==>Java 8-11

==>Reactive Programming (Spring)

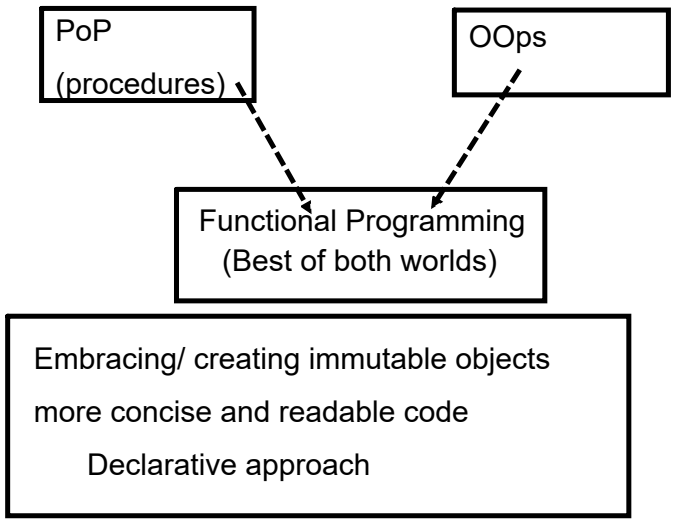
java 1.5 Functional Programming

java 1.8

==> Technological advancement : mobile/laptops/system

Java new feature simplifies concurrency operations

Functional Programming:



Traditional : Imperative

=> Focus on how to perform

=> Object mutability

Declarative Style:

=>Focus on what result we want

=>Object immutability

=>Analogous to SQL (use of already existing part of library to achieve an objective)

Improvisation on interface:

default method

static method

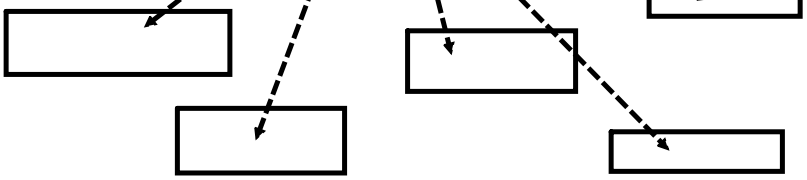
Demand from developer community :

