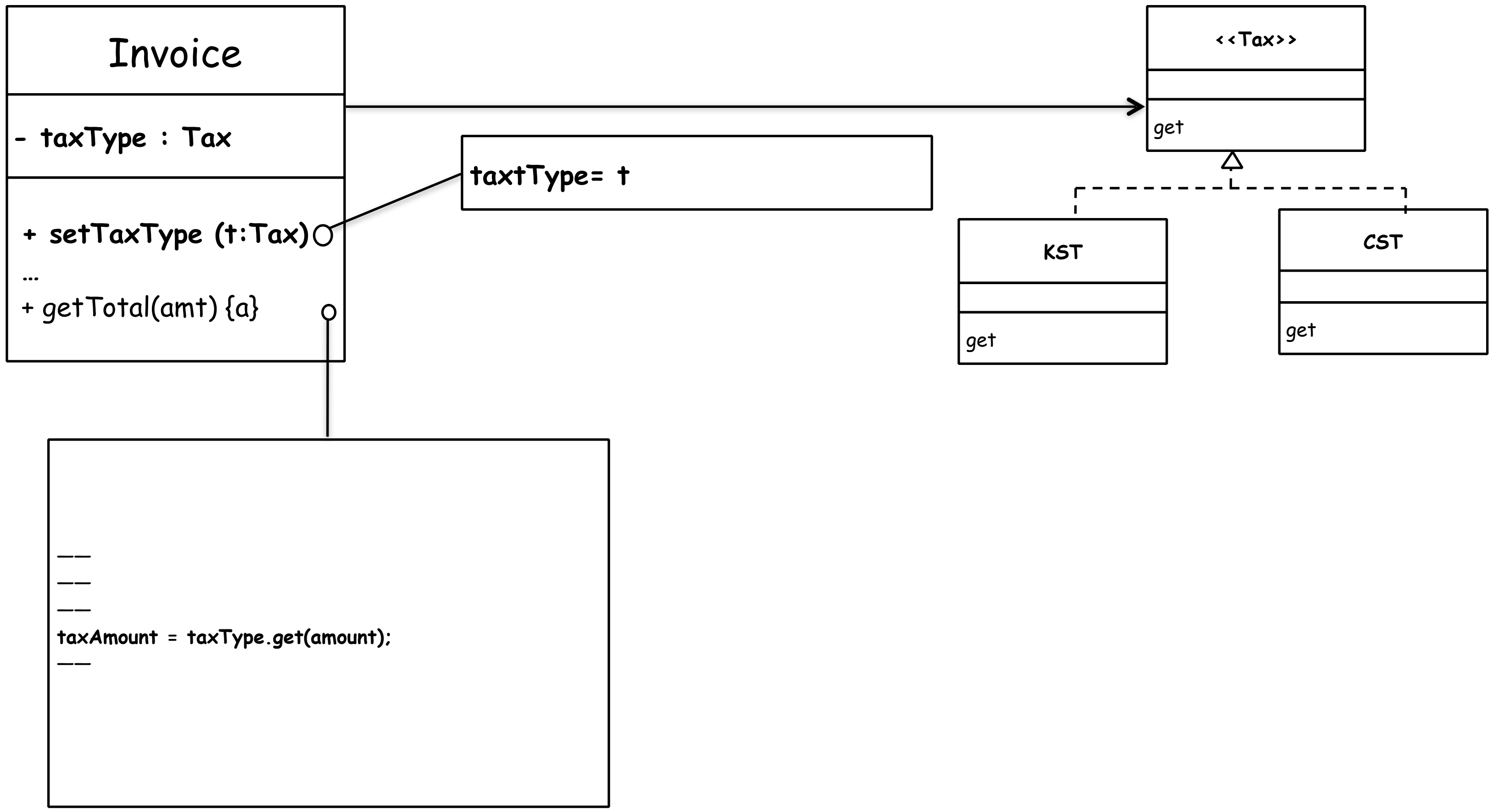
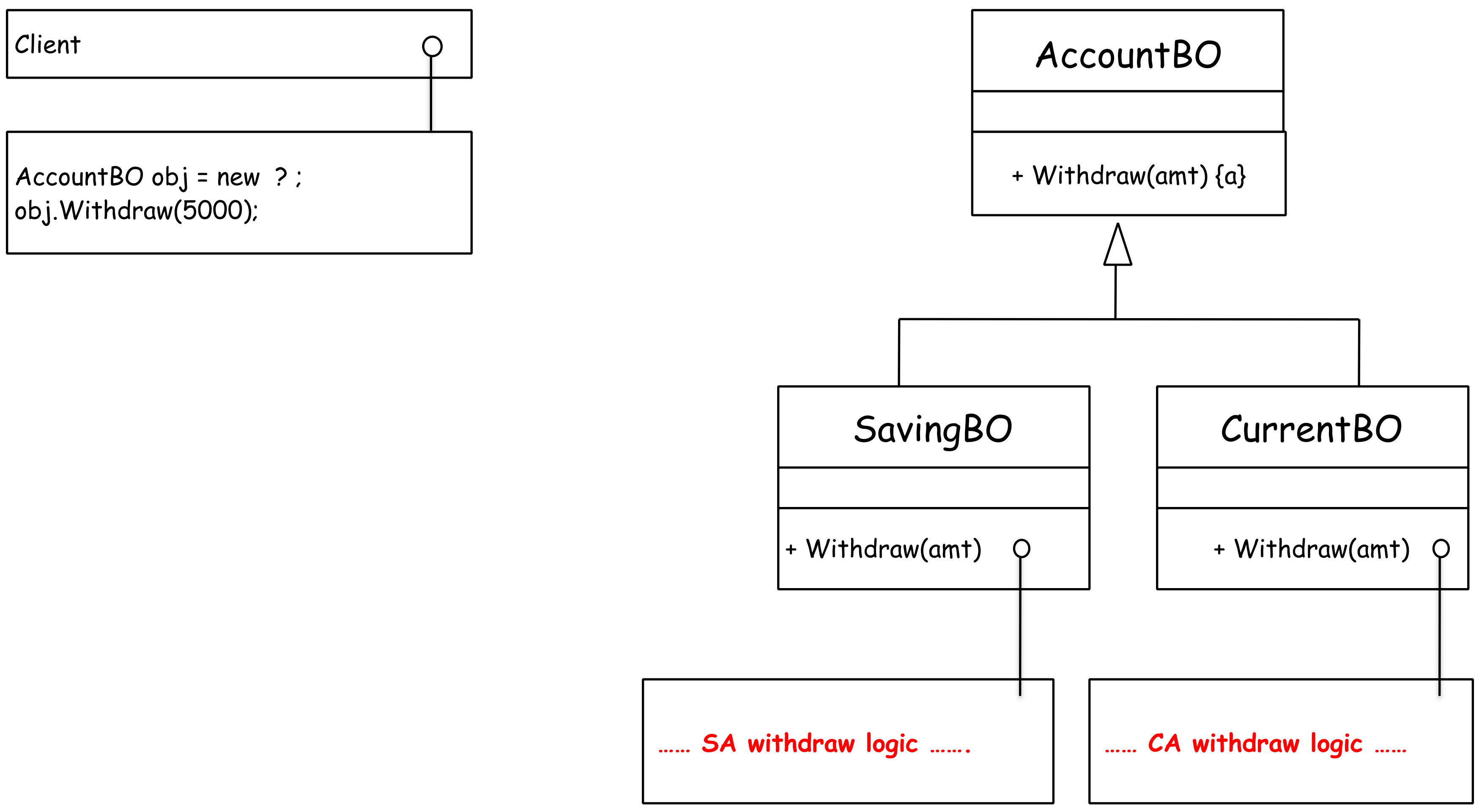
Declarative Programming


Invoice
- taxType : Tax
+ setTaxType (t:Tax)
...
+ getTotal(amt) {a}
taxtType= t
<<Tax>>
get
KST
get
CST
get
——
——
——
taxAmount = taxType.get(amount);
——

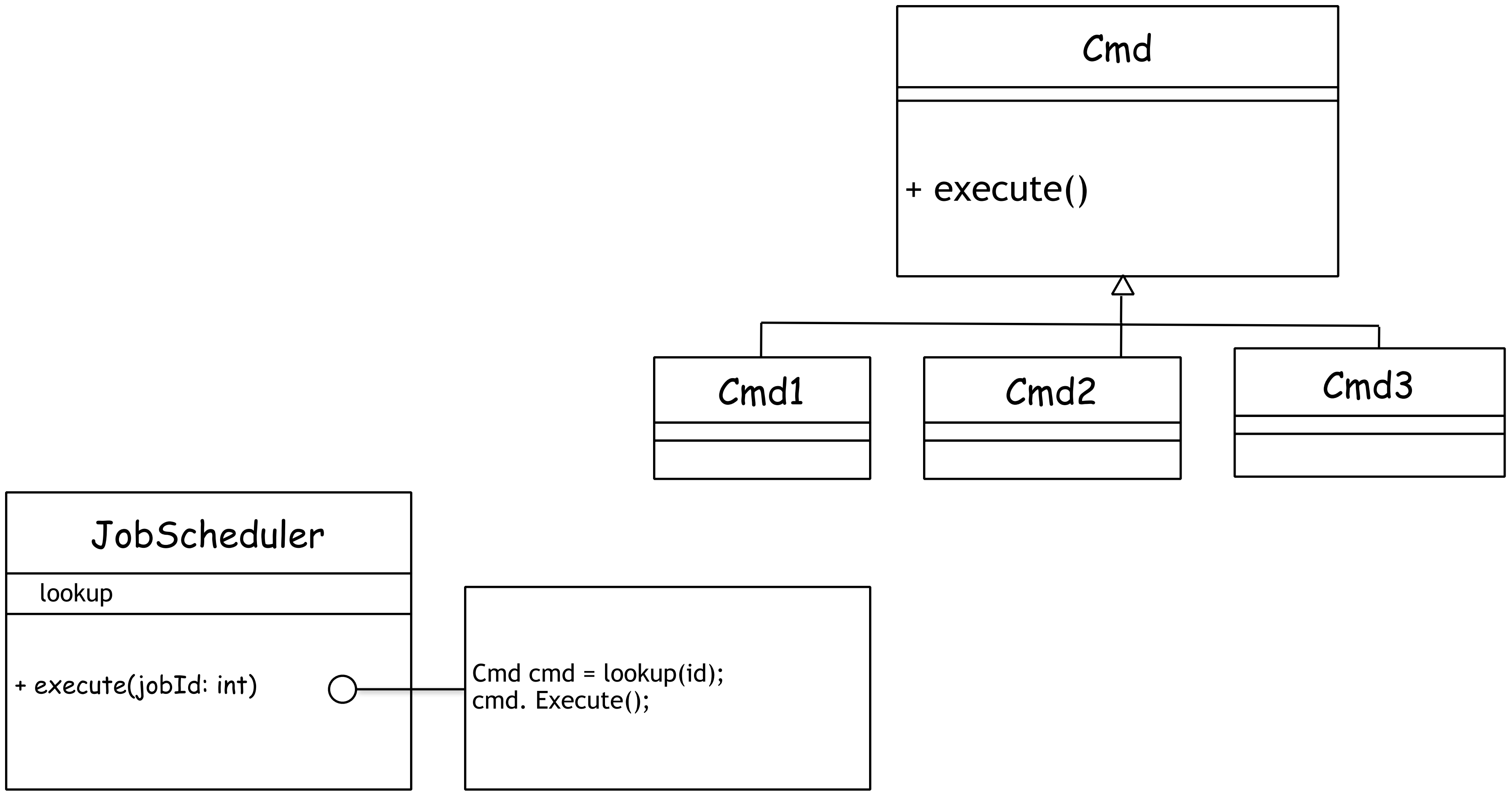
Avoid Conditional Code

```
interface Currency {
            public String format(int amount);
}

class USDCurrency implements Currency {
            public String format(int amount) {
                        //return something like $1,200
            }
}

class RMBCurrency implements Currency {
            public String format(int amount) {
                        //return something like RMB1,200
            }
}

class ESCUDOCurrency implements Currency {
            public String format(int amount) {
                        //return something like $1.200
            }
}
```

