็น parameter ก่อน โดย result จะมาเป็น input ใน function อื่น
andThen จะทำ function ที่เรียกก่อน แล้วจะ result ที่ได้ไปเป็น parameter ของ function ต่อไป

ตัวอย่าง

```
Function<String, String> f1 = s -> {  
    return s.toUpperCase();  
};  
Function<String, String> f2 = s -> {  
    return s.toLowerCase();  
};  
System.out.println(f1.compose(f2).apply("Compose"));  
System.out.println(f1.andThen(f2).apply("AndThen"));
```

The output is
COMPOSE
and then

ใน case ที่ 1 , f1 คือ function สุดท้ายที่ใช้
ใน case ที่ 2 , f2 คือ function สุดท้ายที่ใช้

ใน Interface นี้ มี static method ด้วย

```
static <T> Function<T, T> identity()
```

จะ return function ที่ return input output เหมือนกัน

```
Function<Integer,Integer> id = Function.identity();  
System.out.println(id.apply(3));
```

Primitive version สามารถนำมาใช้ int , long และ double แต่มันมีมากกว่า interface ก่อนหน้า

- ถ้าต้องการ return เป็น generic แต่ input เป็น primitive จะใช้ชื่อ XXXFunction เช่น IntFunction

@FunctionalInterface

```
public interface IntFunction<R> {  
    R apply(int value);  
}
```

- ถ้าต้องการใช้ function ที่ return primitive และ input เป็น generic ให้ใช้ ToXXXFunction เช่น ToIntFunction

@FunctionalInterface

```
public interface ToIntFunction<T> {  
    int applyAsInt(T value);  
}
```

- ถ้าต้องการให้ input primitive และ return primitive อื่น ให้ใช้ XXXToYYYFunction โดย XXX เป็น input (argument type) และ YYY เป็น return type

@FunctionalInterface

```
public interface IntToDoubleFunction {  
    double applyAsDouble(int value);  
}
```


Interface พวกนี้เป็นตัวช่วยให้ทำงานกับ primitive ได้สะดวกขึ้น เช่น
DoubleFunction<R> instead of Function<Double, R>
ToLongFunction<T> instead of Function<T, Long>
IntToLongFunction instead of Function<Integer, Long>

Supplier

supplier จะตรงข้ามกับ consumer มันจะไม่รับ argument แต่จะ return some value

การประกาศ

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

functional descriptor (method signature) คือ
() -> T

วิธีใช้ด้วย anonymous class

```
String t = "One";
Supplier<String> supplierStr = new Supplier<String>() {
    @Override
    public String get(){
        return t.toUpperCase();
    }
}
System.out.println(supplierStr.get());
```

วิธีใช้ด้วย lambda expression

```
String t = "One";
Supplier<String> supplierStr = () -> t.toUpperCase();
System.out.println(supplierStr.get());
```

Interface นี้ ไม่มี default method

Interface นี้มี primitive version ให้ด้วย int , long , double และ boolean พวกนี้ไม่ได้ extends Supplier

ตัวอย่างนี้เป็นการประกาศ BooleanSupplier

```
@FunctionalInterface
public interface BooleanSupplier {
    boolean getAsBoolean();
}
```

สามารถใช้แทน supplier ได้

UnaryOperator

UnaryOperator เป็นส่วนขยายของ function (extends function) เมื่อต้องการให้

argument และ result เป็น type เดียวกัน

การประกาศ

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
    // Just the identity
    // method is defined
}
```

functional descriptor (method signature) คือ

T -> T

วิธีใช้โดย anonymous class

```
UnaryOperator<String> uOp = new UnaryOperator<String>() {
    @Override
    public String apply(String t) {
        return t.substring(0,2);
    }
};
System.out.println(uOp.apply("Hello"));
```

วิธีใช้ด้วย lambda

```
UnaryOperator<String> uOp = t -> t.substring(0,2);
System.out.println(uOp.apply("Hello"));
```

Interface นี้ก็จะได้รับการสืบทอด method มา