#provision to add new functionality at the top without breaking existing API

interface exists at top level

10 functionality + 2

eg : Stream feature req to be added in existing collection api



Functional Interface : Lambda

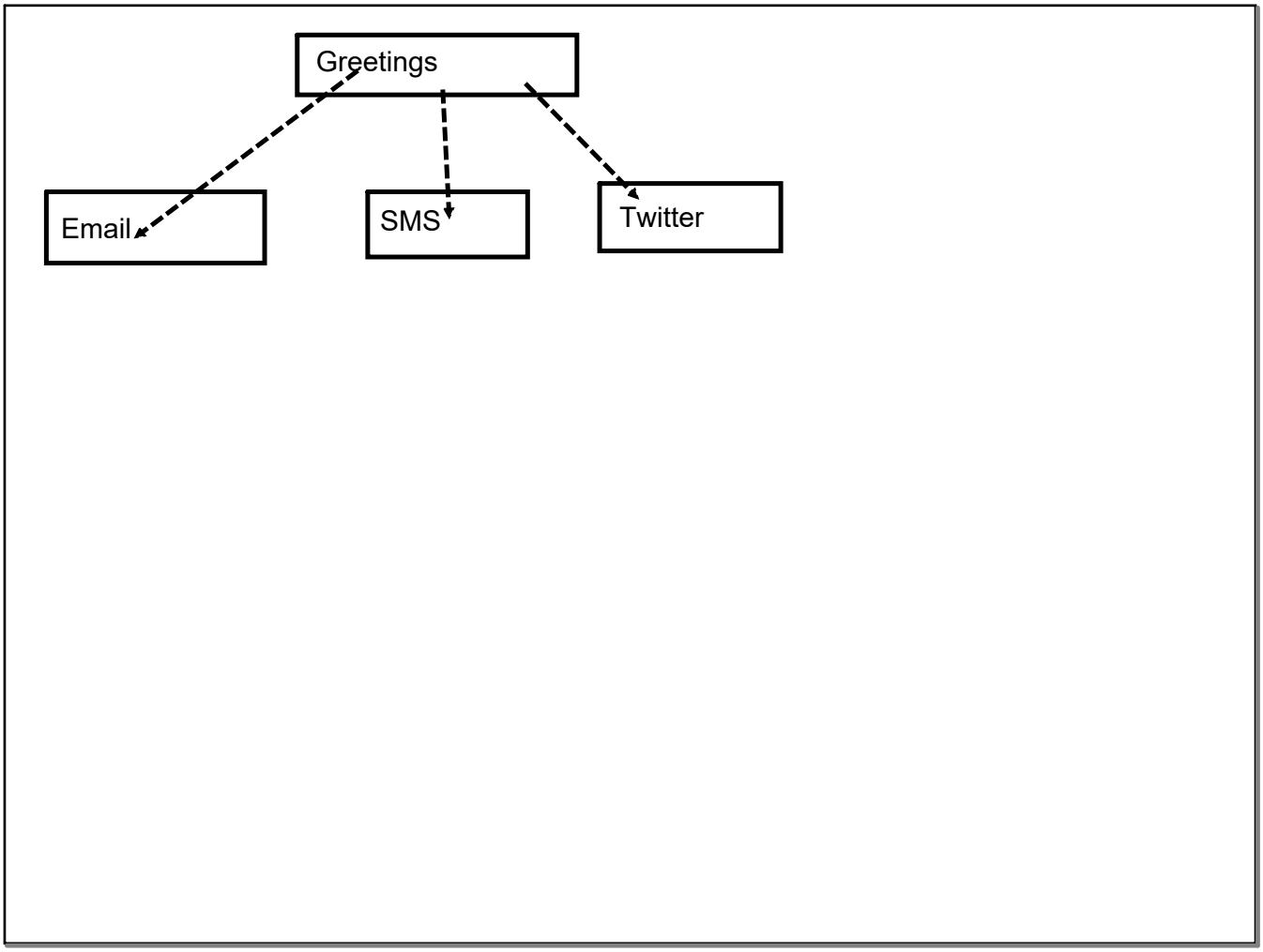
Contains only one abstract method

might have static method, default method (in any count)

jdk1.8 : special annotation

@FunctionalInterface (Compile time)/ optional

would restricts addition of any more abstract method


Greetings
Email
SMS
Twitter

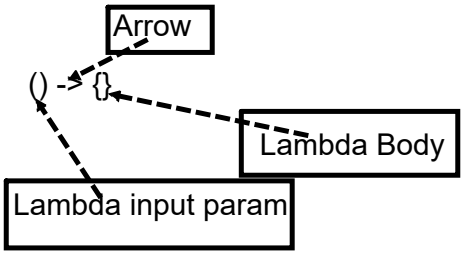
Interface can now act as a <Function type>

1. Functional Interface
2. Function definition that its reference is going to hold must match the prototype only abstract method inside it

Lambda expression:

1. anonymous function
2. method param, method body, return spec
3. Not encapsulated under any class
4. can be assigned to a variable, can be passed around

Lambda expression syntax : Compact



no class implementation

no object management

saved lots of runtime overheads

```
(msg) -> {}
msg -> {} // if single param no need to bind in para
(msg, other) -> {} // multiple, it is necessary
() -> {} // if no param
msg -> single instruction // no need to bind in braces
(a,b) -> a+b // return a+b
// if no braces then single stmt is by default associated with return
```

Functional Interface : (SAM)

=>Runnable

=>Comparator

```
interface Runnable{
    void run()
}

interface Comparator<T>{
    int compare(T o1, T o2);
}
```

New Functional Interfaces of Java 8

Lambdas connect themselves with specific signature

Java 8 has been introduced with group of
functional interfaces containing some very common prototype method
Usage of them has been updated in existing APIs

java.util.function

Consumer : BiConsumer, <Primitive type implementation>

Predicate : BiPredicate, <Primitive type implementation>

Function : BiFunction, UnaryOperator, BinaryOperator

Supplier

Consumer interface :

Single abstract method

void accept(<T>)

BiConsumer interface :

Single abstract method

void accept(<T>, <M>)

Chaining :

Almost all functional interface has chaining facility

#connect multiple implementations of same interface

default / static

Predicate

Single abstract method

boolean test(<T>)

Chaining :

and

or

negate

Function

only abstract method

<R> apply(<T>)

Chaining

andThen()

compose();

BiFunction

R apply(T1, T2)

UnaryOperator:

extension of Function interface

passing param type and

return value type is same

BinaryOperator:

extension of BiFunction interface

passing 2 param type and

return value type is same

2 static methods of BinaryOperator

both returns a lambda ref, which when called will return the max/min values

maxBy(<Comparator>)

minBy(<Comparator>)

Supplier FunctionalInterface

only abstract method

<T> get();

No chaining methods available...