Abandon RTTI



use lookup



```
Cmd cmd = lookup(id);
cmd. Execute();
```

Procedure

```
var value = 0

func increment() -> Int {
 value += 1
 return value
 }
```

Function

```
func increment(value: Int) -> Int {
 return value + 1
}
```

Stateful

```
var value = 0

func increment() -> Int {
 value += 1
 return value
 }
```

Stateless

```
func increment(value: Int) -> Int {
 return value + 1
}
```

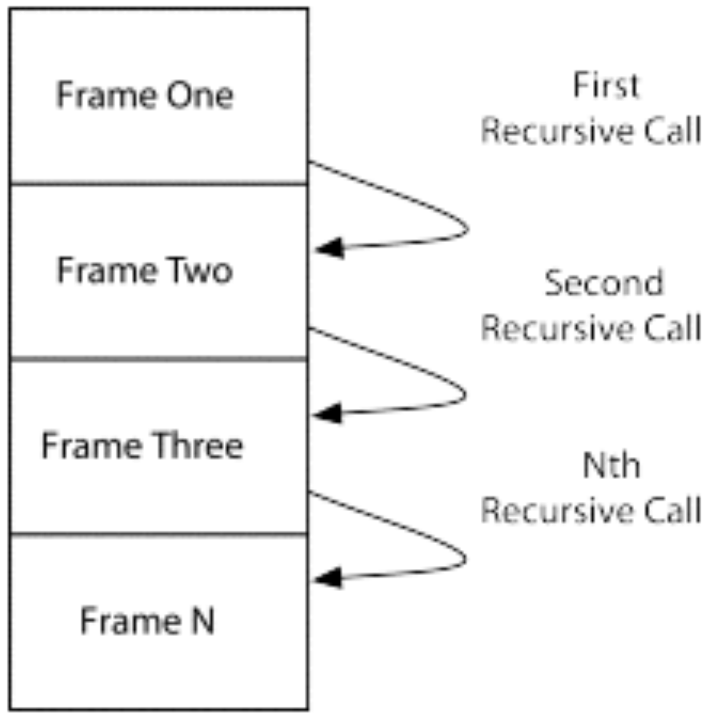
Stateful vs Stateless

```
class Math
{
        public int Add(int x,int y)
        {
                return x+ y;
        }
        public int Sub(int x,int y)
        {
                return x- y;
        }
}
```

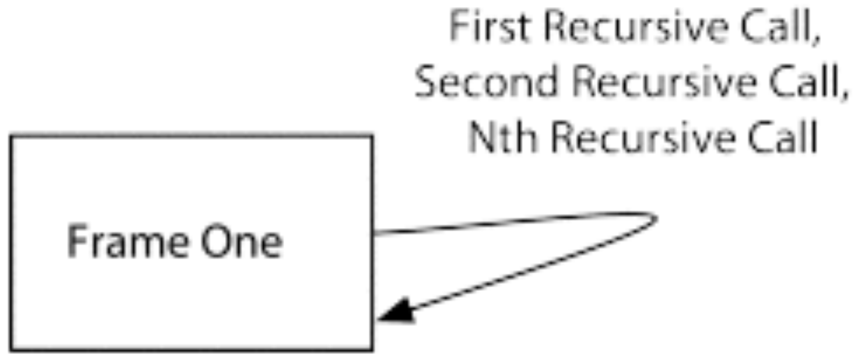
```
class Account
{
        double balance;

        public void withdraw(double amt)
        {
                balance -= amt;;
        }
        public void deposit(int amt)
        {
                balance +=  amt;
        }
}
```

```
#Iteration is an imperative technique that requires mutable state
# requires both i and sum to be mutable


def sum(upTo):
    sum = 0
    for i in range(0,upTo):
        sum += i
    return sum;


res = sum(10)
print(res)
```

```
#Since the functional world emphasizes immutability, iteration is out. In its
#place, we can use recursion, which does not require immutability

def sum(upTo):
    if (upTo == 0):
        return 0
    else:
        return upTo + sum(upTo - 1)

res = sum(10)
print(res)
```



```
# make sure that the recursive call was the last thing that happens
#in each branch of the function, known as the tail position


def sum(upTo, currentSum):
    if (upTo == 0):
        return currentSum
    else:
        return sum(upTo -1, currentSum+ upTo)


res = sum(10,0)
print(res)
```



```
import types


def countdown(start):
    print (start)
    if start == 0:
        yield 0
    else:
        yield countdown(start - 1)


def tramp(gen, *args, **kwargs):
    g = gen(*args, **kwargs)
    while isinstance(g, types.GeneratorType):
        g=g.__next__()
    return g


tramp(countdown, 1000)
```

pure functions

A function is pure if it meets two qualifications:

1. ”referential transparency”

2. It doesn't cause any side effects

Stateful vs Stateless

```
class Math
{
        public int Add(int x,int y)
        {
                return x+ y;
        }
        public int Sub(int x,int y)
        {
                return x- y;
        }
}
```

```
class Account
{
        double balance;

        public void withdraw(double amt)
        {
                balance -= amt;;
        }
        public void deposit(int amt)
        {
                balance +=  amt;
        }
}
```

	Stateful	stateless
thread friendly		
cache		
unit testable		
referential transparency		
Side Effects		
Immutable		
Pure Function		
Idempotent		
temporal coupling		

Side Effects

when a function changes the state of the system

- modifying a global variable or static variable
- modifying one of the functions arguments
- raising an exception
- writing data to a display or file
- reading data from a file or database
- calling other side-effecting functions

- A necessary evil: they can surprise us
  - and break the Least Astonishment Principle

RT

```
String x = "purple";
String r1 = x.replace('p', 't');
String r2 = x.replace('p', 't');

String r1 = "purple".replace('p', 't');
    r1: "turtle"
String r2 = "purple".replace('p', 't');
    r2: "turtle"
```

Non-RT

```
StringBuilder x = new StringBuilder("Hi");
StringBuilder y = x.append(", mom");
String r1 = y.toString();
String r2 = y.toString();

String r1 = x.append(", mom").toString();
    r1: "Hi, mom"
String r2 = x.append(", mom").toString();
    r1: "Hi, mom, mom"
```

means that an expression always evaluates to the same result in any context.

A pure function is a function without side-effects where the output is solely determined by the input

An idempotent function is one that can be applied multiple times without changing the result -

A function can be pure, idempotent, both, or neither.

Temporal Coupling

- A consequence of side effects
  - e.g. Open the DB connection → do stuff → close DB connection
- Leads to mysterious code
  - *“Hack!! No one told me that I should have called `.initialize()` first !!"*

principle of least astonishment

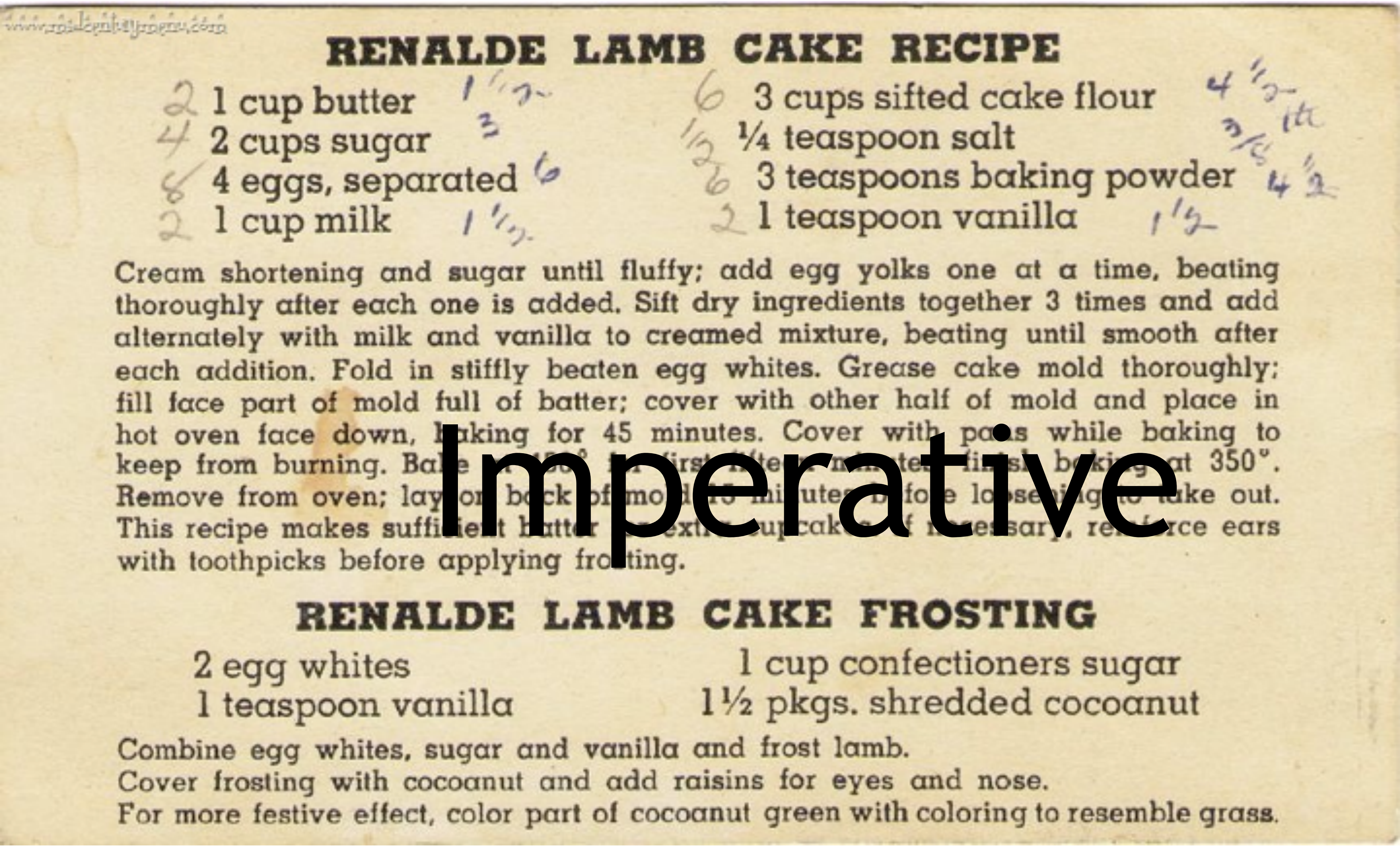
the name of a function should reflect what it does result of performing some operation should be obvious, consistent, and predictable, based upon the name of the operation