```
default <V> Function<V,R> compose(
    Function<? super V,? extends T> before)
default <V> Function<T,V> andThen(
    Function<? super R,? extends V> after)
```

มีเพียง static method identity() เพราะว่า static method สืบทอดไม่ได้

```
static <T> UnaryOperator<T> identity()
```

return UnaryOperator ที่ input output เหมือนกัน

```
UnaryOperator<String> dd = UnaryOperator.identity();
System.out.println(dd.apply("3"));
```

แล้วยังมี primitive version อีกด้วย int , long และ double พวกนี้ไม่ได้ extend UnaryOperator

ตัวอย่างการประกาศ IntUnaryOperator

```
@FunctionalInterface
public interface IntUnaryOperator {
    int applyAsInt(int value);
}
```

```
        // Definitions for compose, andThen, and identity
    }
}
```

แทนที่จะใช้

```
int[] a = {1,2,3,4,5,6,7,8};
int sum = sumNumbers(a, t -> t * 2);
//...
int sumNumbers(int[] a, UnaryOperator<Integer> unary) {
    int sum = 0;
    for(int i : a) {
        sum += unary.apply(i);
    }
    return sum;
}
```

เป็นแบบนี้แทน

```
int[] a = {1,2,3,4,5,6,7,8};
int sum = sumNumbers(a, t -> t * 2);

//...
int sumNumbers(int[] a, IntUnaryOperator unary) {
    int sum = 0;
    for(int i : a) {
        sum += unary.applyAsInt(i);
    }
    return sum;
}
```

BiPredicate

Interface นี้จะแสดง Predicate ที่รับ argument 2 ตัว
การประกาศ

@FunctionalInterface

```
public interface BiPredicate<T, U> {
    boolean test(T t, U u);
    // Default methods are defined also
}
```

functional descriptor (method signature) คือ
(T, U) -> boolean

วิธีใช้ด้วย anonymous class

```
BiPredicate<Integer, Integer> divisible =
    new BiPredicate<Integer, Integer>() {
        @Override
        public boolean test(Integer t, Integer u) {
            return t % u == 0;
        }
    };
boolean result = divisible.test(10, 5);
```

วิธีใช้ด้วย Lambda Expression

```
BiPredicate<Integer, Integer> divisible =  
    (t, u) -> t % u == 0;  
boolean result = divisible.test(10, 5);
```

ใน interface นี้ไม่มี primitive versions

BiConsumer

Interface นี้แสดงผลเป็น Consumer ที่รับ argument 2 ตัว
การประกาศ

```
@FunctionalInterface  
public interface BiConsumer<T, U> {  
    void accept(T t, U u);  
    // andThen default method is defined  
}
```

functional descriptor (method signature) คือ
(T, U) -> void

การใช้งานผ่าน anonymous class

```
BiConsumer<String, String> consumeStr =  
    new Consumer<String, String>() {  
        @Override  
        public void accept(String t, String u) {  
            System.out.println(t + " " + u);  
        }  
    };  
consumeStr.accept("Hi", "there");
```

การใช้งานผ่าน Lambda Expression

```
BiConsumer<String> consumeStr =  
    (t, u) -> System.out.println(t + " " + u);  
consumeStr.accept("Hi", "there");
```

ใน Interface นี้ มี default method ให้ด้วย

```
default BiConsumer<T, U> andThen(  
    BiConsumer<? super T, ? super U> after)
```

ใน method นี้จะ return ส่วนประกอบในการดำเนินการของ BiConsumer ตามลำดับ การทำงานของ Consumer ตาม operation of the parameter
method นี้มีประโยชน์มากในการรวมกันของ BiConsumer ทำให้ง่ายต่อการอ่าน code