1.. Predicate that can filter on variable value

Function functional implementation : return back such predicate

Method References

simplifies the implementation of Functional interfaces

Shortcut for writing lambda expression

=> to use already existing methods as lambdas

Syntax:

ClassName :: instance-method name

ClassName :: static-method name

instance :: method-name

1. WE are able call instance method through class name (conflict : not allowed all times)
2. method reference not matching the method signature of consumer is still valid

```
Student :: printAllActivities; Shortcut of writing lambda
(recieve an object of type Student

student -> student.printAllActivities();
```

Any instance when called, is automatically passed the instance of that class

printAllActivities(<Student>)

Constructor Reference

Constructors can also represent a lambda

Constructor are expected to return a instance

---

default methods

Custom class , we need to have a custom comparision for sorting

Pre JDK 8 :

create an implementation of Comparator, inject that comparator instance

JDK 8 : special comparator default method that can allow to inject comparision criteria

(Functional) Interface are more like Abstract Classes

direct instance related feature could not be defined

==> Multiple Inheritance in interface

JDK 8 allows to multiple inherit functionalities

Stream

Lambdas and Local Variables

Local Variable : declared inside any method

Lambdas have some restrictions on using local variables

=> Lambda expressions are not allowed to use the same name as

local variable as param or even inside lambda body for redeclaration

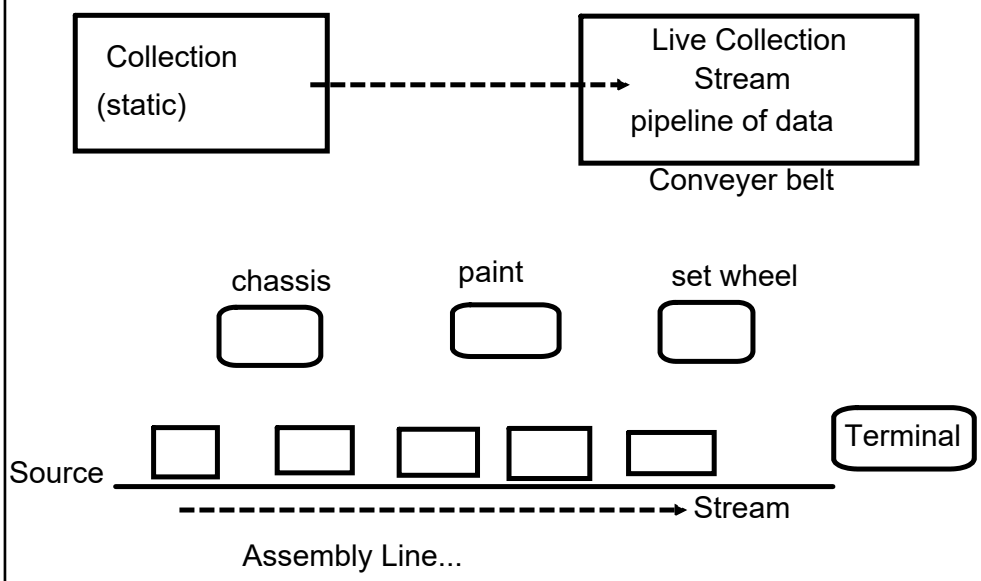
but can use it

(No restriction on class variables)

=> local variable are effectively final (no need to use final keyword)