External Iterator

```
for(Person p: persons)
{
        p.setLastName("Doe")
}
```

Internal Iterator

```
List<Person> persons = asList(new Person("Joe"), new Person("Jim"),
new Person(“John"));

persons.forEach(p -> p.setLastName("Doe"))
```



Imperative

RENALDE LAMB CAKE RECIPE

2 1 cup butter 1½ — 6 3 cups sifted cake flour 4½
4 2 cups sugar 3 — ½ ¼ teaspoon salt 3/8 tt
8 4 eggs, separated 6 — 6 3 teaspoons baking powder 4½
2 1 cup milk 1½ — 2 1 teaspoon vanilla 1½

Cream shortening and sugar until fluffy; add egg yolks one at a time, beating thoroughly after each one is added. Sift dry ingredients together 3 times and add alternately with milk and vanilla to creamed mixture, beating until smooth after each addition. Fold in stiffly beaten egg whites. Grease cake mold thoroughly; fill face part of mold full of batter; cover with other half of mold and place in hot oven face down, baking for 45 minutes. Cover with pans while baking to keep from burning. Bake at 450° for first fifteen minutes, finish baking at 350°. Remove from oven; lay on back of mold 5 minutes before loosening to take out. This recipe makes sufficient batter for extra cupcakes. If necessary, reinforce ears with toothpicks before applying frosting.

RENALDE LAMB CAKE FROSTING

2 egg whites — 1 cup confectioners sugar
1 teaspoon vanilla — 1½ pkgs. shredded cocoanut

Combine egg whites, sugar and vanilla and frost lamb.
Cover frosting with cocoanut and add raisins for eyes and nose.
For more festive effect, color part of cocoanut green with coloring to resemble grass.

Imperative

```
let array = [0, 1, 2, 3, 4, 5]

var evenNumbers = [Int]()

for element in array {
 if element % 2 == 0 {
 evenNumbers.append(element)
 }
}
```

Declarative

Declarative

```
let array = [0, 1, 2, 3, 4, 5]
let evenNumbers = array.filter { $0 % 2 == 0 }
```

```
var data = [1, 2, 3, 4, 5];

var numbers = data.map(function (nr) {
    return nr + 1;
});

// numbers = [2, 3, 4, 5, 6]
```

```
var data = [1, 2, 3, 4, 5, 6, 7];

var numbers = data.filter(function (nr) {
    return nr % 2 === 0;
});

// numbers = [2, 4, 6]
```

```
var data = [1, 2, 3, 4, 5, 6, 7];

var numbers = data.map(function (nr) {
    return nr + 1;
}).filter(function (nr) {
    return nr % 2 === 0;
});

// numbers = [2, 4, 6, 8]
```

```
var data = [[1, 2], [3, 4], 5, [6], 7, 8];

var numbers = data.mergeAll();

// numbers = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
var data = [{
   numbers: [1, 2]
}, {
   numbers: [3, 4]
}];
var numbersFlatMap = data.flatMap(function (object) {
      return object.numbers;
});
// numbersMap = [[1, 2], [3, 4]]
// numbersFlatMap = [1, 2, 3, 4]
```

```
var data = [1, 2, 3, 4];

var sum = data.reduce(function(acc, value)
{
    return acc + value;
});

// sum = 10
```

```
var data = [5, 7, 3, 4];

var min = data.reduce(function(acc, value)
{
    return acc < value ? acc : value;
});

// min = 3
```

```
var array1 = [1, 2, 3];
var array2 = [4, 5, 6];

var array = Array.zip(array1, array2,
    function(left, right) {
        return [left, right];
});

// array = [[1, 4], [2, 5], [3, 6]]
```

```
public Article getFirstJavaArticle() {
    for (Article article : articles) {
        if (article.getTags().contains("Java")) {
            return article;
        }
    }
    return null;
}
```

```
public Optional<Article> getFirstJavaArticle() {
    return articles.stream()
        .filter(article -> article.getTags().contains("Java"))
        .findFirst();
    }
```

```
public List<Article> getAllJavaArticles() {

 List<Article> result = new ArrayList<>();

 for (Article article : articles) {
    if (article.getTags().contains("Java")) {
        result.add(article);
    }
 }
```

```
public List<Article> getAllJavaArticles() {
   return articles.stream()
       .filter(article -> article.getTags().contains("Java"))
       .collect(Collectors.toList());
   }
```

```
public Map<String, List<Article>> groupByAuthor() {

  Map<String, List<Article>> result = new HashMap<>();

  for (Article article : articles) {
     if (result.containsKey(article.getAuthor())) {
        result.get(article.getAuthor()).add(article);
     } else {
        ArrayList<Article> articles = new ArrayList<>();
        articles.add(article);
        result.put(article.getAuthor(), articles);
     }
  }
 return result;
}
```

```
public Map<String, List<Article>> groupByAuthor() {
   return articles.stream()
      .collect(Collectors.groupingBy(Article::getAuthor));
}
```

```
public Set<String> getDistinctTags() {

  Set<String> result = new HashSet<>();

  for (Article article : articles) {
      result.addAll(article.getTags());
  }

  return result;
}
```

```
public Set<String> getDistinctTags() {
    return articles.stream()
        .flatMap(article -> article.getTags().stream())
        .collect(Collectors.toSet());
}
```

Imperative Style Code

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class ImperativeStyleDemo
{
    public static void main(String[] args)
    {
        List numbers = Arrays.asList(1, 2, 3, 4, 5);

        List modifiedNumbers = new ArrayList<>();
        for (Integer number: numbers)
        {
            if (isEven(number))
                modifiedNumbers.add(ImperativeStyleDemo.doubleInteger(number));
        }
        for (String number: modifiedNumbers){
            System.out.println(number);
        }
    }
    public static String doubleInteger(Integer number)
    {
        return String.valueOf(number * 2);
    }
    public static Boolean isEven(Integer number)
    {
        return number % 2 == 0;
    }
}
```

Declarative Style Code

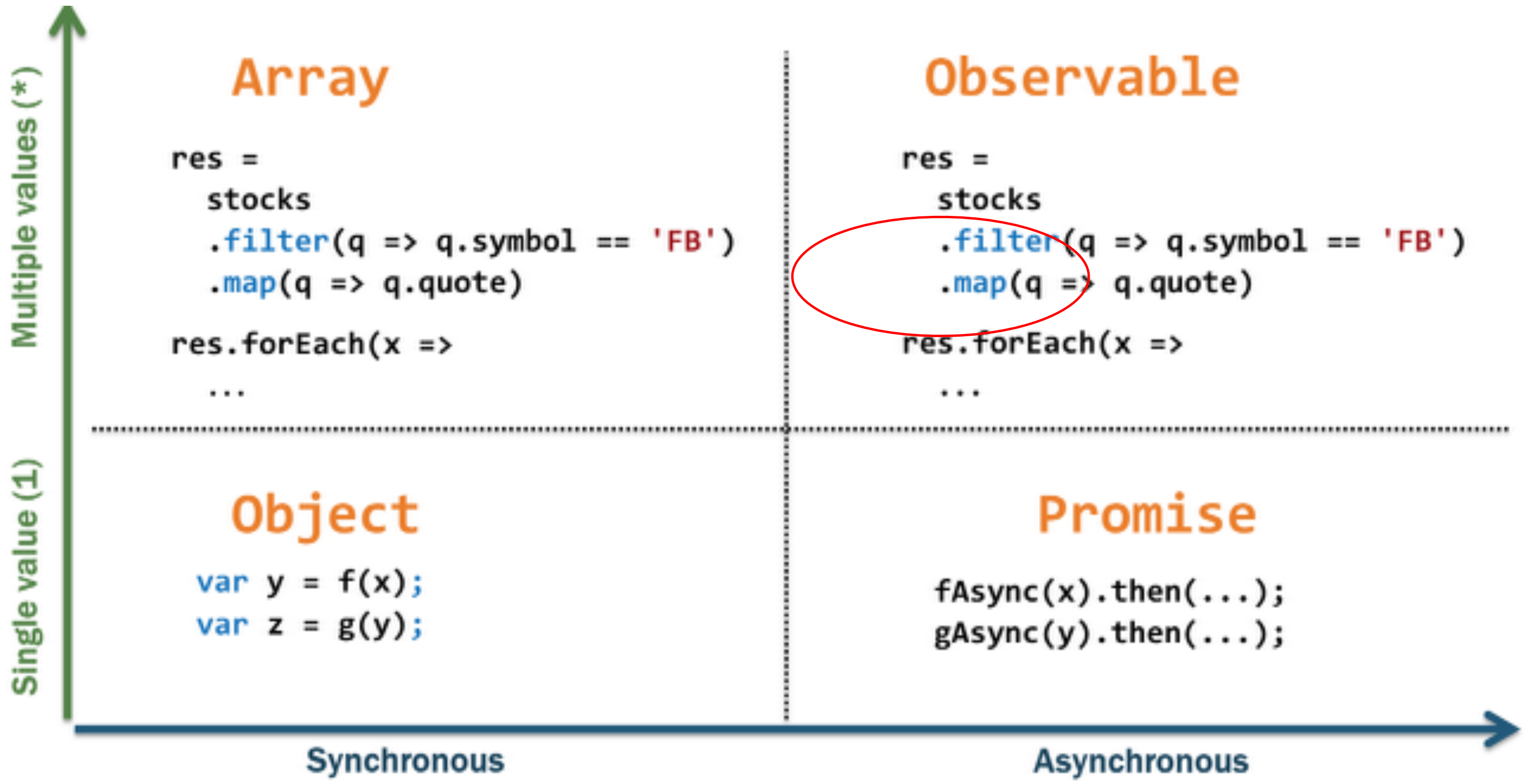
```
import java.util.Arrays;
import java.util.List;

class DeclarativeStyle
{
    public static void main(String[] args)
    {
        List numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

        numbers.stream()                          //A Fancy Collection
           .filter(DeclarativeStyle::isEven)      //Filter the data
           .map(DeclarativeStyle::doubleInteger) //Map new values
           .forEach(System.out::println);         //Iterate & perform the operation
    }
    public static String doubleInteger(Integer number)
    {
        return String.valueOf(number * 2);
    }
    public static Boolean isEven(Integer number)
    {
        return number % 2 == 0;
    }
}
```

Imperative Style Code