```
BiConsumer<String, String> first = (t, u) -> System.out.println(t.toUpperCase());  
BiConsumer<String, String> second = (t, u) -> System.out.println(t.toLowerCase());  
first.andThen(second).accept("Again", " and again");
```

ผลลัพธ์ที่ได้

AGAIN AND AGAIN again and again

ยังมี primitive versions สำหรับ int , long และ double พวกนี้ไม่ได้ extend BiConsumer แทนที่เราจะส่ง 2 argument

ในตัวอย่าง เราสามารถส่ง 1 object และ primitive value เป็น argument ที่สอง ดังนั้น การตั้งชื่อเรียกเปลี่ยนไปเป็น ObjXXXConsumer โดย XXX คือ primitive type ตัวอย่างต่อไปนี้เป็น ObjIntConsumer

การประกาศ

```
@FunctionalInterface
public interface ObjIntConsumer<T> {
    void accept(T t, int value);
}
```

แทนที่จะใช้

```
int[] a = {1,2,3,4,5,6,7,8};
printList(a, (t, i) -> System.out.println(t + i));
//...
void printList(int[] a, BiConsumer<String, Integer> c) {
    for(int i : a) {
        c.accept("Number:", i);
    }
}
```

เราสามารถให้

```
int[] a = {1,2,3,4,5,6,7,8};
printList(a, (t, i) -> System.out.println(t + i));
//...
void printList(int[] a, ObjIntConsumer<String> c) {
    for(int i : a) {
        c.accept("Number:", i);
    }
}
```

BiFunction

จะแสดงผลเป็น function ที่รับ argument 2 ตัว

การประกาศ

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u); // Other default and static methods
    // ...
}
```

functional descriptor (method signature) คือ

(T, U) -> R

Assuming a method

```
void round(
    double d1, double d2, BiFunction<Double, Double, Long> f) {
    long result = f.apply(d1, d2);
    System.out.println(result);
}
```

วิธีใช้ด้วย anonymous class

```
round(5.4, 3.8, new BiFunction<Double, Double, Long>() {  
    Long apply(Double d1, Double d2) {  
        return Math.round(d1 + d2);  
    }  
});
```

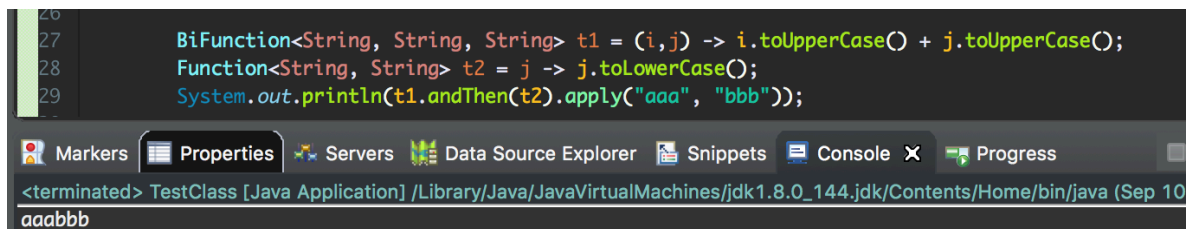
วิธีใช้ด้วย Lambda Expression

```
round(5.4, 3.8, (d1, d2) -> Math.round(d1, d2));
```

ใน interface ไม่เหมือน Function มีเพียง default method อันเดียวคือ

```
default <V> Function<T, V> andThen(  
    Function<? super R, ? extends V> after)
```

andThen จะทำ function ที่เรียกก่อน แล้วจะ result ที่ได้ไปเป็น parameter ของ function ต่อไป



```
26  
27 BiFunction<String, String, String> t1 = (i,j) -> i.toUpperCase() + j.toUpperCase();  
28 Function<String, String> t2 = j -> j.toLowerCase();  
29 System.out.println(t1.andThen(t2).apply("aaa", "bbb"));  
  
Markers Properties Servers Data Source Explorer Snippets Console X Progress  
<terminated> TestClass [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_144.jdk/Contents/Home/bin/java (Sep 10  
aaabbb
```

Interface นี้ยังมี primitive versions น้อยกว่า function มีเพียงแค่รับ generic type แล้ว return เป็น int , long และ double primitive types
เราจะประกาศด้วย ToXXBiFunction โดยที่ XXX คือ Primitive type