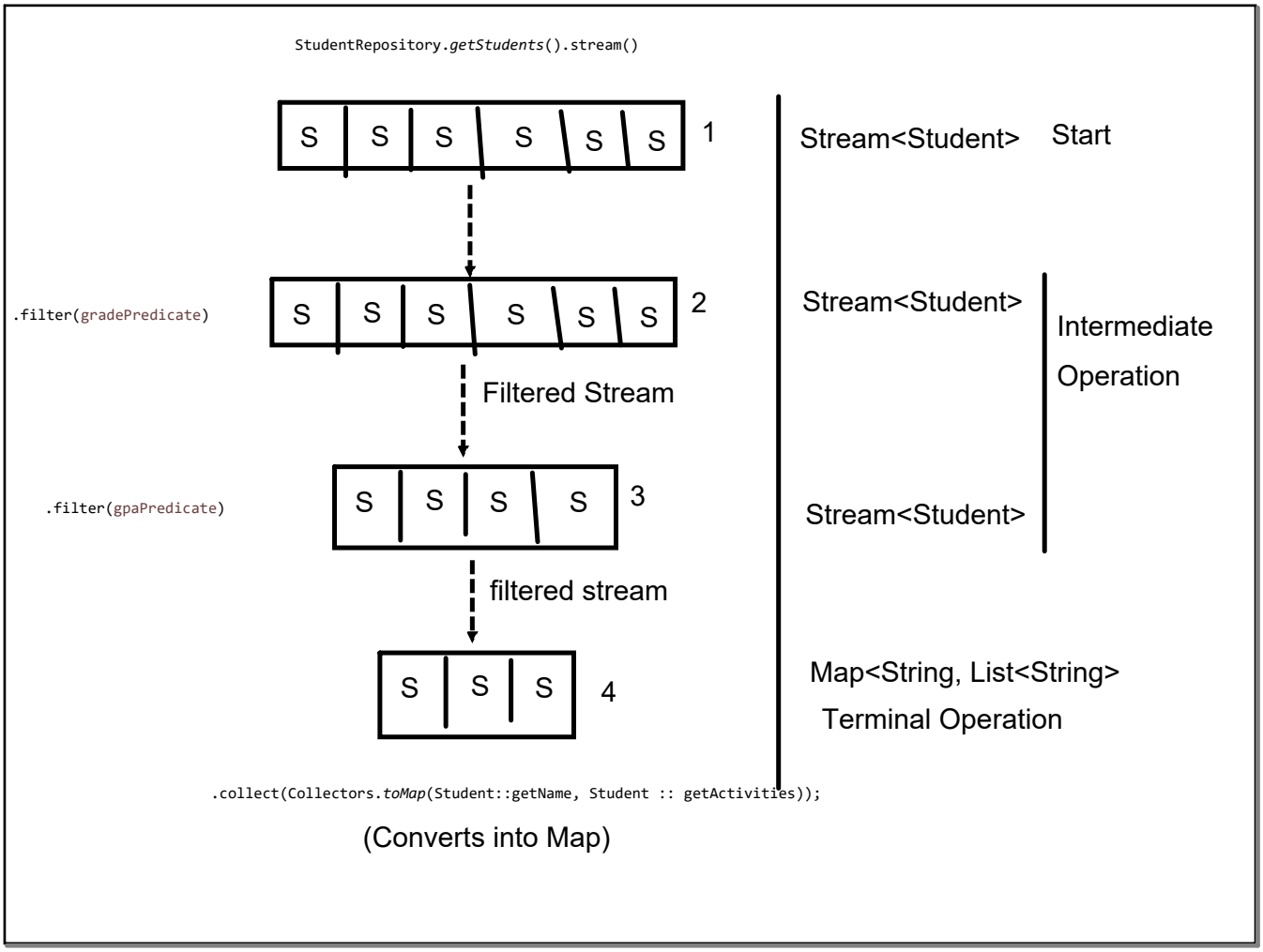
conditionally effectively final

Stream : can be created out of a Collection (warehouse), I/O



```
List<String> names = Arrays.asList('First', 'Second','Third');
names.stream(); // creates a stream
sequential or parallel
names.parallelStream(); // creates a Parallel stream
```

Collection	Stream
Add or modify any element of collection	Works on immutable/ fixed set of data
Elements can be accessed in any order	Elements can be accessed in a sequence
Collections can be traversed n number of times	Streams can traversed only once (one set of activities at a time)
External iterations	Internal Iteration
Collections are eagerly constructed	Streams are lazily constructed (will going to take place only if terminal operation ia available)


StudentRepository.getStudents().stream()
S S S S S S
1
Stream<Student>
Start
.filter(gradePredicate)
S S S S S S
2
Stream<Student>
Intermediate
Operation
Filtered Stream
.filter(gpaPredicate)
S S S S
3
Stream<Student>
filtered stream
S S S
4
Map<String, List<String>
Terminal Operation
.collect(Collectors.toMap(Student::getName, Student :: getActivities));
(Converts into Map)

Stream methods

=>Intermediatory Operation

=>Terminal Operation

Debug the stream : look into conveyer belt : peek()

Stream API : map()

Convert/Transform

flatMap():

like map : transform

used in context where each element in the stream represents multiple elements

Eg:

Stream<List>

Stream<Array>

Flatten the stream (non-nested stream) and transform

distinct() : Returns a stream with unique elements

count() : Returns a long: total no of elements in Stream (terminal activity)