```
List<Transaction> groceryTransactions = new Arraylist<>();
for(Transaction t: transactions){
       if(t.getType() == Transaction.GROCERY)
       {
           groceryTransactions.add(t);
       }
}
Collections.sort(groceryTransactions, new Comparator()
{
       public int compare(Transaction t1, Transaction t2)
       {
           return t2.getValue().compareTo(t1.getValue());
       }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions)
{
              transactionsIds.add(t.getId());
}
```

Declarative Style Code

```
List<Integer> transactionsIds =
    transactions.stream()
            .filter(t->t.getType() == Transaction.GROCERY)
            .sorted(comparing(Transaction::getValue).reversed())
            .map(Transaction::getId)
            .collect(toList());
```



```
res =
  stocks
  .filter(q => q.symbol == 'FB')
  .map(q => q.quote)
res.forEach(x =>
  ...
```

```
res =
  stocks
  .filter(q => q.symbol == 'FB')
  .map(q => q.quote)
res.forEach(x =>
  ...
```

```
var y = f(x);
var z = g(y);
```

```
fAsync(x).then(...);
gAsync(y).then(...);
```

- Asynchronous data streams
- **Everything** is a stream

Why Rx

```
public List<Todo> getTodos() {
    List<Todo> todosFromWeb = // query a webservice
                                        (with bad network latency)
    return todosFromWeb;
}
```

non blocking call

```
public void getTodos(Consumer<List<Todo>>
todosCallback) {

    Thread thread = new Thread(()-> {
        List<Todo> todosFromWeb = // query a webservice

        todosCallback.accept(todosFromWeb);
    });
    thread.start();
}
```

how to handle errors ?

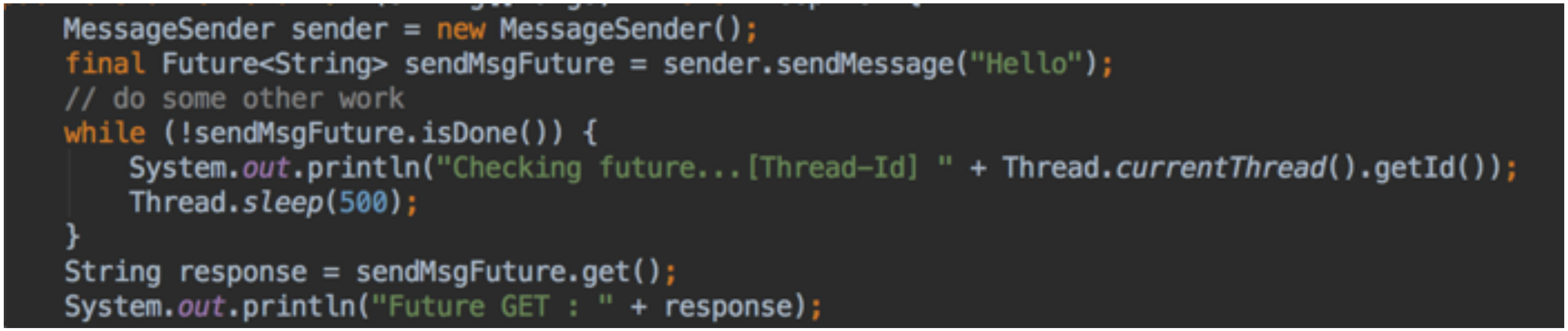
```
public void getTodos(FailableCallback<List<Todo>>
todosCallback) {

    Thread thread = new Thread(()-> {
        try {
            List<Todo> todosFromWeb = // query a service

            todosCallback.accept(todosFromWeb);
        } catch(Exception ex) {
            todosCallback.error(ex);
        }
    });
    thread.start();
}
```

one service depends on another

```
public void getTodos(FailableCallback<List<Todo>> todosCallback) {

  Thread thread = new Thread(()-> {
    getUserPermission(new FailableCallback() {

      public void accept(UserPermission permission) {
        if(permission.isValid()) {
          try {
            List<Todo> todosFromWeb = // query a web service

            if(!todosCallbackInstance.isDisposed()) {
              if(syncWithUIThread()) {
                todosCallback.accept(todosFromWeb);
              }
            }
          } catch(Exception ex) {
            if(!todosCallbackInstance.isDisposed()) {
              if(syncWithUIThread()) {
                todosCallback.error(ex);
              }
            }
          }
        }
      }

      public void error(Exception ex) {
        // Oh no!
      }
    });
  });
```

```
public void getUserPermission(FailableCallback<UserPermission> permissionCallback) {
  Thread thread = new Thread(()-> {
    try {
      UserPermission permission = // query a web service

      permissionCallback.accept(permission);
    } catch(Exception ex) {
      permission.error(ex);
    }
  });
  thread.start();
}
```

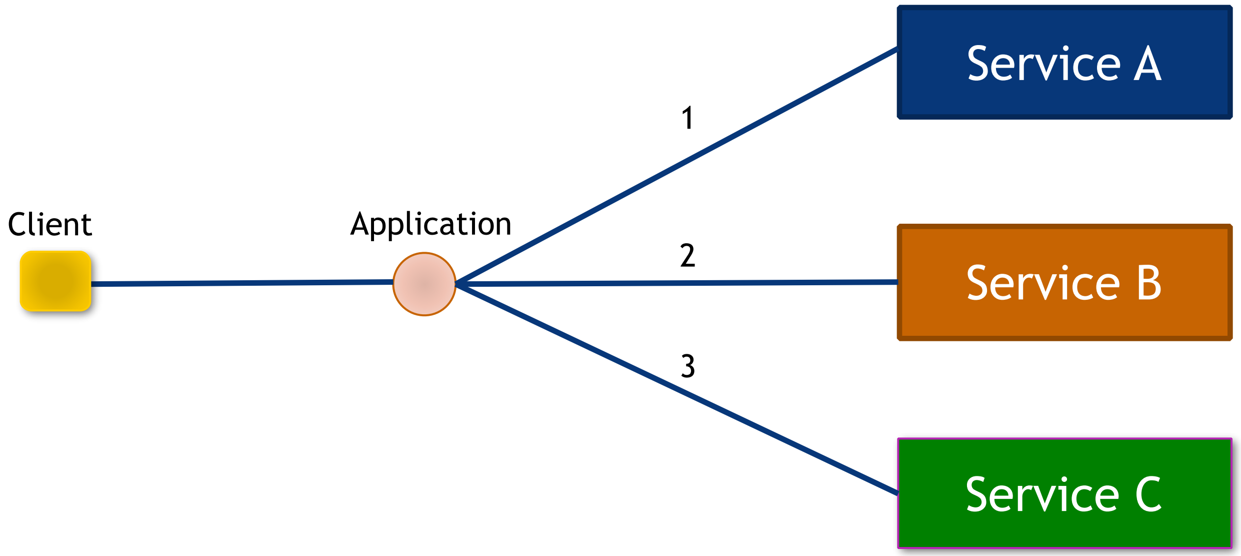
Java Futures



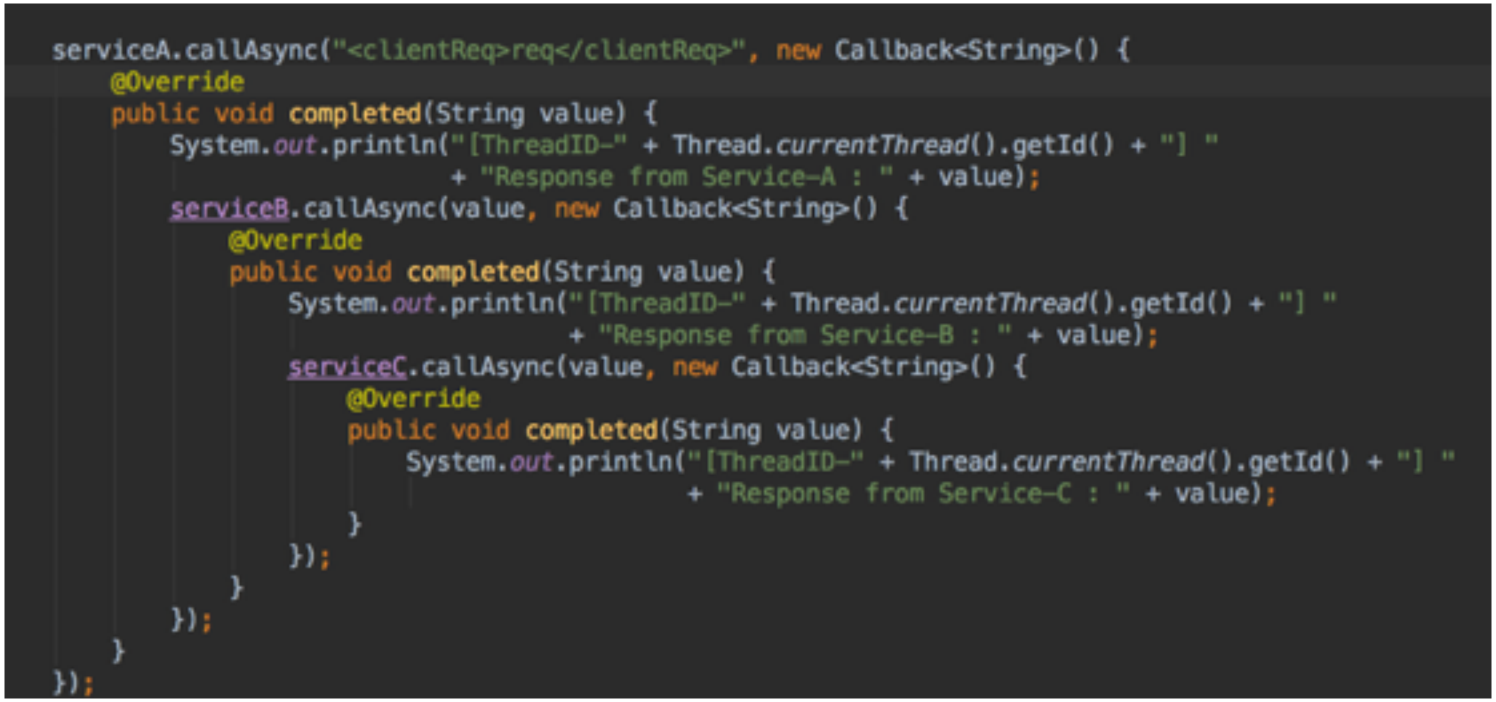
```
MessageSender sender = new MessageSender();
final Future<String> sendMsgFuture = sender.sendMessage("Hello");
// do some other work
while (!sendMsgFuture.isDone()) {
    System.out.println("Checking future...[Thread-Id] " + Thread.currentThread().getId());
    Thread.sleep(500);
}
String response = sendMsgFuture.get();
System.out.println("Future GET : " + response);
```

Callbacks

- Use case :
  - Client send a request to the Application
  - Application calls service A
  - Get the response from ServiceA and send it to service B
  - Get the response from ServiceB and send it to Service C
  - Get the response from ServiceC and send it to the Client



Callback Hell

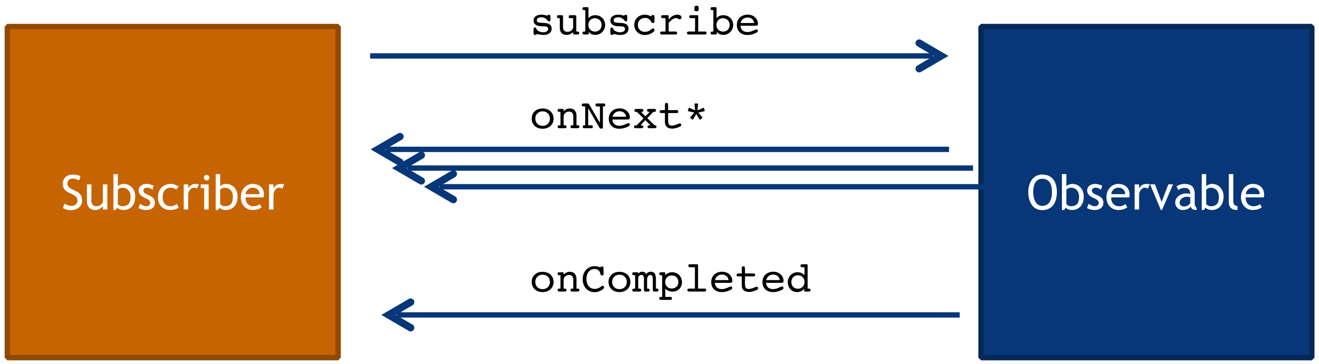