ตัวอย่างการประกาศ ToIntBiFunction

```
@FunctionalInterface  
public interface ToIntBiFunction<T, U> {  
    int applyAsInt(T t, U u);  
}
```

สามารถใช้แทน BiFunction ได้

BinaryOperator

เป็นตัวที่ Extend BiFunction ใช้เมื่อต้องการให้ argument และ result เป็น type เดียวกัน
วิธีการประกาศ

```
@FunctionalInterface  
public interface BinaryOperator<T>  
    extends BiFunction<T,T,T> {  
    // Two static method are defined  
}
```

functional descriptor (method signature) คือ
(T, T) -> T

วิธีใช้โดย anonymous class

```
BinaryOperator<String> binOp = new BinaryOperator<String>() {  
    @Override  
    public String apply(String t, String u) {  
        return t.concat(u);  
    }  
};  
System.out.println(binOp.apply("Hello", " there"));
```

วิธีใช้โดย Lambda Expression

```
BinaryOperator<String> binOp = (t, u) -> t.concat(u);  
System.out.println(binOp.apply("Hello", " there"));
```

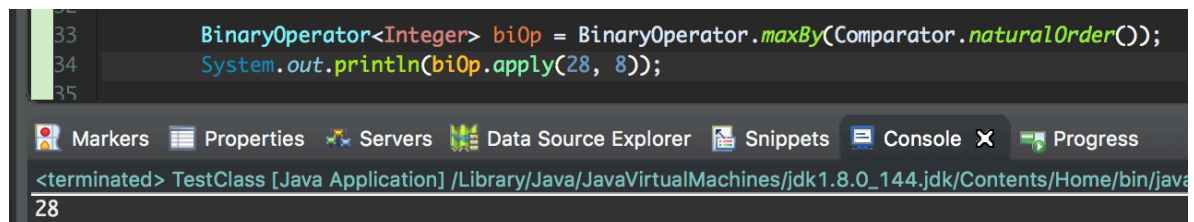
Interface นี้ได้รับการสืบทอด method จาก BiFunction

```
default <V> Function<T, V> andThen(  
    Function<? super R, ? extends V> after)
```

และมีการประกาศ static method ขึ้นมาใหม่

```
static <T> BinaryOperator<T> minBy(  
    Comparator<? super T> comparator)  
static <T> BinaryOperator<T> maxBy(  
    Comparator<? super T> comparator)
```

พวกนี้ return BinaryOperator โดย return ค่า น้อยกว่า มากกว่า ของ 2 element โดยใช้ Comparator
ดังตัวอย่าง



Comparator.naturalOrder() returns comparator เพื่อเปรียบเทียบ object ใน natural order โดยใช้ผ่าน apply() method แล้วส่ง argument 2 ตัว ผ่าน BinaryOperator

ยังมี primitive versions สำหรับ int , long และ double ที่ argument 2 ตัว และ return type เป็น primitive type เดียวกัน พวกนี้ไม่ได้ extend BinaryOperator หรือ BiFunction

ดังตัวอย่าง IntBinaryOperator

การประกาศ

```
@FunctionalInterface  
public interface IntBinaryOperator {  
    int applyAsInt(int left, int right);  
}
```

