

### Algorithm Classes

1. Arrays.
2. Collections.

@ Provide common functionalities can be applied on Array `int []` and Collection

`Collections.concurrentList();` // Thread safe equivalent



## Java - 8

### Functional-Programming

- # functional interface

- # default methods

- # static methods

- # lambdas

- # streams

- # methods references

DateTime API

Optional

Nashorn engine (javascript engine)

Extension in collection API



Traditional :( Pure OOPs ) : Imperative

Functional : Declarative

Imperative :

How : focus

Pure OOPs

Embraces data mutability

Declarative

What : focus

Functional Programming (pure function)

Data immutability

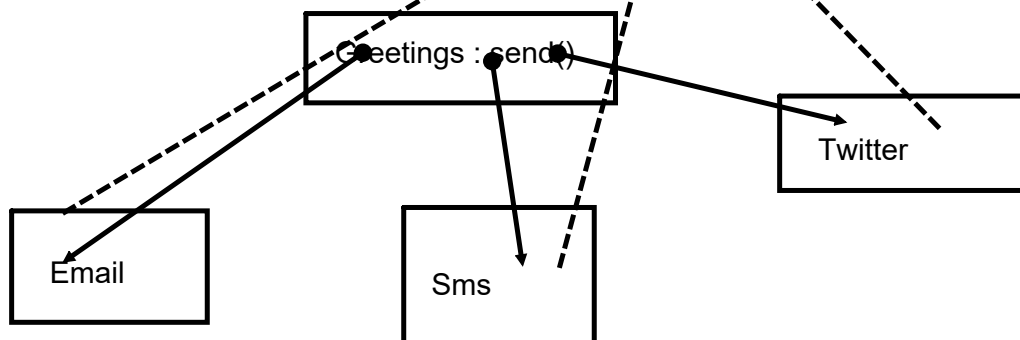
SQL style

Interfaces:

# can have functions with definitions ( default methods)

# static methods with definitions

```
public void conveyMessage(String message, Greeting){  
    // call a single method send()  
}
```



Lambda Syntax for pure function

1. not have any accessibility modifier (nothing related to class)
2. Do not any name (anonymous function)
3. no return type
4. no param type
5. (<param>) -> { <definition> }

```
void fun(String str1, int n){  
}  
(str1,n) -> {  
}
```

```
void fun(String str1){  
}  
str1 -> {  
}
```

```
void fun(){  
}  
() -> {  
}
```

```
void fun(String str1){  
    // only single instruction  
}  
str1 -> single instruction
```

```
void fun(String str1){  
    // instruction  
    return a;  
}
```

```
str1 -> {  
    // instruction  
    return a;  
}
```

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

```
// for single inst not bounded in braces return is default associated  
(a, b) -> a+b;
```

interface <reference> = <lambda expression>

# only when interface is functional interface :

contains only one abstract methods

can have any number of default/static

# refrence of an interface can refer to only those lambda expression whose signature matches with the only abstract method inside the interface

Runnable :

Comparator

Comparable

# few protoypes have been identified which are common in use

api : functional interface

Consumer : void accept(<>), BiConsumer [ two param ], IntegerConsumer()

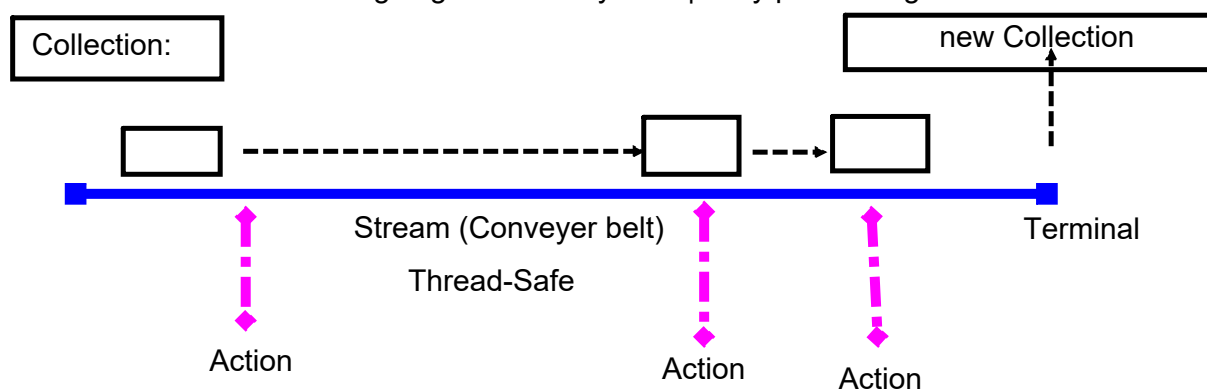
Predicate : boolean test(<>)

Function : <> apply(<>)

Supplier : <> get()

Streams : Safe/efficient way to process a collection

Not a data-structure/ not going to store any data | Lazy processing model



Stream method

1. action <intermediary methods>
2. Terminal method (stops the stream) cannot add any new activity after it

Stream are not going to initiate if no terminal activity is there