```
serviceA.callAsync("<clientReq>req</clientReq>", new Callback<String>() {
    @Override
    public void completed(String value) {
        System.out.println("[ThreadID-" + Thread.currentThread().getId() + "] "
                                + "Response from Service-A : " + value);
        serviceB.callAsync(value, new Callback<String>() {
            @Override
            public void completed(String value) {
                System.out.println("[ThreadID-" + Thread.currentThread().getId() + "] "
                                        + "Response from Service-B : " + value);
                serviceC.callAsync(value, new Callback<String>() {
                    @Override
                    public void completed(String value) {
                        System.out.println("[ThreadID-" + Thread.currentThread().getId() + "] "
                                                + "Response from Service-C : " + value);
                    }
                });
            }
        });
    }
});
```

Observable

- Rx-Java introduces ‘Observable’ data type by extending the traditional Observer Pattern.

Observable and Subscriber



```
Observable<Integer> source = Observable.range(1, 5);

Subscriber<Integer> consumer = new Subscriber<Integer>() {
    @Override
    public void onNext(Integer number){
            System.out.println(number); }
    @Override
    public void onError(Throwable e) {
            System.out.println("error"); }
    @Override
    public void onCompleted() {
            System.out.println("completed"); }
};

source.subscribe(consumer);
```

Observable Basics

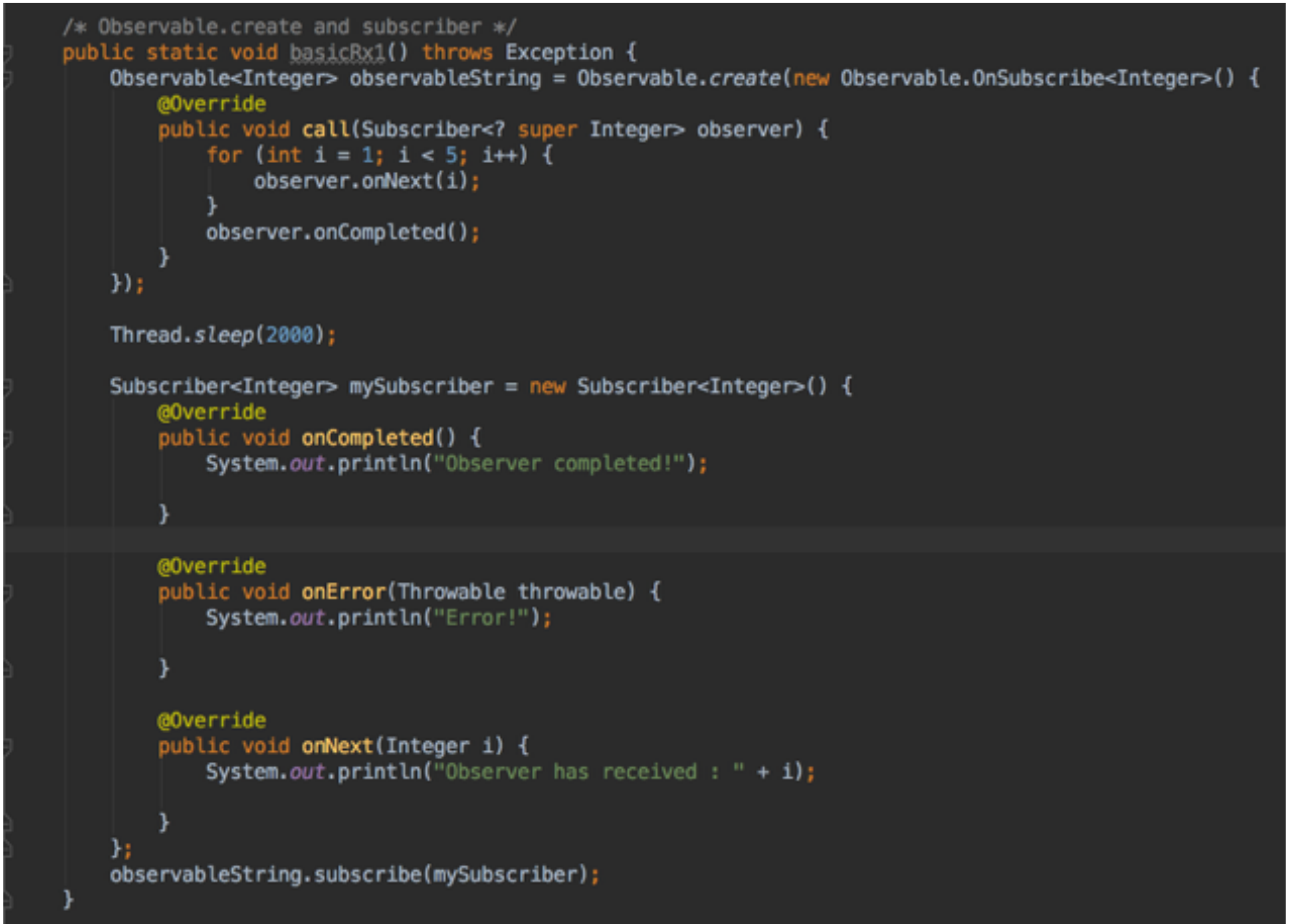
```
var range = Rx.Observable.range(1, 3); // 1, 2, 3

var range = range.subscribe(
      function(value) {},
      function(error) {},
      function() {}
);
```

optional

Observable Creation

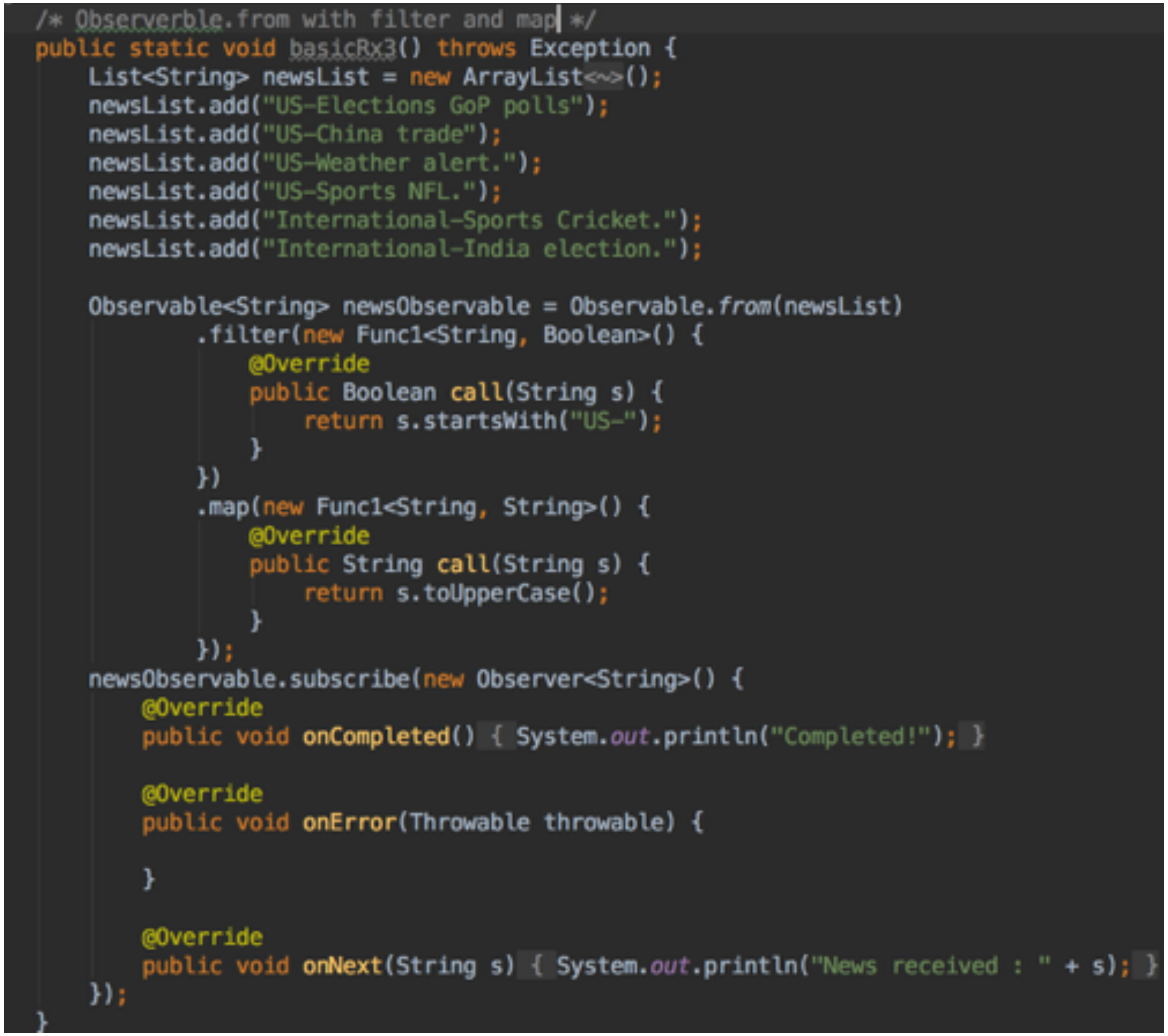
```
private static IObservable<string> GetProducts()
{
return Observable.Create<string>(o =>{
          using(var conn = new SqlConnection(connectionString))
          using (var cmd = new SqlCommand("…", conn))
          {
                   conn.Open();
                   SqlDataReader reader = cmd.ExecuteReader();
                   while (reader.Read())
                   {
                            o.OnNext(reader.GetString(0));
                   }
                   o.OnCompleted();
                   return Disposable.Create(()=>Console.WriteLine("--Disposed--"));
          }
          });
}

var products = GetProducts().Take(3);
products.Subscribe(Console.WriteLine);
```

Observable and Subscriber