สามารถใช้แทน BinaryOperator ได้

Key Points

- Java 8 มี functional interfaces ใหม่ เพื่อใช้งานกับ lambda expressions เพื่อให้ครอบคลุม common scenarios จะอยู่ใน package java.util.function
- มีทั้งหมด 5 ตัวหลักๆ คือ
1. Predicate
 2. Consumer
 3. Function
 4. Supplier
 5. UnaryOperator
- Interface พวกนี้มี primitive versions สำหรับ int , long , double และ boolean (เฉพาะ Supplier) เพื่อหลีกเลี่ยงการ converting wrapper class เป็น primitive value เช่น Integer to int
 - Interface ที่รับ argument เดียว (generic type T) but Supplier ไม่ต้องส่ง argument พวกนี้จะมี version ที่สามารถส่ง 2 argument ได้ โดยเรียกผ่าน binary version
 - Predicate สามารถใช้ได้ทุกที่ที่ต้องการจะตรวจสอบเงื่อนไข boolean method signature คือ T -> boolean
 - Predicate มี Primitive version int , long , double ตัวอย่างเช่น IntPredicate
 - Consumer รับ argument เดียวและไม่ return function descriptor คือ T -> void
 - Consumer มี Primitive version int , long , double ตัวอย่างเช่น IntConsumer
 - Function รับ argument เดียว และ return type อื่น function descriptor คือ T -> R
 - Function มี Primitive version มาก int , long , double สามารถแบ่งเป็น 3 ประเภทหลักๆ
 - ถ้าต้องการจะใช้ Function ที่ return generic Type แล้วรับ primitive type ให้เรียก XXXFunction ตัวอย่างเช่น IntFunction
 - ถ้าต้องการจะใช้ Function ที่ return primitive type แล้วรับ generic type ให้เรียก ToXXXFunction เช่น ToIntFunction
 - ถ้าต้องการจะใช้ Function ที่ รับ primitive type แล้ว return primitive type อื่น ให้เรียก XXXToYYYFunction โดย XXX คือ argument type และ YYY คือ return type เช่น IntToDoubleFunction
 - Supplier ไม่ต้องส่ง argument แต่จะ return บางอย่าง function descriptor คือ () -> T
 - Supplier มี Primitive version สำหรับ int , long , double และ boolean เช่น IntSupplier
 - UnaryOperator เป็นส่วนเสริมของ Function เป็นส่วนที่ extend มาจาก function จะใช้ก็ต่อเมื่อต้องการให้ argument และ result เป็น type เดียวกัน functional descriptor คือ T -> T
 - UnaryOperator มี Primitive version สำหรับ int , long , double เช่น IntUnaryOperator
 - BiPredicate จะเหมือน Predicate แต่รับ 2 argument function descriptor คือ (T,U) -> boolean
 - BiPredicate ไม่มี primitive versions
 - BiConsumer จะเหมือน consumer แต่รับ 2 argument function descriptor คือ (T,U) -> void

- BiConsumer มี Primitive versions สำหรับ int , long ,double ส่ง Object เดียวและ primitive value วิธีเรียกจะต่างออกไปเป็น ObjXXXConsumer โดยที่ XXX คือ Primitive type เช่น ObjIntConsumer
- BiFunction จะเหมือน function แต่จะส่ง 2 argument และจะ return type อื่นออกมา function descriptor คือ (T,U) -> R
- BiFunction จะมี Primitive versions ที่จะส่ง generic type แล้ว return Primitive Type สำหรับ int , long , double โดยเรียกผ่าน ToXXXBifunction โดย XXX คือ Primitive Type
- BinaryOperator เป็นส่วนเสริมของ BiFunction เป็นส่วนที่ Extend มา จะใช้เมื่อต้องการให้ argument และ result เป็น type เดียวกัน function descriptor คือ (T , T) -> T
- BinaryOperator มีการประกาศ static method อยู่ 2 ตัว คือ

```
static <T> BinaryOperator<T> minBy(
    Comparator<? super T> comparator)
static <T> BinaryOperator<T> maxBy(
    Comparator<? super T> comparator)
```

- BinaryOperator มี Primitive versions สำหรับ int , long , double เช่น IntBinaryOperator

PCS U FUNCTION

TAG A A

IF INPUTRETRUN IN/OUT

| To | L | I | D |
|----|---|---|---|
| L | | ✓ | ✓ |
| I | ✓ | | ✓ |
| D | ✓ | ✓ | |

X ObjLID XX LIDAS ToLIDAS

LID LID LIDBAS LIDAS LID

<T>

<T,R>

P C S U FUNCTION

T A G A A

IF INPUT RETRUN IN/OUT