```java
/* Observable.create and subscriber */
public static void basicRx1() throws Exception {
    Observable<Integer> observableString = Observable.create(new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> observer) {
            for (int i = 1; i < 5; i++) {
                observer.onNext(i);
            }
            observer.onCompleted();
        }
    });

    Thread.sleep(2000);

    Subscriber<Integer> mySubscriber = new Subscriber<Integer>() {
        @Override
        public void onCompleted() {
            System.out.println("Observer completed!");

        }

        @Override
        public void onError(Throwable throwable) {
            System.out.println("Error!");

        }

        @Override
        public void onNext(Integer i) {
            System.out.println("Observer has received : " + i);

        }
    };
    observableString.subscribe(mySubscriber);
}
```

Filtering and Transforming



```
/* Observerble.from with filter and map */
public static void basicRx3() throws Exception {
    List<String> newsList = new ArrayList<~>();
    newsList.add("US-Elections GoP polls");
    newsList.add("US-China trade");
    newsList.add("US-Weather alert.");
    newsList.add("US-Sports NFL.");
    newsList.add("International-Sports Cricket.");
    newsList.add("International-India election.");

    Observable<String> newsObservable = Observable.from(newsList)
            .filter(new Func1<String, Boolean>() {
                @Override
                public Boolean call(String s) {
                    return s.startsWith("US-");
                }
            })
            .map(new Func1<String, String>() {
                @Override
                public String call(String s) {
                    return s.toUpperCase();
                }
            });
    newsObservable.subscribe(new Observer<String>() {
        @Override
        public void onCompleted() { System.out.println("Completed!"); }

        @Override
        public void onError(Throwable throwable) {

        }

        @Override
        public void onNext(String s) { System.out.println("News received : " + s); }
    });
}
```

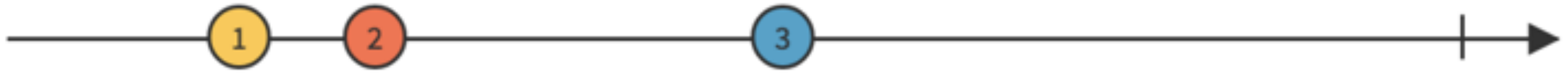
Iterable v/s Observable

EVENT	*ITERABLE (PULL)*	*OBSERVABLE (PUSH)*
retrieve data	*T next()*	*onNext(T)*
discover error	*throws Exception*	*onError(Exception)*
complete	*!hasNext()*	*onCompleted()*

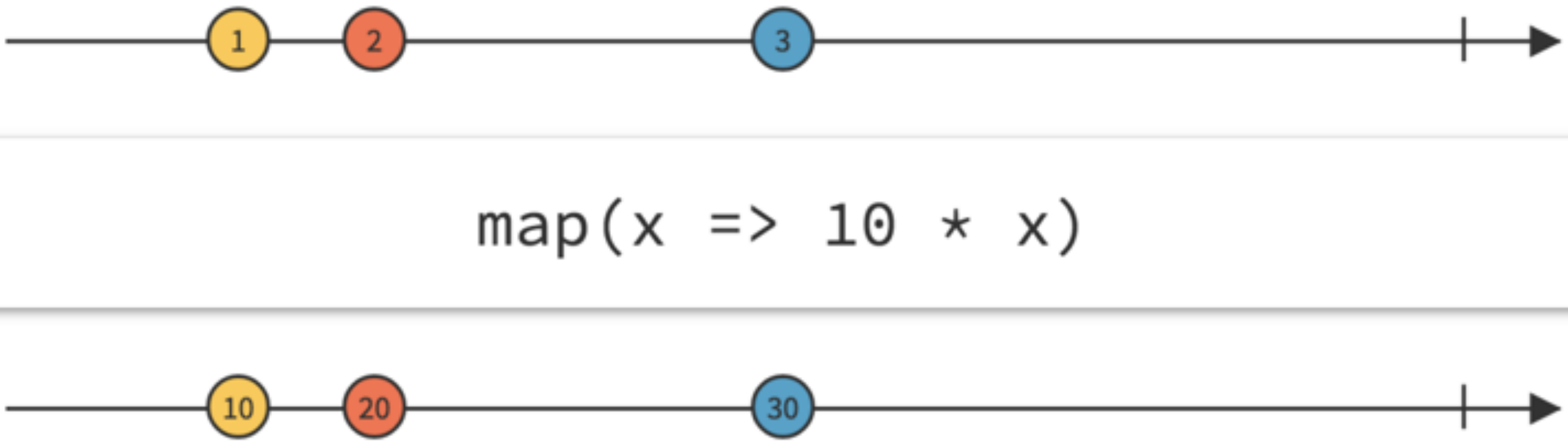
iterator v/s stream v/s Observable

Iterator	*stream*	*OBSERVABLE*
(PULL)	***(PULL)***	***(PUSH)***
External	***Internal***	***internal***

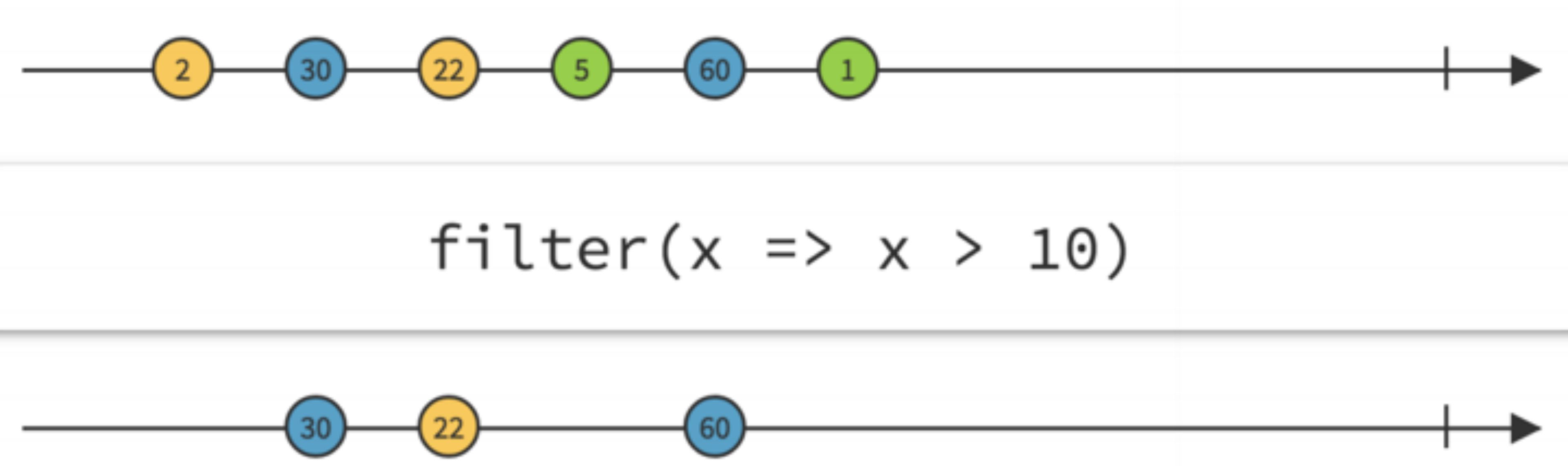
Stream



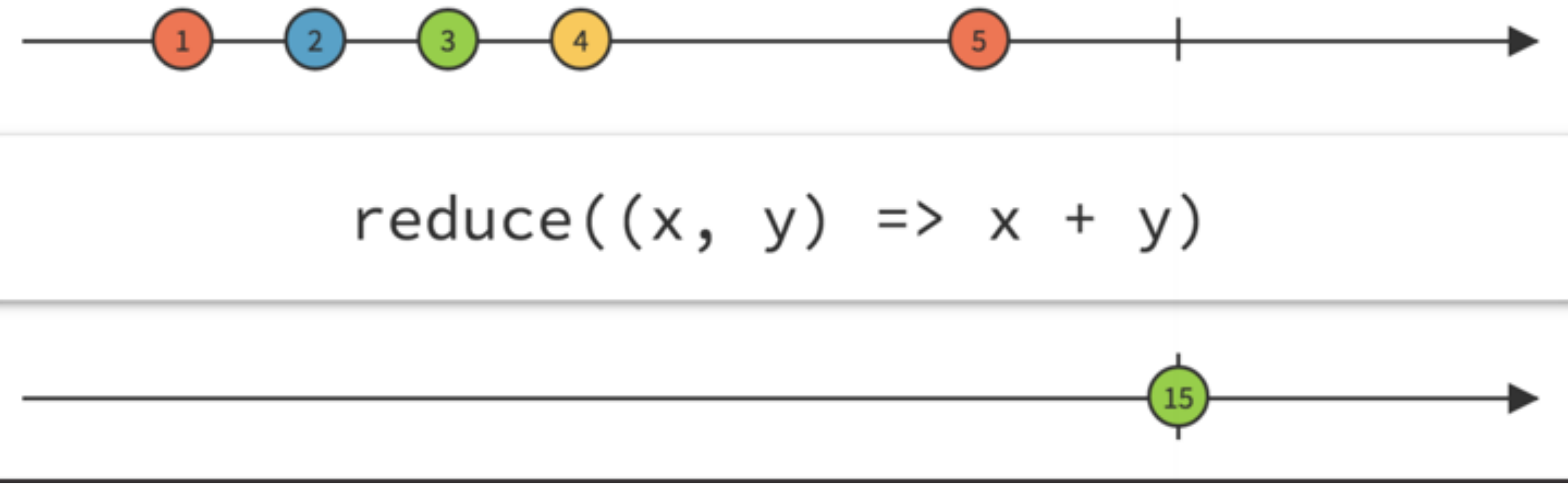
Map



filter

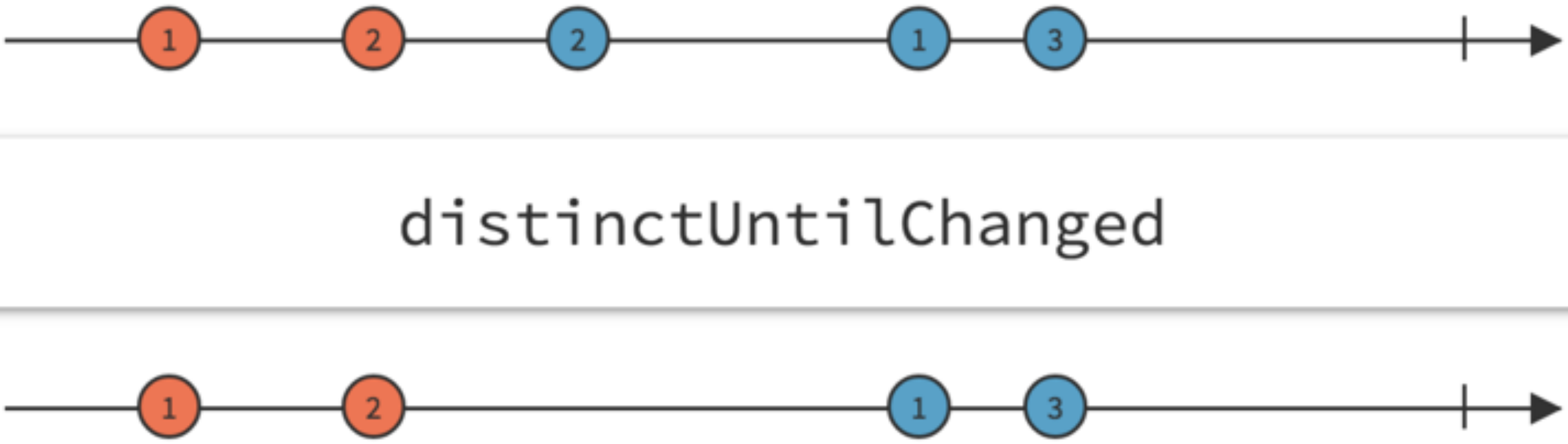


Aggregating

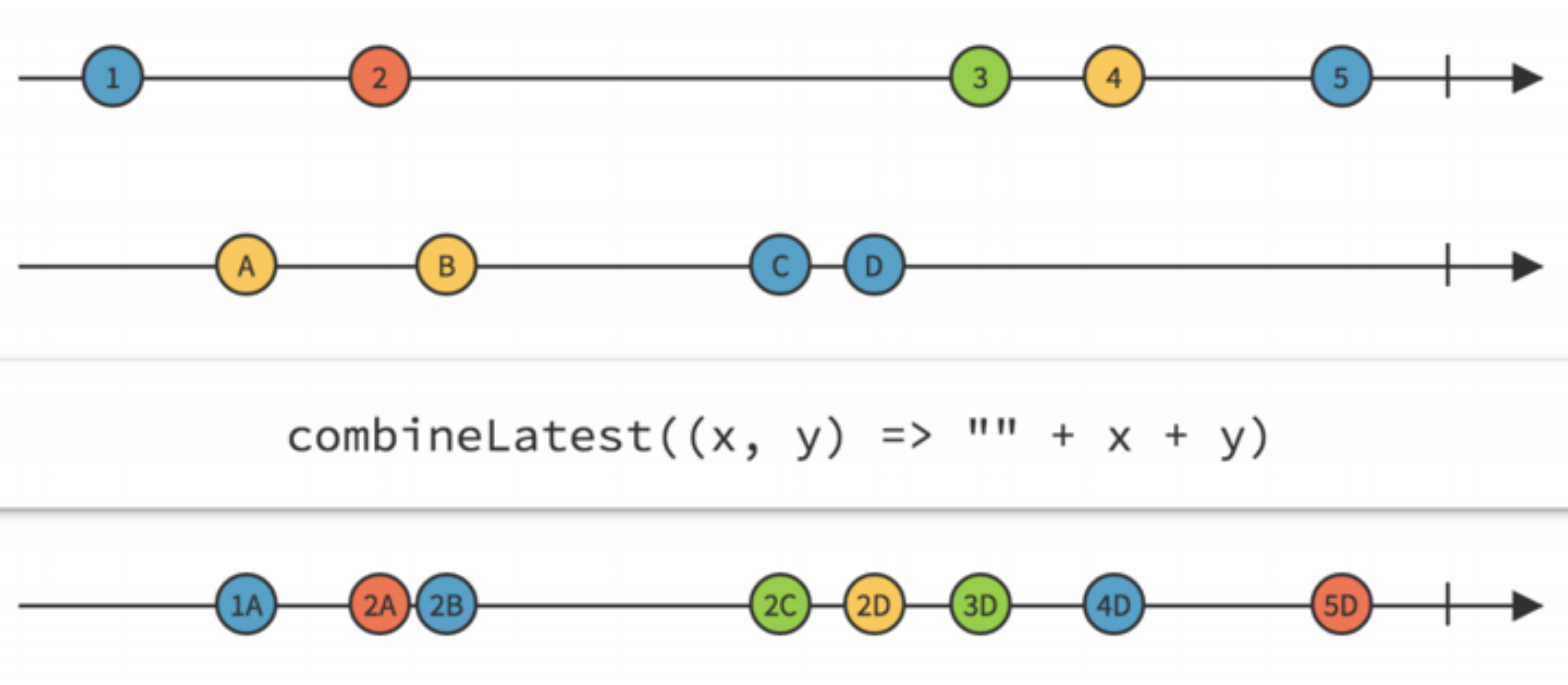


```
reduce((x, y) => x + y)
```

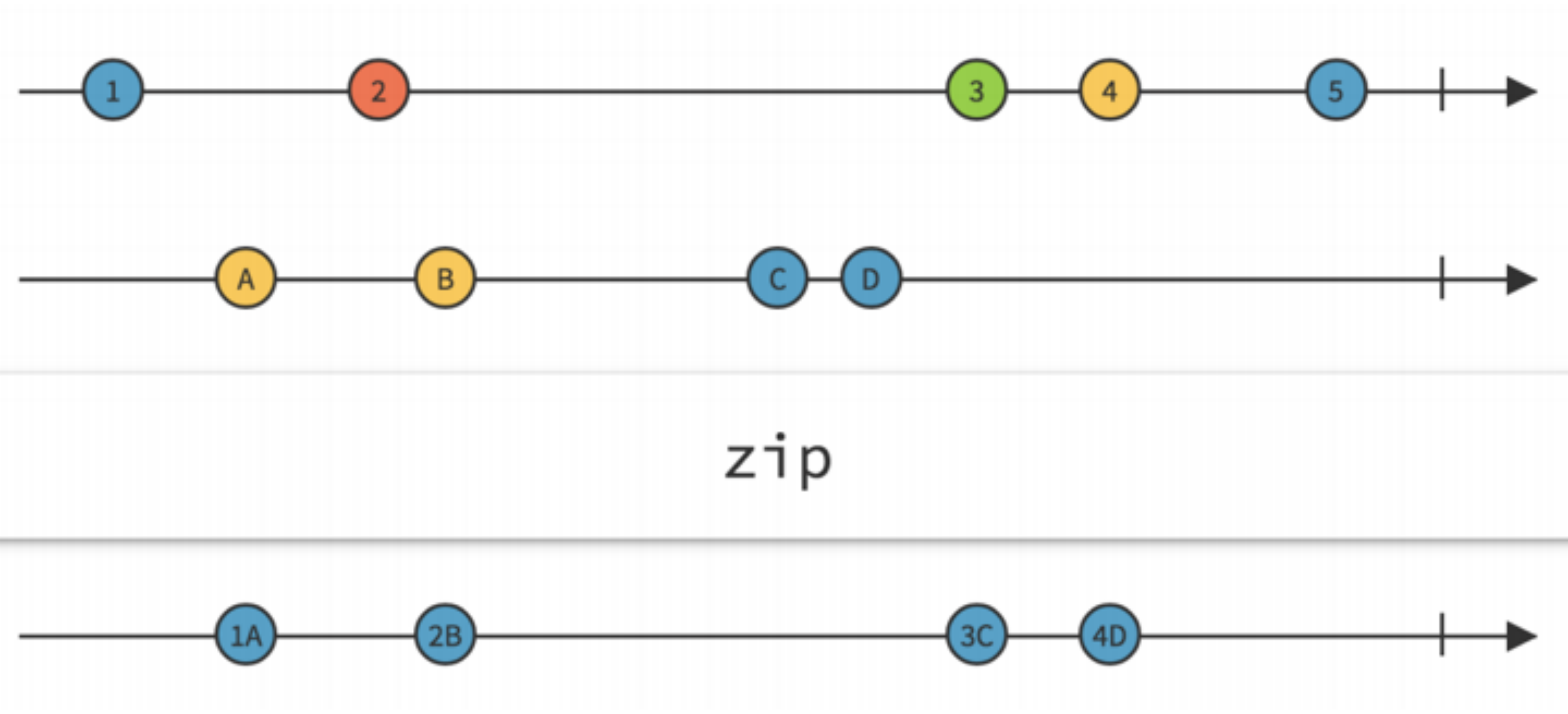
Skip Repeats



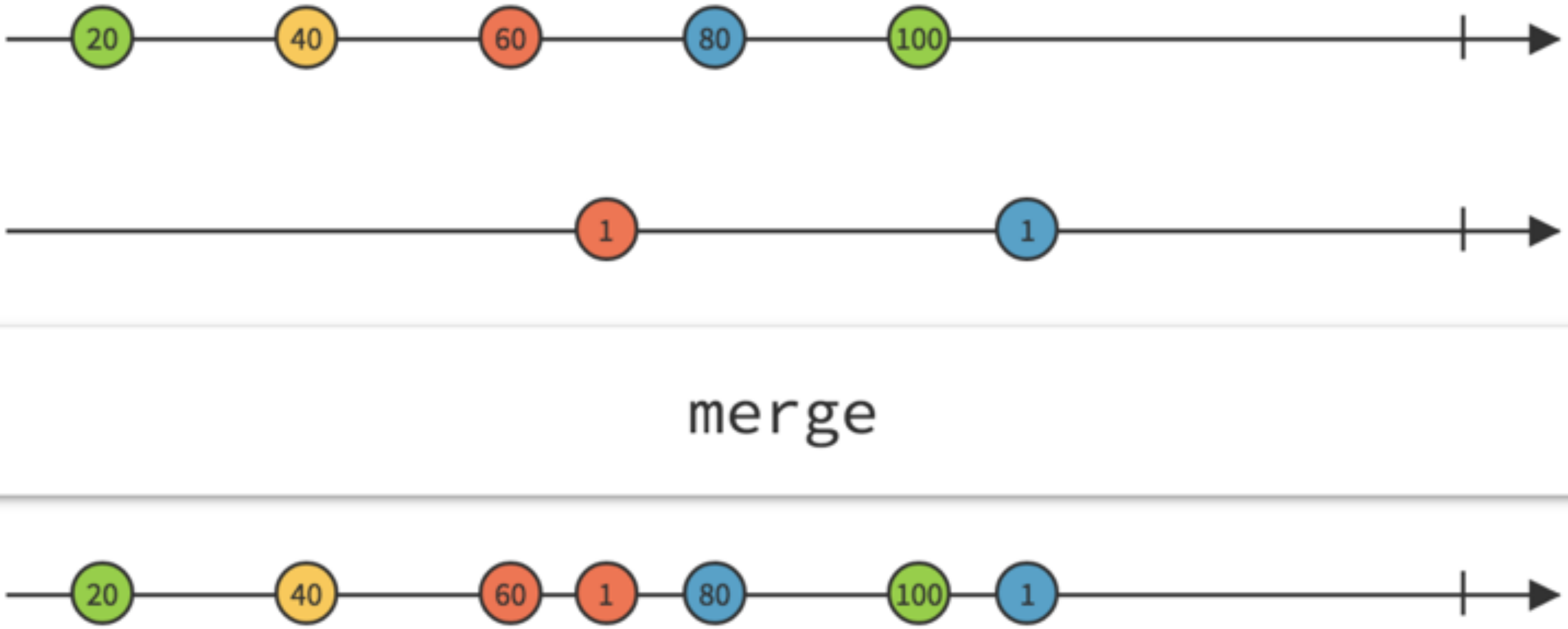
Combine latest



Zip



Merge



appendix

Observable Basics

```
var range = Rx.Observable.range(1, 10) // 1, 2, 3 ...
    .filter(function(value) { return value % 2 === 0; })
    .map(function(value) { return "<span>" + value + "</
span>"; })
    .takeLast(1);

var subscription = range.subscribe(
    function(value) { console.log("last even value: " +
value); });
// "last even value: <span>10</span>"
```

Observable Creation

```
Rx.Observable.fromArray([1, 2, 3]);

Rx.Observable.fromEvent(input, 'click');
Rx.Observable.fromEvent(eventEmitter, 'data', fn);

Rx.Observable.fromCallback(fs.exists);
Rx.Observable.fromNodeCallback(fs.exists);
Rx.Observable.fromPromise(somePromise);
Rx.Observable.fromIterable(function*() {yield 20});
```

COLD is when your observable creates the producer

```
// COLD
var cold = new Observable((observer) => {
  var producer = new Producer();
  // have observer listen to producer here
});
```

HOT is when your observable closes over the producer

```
// HOT
var producer = new Producer();
var hot = new Observable((observer) => {
  // have observer listen to producer here
});
```

COLD is when Producers created *inside*

```
var cold = new Observable((observer) => {
  const source = new Observable((observer) => {
  const socket = new WebSocket('ws://someurl');
  socket.addEventListener('message', (e) => observer.next(e));
  return () => socket.close();
});
```

HOT is when your Producers created *outside*

```
const socket = new WebSocket('ws://someurl');

const hot = new Observable((observer) => {
  socket.addEventListener('message', (e) => observer.next(e));
});
```

anything that subscribes to `source` will share the same WebSocket instance. It wil
effectively multicast to all subscribers now.

Some Cool Stuff on Observables

```
.bufferWithTime(500)

.pausable(pauser), .pausableBuffered(..)

.repeat(3)

.skip(1), skipUntilWithTime(..)

.do() // for side-effects like logging

.onErrorResumeNext(second) // resume with other obs

.window() // project into windows

.timestamp() // add time for each value

.delay()
```

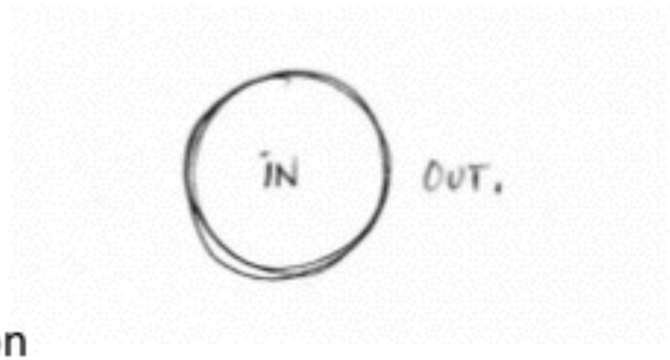
FP and OOP are orthogonal in nature.

Reactive programming deals with data. Ultimately this is a special case of event-driven programming.

reactive vs actor

Internal vs External Iteration

```
for (Employee e : employees) {
    e.setSalary(e.getSalary() * 1.03);
}
```



- – Inherently serial
- – Client has to manage iteration
- – Nested loops are poorly readable

```
employees.forEach(e -> e.setSalary(e.getSalary() * 1.03));
```

Not only a syntactic change!

- + Library is in control → opportunity for internal optimizations as parallelization, lazy evaluation, out-of-order execution
- + More *what*, less *how* → better readability
- + Fluent (pipelined) operations → better readability
- + Client can pass behaviors into the API as data → possibility to abstract and generalize over behavior → more powerful, expressive APIs

Referential transparency

An expression **e** is *referentially transparent* if for all programs **p**, all occurrences of **e** in **p** can be replaced by the result of evaluating **e**, without affecting the observable behavior of **p**

A function **f** is *pure* if the expression **f(x)** is referentially transparent for all referentially transparent **x**

RT wins

- Under a **developer** point of view:
  - ✓ Easier to reason about since effects of evaluation are purely local
  - ✓ Use of the *substitution model*: it's possible to replace a term with an equivalent one

- Under a **performance** point of view:
  - ✓ The JVM is free to optimize the code by safely reordering the instructions
  - ✓ No need to synchronize access to shared data
  - ✓ Possible to cache the result of time consuming functions (*memoization*), e.g. with

```
Map.computeIfAbsent(K key,
      Function<? super K,? extends V> mappingFunction)
```

Immutability

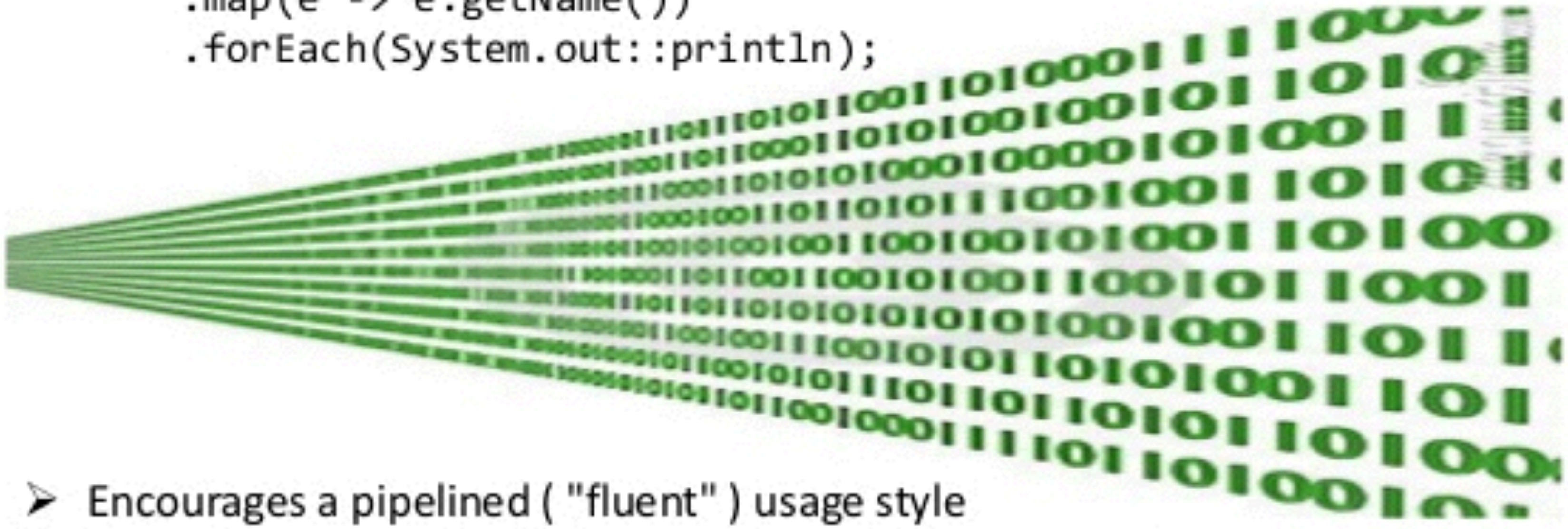
- Immutable objects can be shared among many threads exactly because none of them can modify it

- In the same way immutable (persistent) data structures can be shared without any need to synchronize the different threads accessing them

Characteristic	Imperative approach	Functional approach
Programmer focus	How to perform tasks (algorithms) and how to track changes in state.	What information is desired and what transformations are required.
State changes	Important.	Non-existent.
Order of execution	Important.	Low importance.
Primary flow control	Loops, conditionals, and function (method) calls.	Function calls, including recursion.
Primary manipulation unit	Instances of structures or classes.	Functions as first-class objects and data collections.
Mutability	Mutable	Immutable

Streams - Efficiency with laziness

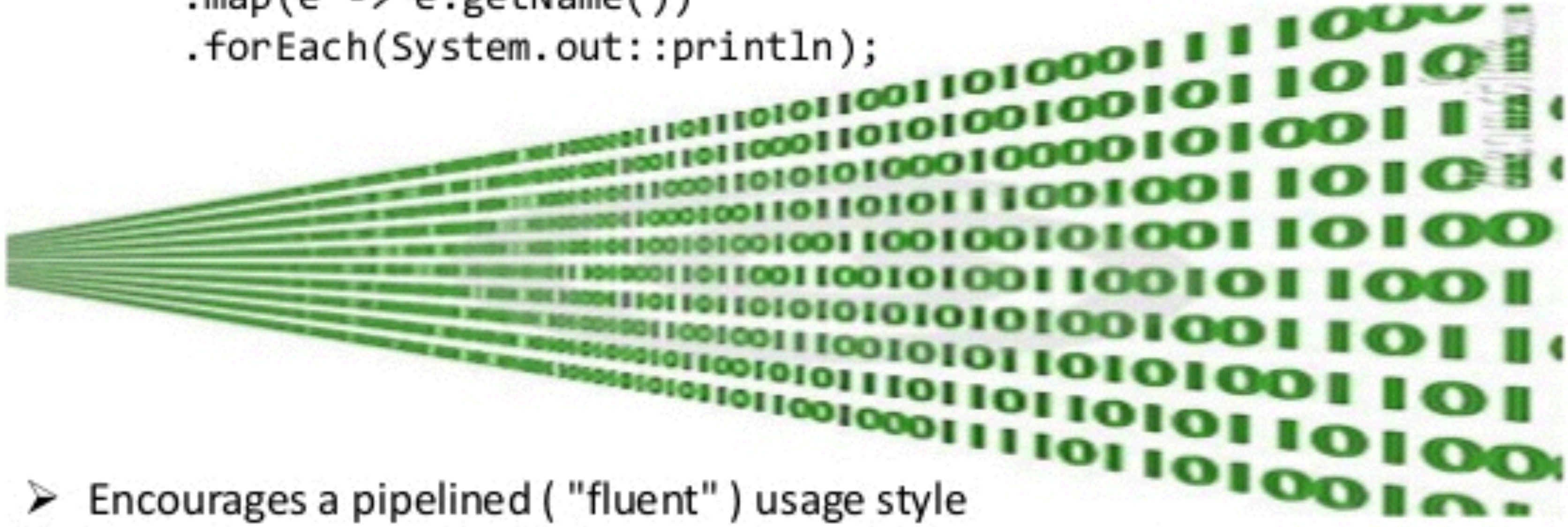
```
employees.stream()
          .filter(e -> e.getIncome() > 50000)
          .map(e -> e.getName())
          .forEach(System.out::println);
```



- ➢ Encourages a pipelined ("fluent") usage style
- ➢ Operations are divided between intermediate and terminal
- ➢ Lazy in nature: only terminal operations actually trigger a computation

Streams - Efficiency with laziness

```
employees. parallelStream()
          .filter(e -> e.getIncome() > 50000)
          .map(e -> e.getName())
          .forEach(System.out::println);
```



- Encourages a pipelined ("fluent") usage style
- Operations are divided between intermediate and terminal
- Lazy in nature: only terminal operations actually trigger a computation

... and parallelism for free